UNIVERSITÄT
DES
SAARLANDES

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

Towards Principled Dynamic Analysis on Android

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

vorgelegt von
Oliver Schranz

Saarbrücken, 2020

Tag des Kolloquiums:	08.06.2021
Dekan der Fakultät:	Prof. Dr. Thomas Schuster

Prüfungsausschuss:

Vorsitzender des Prüfungsausschusses:	Prof. Dr. Raimund Seidel
Erstgutachter:	Prof. Dr. Michael Backes
Zweitgutachter:	Prof. Dr. Christian Rossow
Akademischer Beisitzer:	Dr. Robert Künnemann

Abstract

The vast amount of information and services accessible through mobile handsets running the Android operating system has led to the tight integration of such devices into our daily routines. However, their capability to capture and operate upon user data provides an unprecedented insight into our private lives that needs to be properly protected, which demands for comprehensive analysis and thorough testing. While dynamic analysis has been applied to these problems in the past, the corresponding literature consists of scattered work that often specializes on sub-problems and keeps on re-inventing the wheel, thus lacking a structured approach. To overcome this unsatisfactory situation, this dissertation introduces two major systems that advance the state-of-the-art of dynamically analyzing the Android platform. First, we introduce a novel, fine-grained and non-intrusive compiler-based instrumentation framework that allows for precise and high-performance modification of Android apps and system components. Second, we present a unifying dynamic analysis platform with a special focus on Android's middleware in order to overcome the common challenges we identified from related work. Together, these two systems allow for a more principled approach for dynamic analysis on Android that enables comparability and composability of both existing and future work.

Zusammenfassung

Die enorme Menge an Informationen und Diensten, die durch mobile Endgeräte mit dem Android Betriebssystem zugänglich gemacht werden, hat zu einer verstärkten Einbindung dieser Geräte in unseren Alltag geführt. Gleichzeitig erlauben die dabei verarbeiteten Benutzerdaten einen beispiellosen Einblick in unser Privatleben. Diese Informationen müssen adäquat geschützt werden, was umfassender Analysen und gründlicher Prüfung bedarf. Dynamische Analysetechniken, die in der Vergangenheit hier bereits angewandt wurden, fokussieren sich oftmals auf Teilprobleme und reimplementieren regelmäßig bereits existierende Komponenten statt einen strukturierten Ansatz zu verfolgen.

Zur Überwindung dieser unbefriedigenden Situation stellt diese Dissertation zwei Systeme vor, die den Stand der Technik dynamischer Analyse der Android Plattform erweitern. Zunächst präsentieren wir ein compilerbasiertes, feingranulares und nur geringfügig eingreifendes Instrumentierungsframework für präzises und performantes Modifizieren von Android Apps und Systemkomponenten. Anschließend führen wir eine auf die Android Middleware spezialisierte Plattform zur Vereinheitlichung von dynamischer Analyse ein, um die aus existierenden Arbeiten extrahierten, gemeinsamen Herausforderungen in diesem Gebiet zu überwinden. Zusammen erlauben diese beiden Systeme einen prinzipienorientierten Ansatz zur dynamischen Analyse, welcher den Vergleich und die Zusammenführung existierender und zukünftiger Arbeiten ermöglicht.

Background of this Dissertation

This dissertation is based on the two major papers mentioned in the following, where I have been the main author.

The **ARTist** project [P1] evolved from my master thesis (TaintARTist), originally proposed by Philipp von Styp-Rekowski. TaintARTist introduced basic taint tracking code into the Android Runtime (ART) compiler and first described the compiler's internals. However, it targeted a specially prepared intermediate Android version based on the current master branch at the time (between Android 5 and 6) and required a custom kernel to work properly. In contrast, ARTist advances the underlying concept from a pure taint tracking implementation to a full-fledged instrumentation framework where taint tracking is only one of multiple use cases. It targets the stable Android versions 6, 7, and 7.1, and comes with a full ecosystem of tools from module development to large-scale evaluations. Sebastian Weisgerber, Parthipan Ramesh and Alexander Fink contributed to the implementation of the prototype and corresponding tools. Sven Bugiel was involved in general writing tasks. All authors performed reviews of the paper.

The original idea behind **Troop** to fuzz-test the Android middleware has been discussed at the Information Security and Cryptography Chair for a while with Sven Bugiel as the driving force behind it. Since we realized that ARTist solved a major conceptual challenge (coverage feedback), I bootstrapped this project by retrofitting the evaluation tool that was built for ARTist to become the foundation of our dynamic analysis platform. The text in this dissertation goes beyond the paper [P2] insofar that it *additionally* contains a more detailed and thorough systematization of related work that was originally part of an unpublished Systematization of Knowledge paper based on [P2]. Sebastian Weisgerber, Kai Greshake, Jonas Cirotzki, Parthipan Ramesh, and Alexander Fink contributed to the implementation of the prototype. Erik Derr contributed static analysis results that are utilized as a part of our dynamic analysis. Sven Bugiel was involved in general writing tasks. All authors performed reviews of the paper.

Author's Papers for this Thesis

[P1] Backes, M., Bugiel, S., Schranz, O., Styp-Rekowsky, P. von, and Weisgerber, S. ARTist: The Android Runtime Instrumentation and Security Toolkit. In: *IEEE EuroS&P'17*.

[P2] Schranz, O., Weisgerber, S., Derr, E., Backes, M., and Bugiel, S. Towards a Principled Approach for Dynamic Analysis of Android's Middleware. Under Submission.

Further Contributions of the Author

[S1] Backes, M., Bugiel, S., Hammer, C., Schranz, O., and Styp-Rekowsky, P. von. Boxify: full-fledged app sandboxing for stock android. In: *USENIX SEC'15*.

[S2] Backes, M., Bugiel, S., Schranz, O., and Styp-Rekowsky, P. von. Boxify: bringing full-fledged app sandboxing to stock android. *USENIX ; login* 41, 2 (2016).

[S3] Huang, J., Schranz, O., Bugiel, S., and Backes, M. The art of app compartmentalization: compiler-based library privilege separation on stock android. In: *ACM CCS'17*.

Technical Reports of the Author

[T1] Schranz, Oliver. *ARTist - A Novel Instrumentation Framework for Reversing and Analyzing Android Apps and the Middleware.* URL: https://i.blackhat.com/us-18/Thu-August-9/us-18-Schranz-ARTist-A-Novel-Instrumentation-Framework-for-Reversing-and-Analyzing-Android-Apps-and-the-Middleware-wp.pdf (Accessed: July 3, 2020).

Acknowledgments

Pursuing a Ph.D. is never done alone, so I want to thank those people that made all of this possible and accompanied me on this unique journey.

First, I want to thank my supervisor Michael Backes who introduced me to the topic of security and gave me the opportunity to concurrently explore both worlds, academia and industry, which eventually led me on my current path. I am very grateful for the support and mentoring I received from him since my bachelors so that I could conduct a Ph.D. at the Information Security & Cryptography group at CISPA together with so many nice and talented peers. I feel deep gratitude towards the amazing people at CISPA, in particular in Michael's group, that I had the chance to share so many unique experiences with, from traveling to conferences to nightly nerf gun fights before deadlines, laser tag, bad movie nights, and many more. The unique working atmosphere to help each other and learn together is one of the major reasons I enjoyed being a Ph.D. student so much, thank you all for making this possible.

Special thanks go to my very talented and enthusiastic co-authors without whom this dissertation would not have been possible, namely (in alphabetical order) Sven Bugiel, Erik Derr, Christian Hammer, Jie Huang, Philipp von Styp-Rekowsky and Sebastian Weisgerber. I had a lot of fun and learned so much from working with you on Android security topics and beyond.

I also had the pleasure to share an office with such great "roomies" over the period of the last 5 years, Sven Bugiel, Milivoj Simeonovski, Sebastian Weisgerber, Philipp von Styp-Rekowsky, Jie Huang, Tin Nguyen and Yang Zou. Thank you for all the interesting discussions and off-topic chats that kept me motivated.

A very special thank you goes to Sven Bugiel who has been a constant in my whole academic life. From supervising my bachelor thesis to helping with every single paper I ever worked on, he always found the time to share advice and lend a hand. This dissertation would not have been possible without him.

Also, I want to thank my friends for always being there to make these past years memorable and enjoyable. Thank you Lukas, Jana, Jana, Jilles, Anna, Hannes, David, Ben, Julia, Jonas, Sophie, Andi, Johannes, Lisa, Caro, Jenni, Pascal, Maike, Jeanette, Fabian, Frank, Mich, Franzi, Dominik, Sebastian and Julian.

Another big thank you goes to Johannes, Andi and Claudia for proof-reading the whole thesis. Moreover, I would like to express my gratitude towards Christian Rossow for being the second examiner for this dissertation. In the context of my dissertation defense, I would like to additionally thank Raimund Seidel for chairing the examination board and Robert Künnemann for writing the protocol.

Last but not least, I want to sincerely thank my family. I am incredibly grateful to my parents Sabine and Wilhelm who always unconditionally supported me on this path. Without their help, none of this would have been possible. Also, I want to thank my grandparents, my brother and sisters, and the whole family that I am so grateful to have. You always have my back and I am so happy to be a part of this family.

Finally, I want to thank Claudia. Throughout all the setbacks, hurdles and sacrifices that a Ph.D. comes with, you were always at my side to cheer me up and keep me running. Thank you for those wonderful 10 years that we have shared already.

Contents

List of Figures

List of Tables

1 Introduction

With the introduction of the Android operating system (OS) as an open and extensible ecosystem, Google revolutionized the mobile market by—for the first time—bringing together a wide range of stakeholders. All those different actors, from private and corporate app developers to device manufacturers and hardware vendors, platform architects, as well as consumers, come with a multitude of requirements. In response to these diverse use cases, device classes range from cheap feature phones to high-end flagships that push the boundaries of the smartphone form factors in terms of performance and new hardware components. At the time of writing, Android dominates the mobile market with a share of ~74% [103] (July 2020). What started as a specialized operating system for resource-constrained mobile devices is now being adapted to a multitude of device families, including smartwatches, TVs, and set-top boxes, Internet-of-Things (IoT) devices, and soon even automotives. This leap towards new platforms and form factors was fueled by Android's modern operating system design and the underlying principle of *appification* [8] where functionality is successively subdivided into specialized applications (apps) that interact with a rich environment provided by the OS and other apps. It is believed that Android's feature-rich middleware that exposes platform capabilities and features in an easily accessible way was a driving factor to reduce the barrier of entry for participating in this ecosystem. In the context of this thesis, two implications of these design decisions are particularly relevant.

First, as a consequence of the low barrier of entry, developers created an overwhelming amount of applications that is made available to end users via app markets (e.g., Google Play Store). In contrast to iOS that requires membership in Apple's developer program to be able to actually deploy apps, and therefore favors corporate development, everybody [1] can upload apps to the Google Play Store, which includes hobby programmers but also malicious actors. While having completely different motivations, both of the latter groups regularly increase the attack surface by (accidentally or intentionally) introducing potential vulnerabilities into end user devices.

Second, exposing such a rich functionality to third-party apps leads to a complex and ever-growing middleware that needs to balance the increasing demand for new features with the inherent requirement to properly shield the exposed private information and other protected resources (e.g., access to hardware sensors). While many of those security measures are traditionally enforced within the kernel, Android's platform design requires to spread those between the underlying Linux kernel and the highly-privileged middleware running in user-space. One prime example is the enforcement of Android permissions — a central access control mechanism that guards access to protected resources — out of which some are enforced in the kernel via user group membership (e.g., Internet access and writing to external storage) while others are exclusively represented and checked in the middleware (e.g., broadcast and content provider permissions). This complexity is further amplified by Android's fragmentation problem that stems from the multitude of available Android versions and customizations. Each new version of the Android Open Source Project (AOSP) comes with changes and additions to the middleware, ranging from simple bug fixes to the addition of completely new services that control access to new resources such as built-in hardware.

[1] At the time of this writing, there is a one-time fee of 25$ to access Google's developer console [70]. Uploading apps is free afterwards.

Vendors further customize these Android versions by including even more functionality, thereby creating a very heterogeneous device landscape. In addition, the middleware evolved as a wild technology mix that involves services running in many inter-connected components written in different programming languages. Altogether, this makes the Android middleware a notoriously hard target for automated analysis.

The large amount of code that renders manual inspection infeasible, as well as the inherent need to protect both, the application layer and the middleware supporting it, fostered two major lines of work in the literature that approach the problem by means of automated analysis.

The first line of work utilizes static analysis, which has proven to be a valuable tool for analyzing such complex systems at scale without relying on large amounts of actual devices. In the literature, it has been applied successfully to problems like bug and vulnerability discovery [90, 83], detecting inconsistent enforcement of security measures [123, 84, 6, 4], permission mapping [22, 28, 5], and many more. Further investigations into this body of literature reveals that most approaches for the app and middleware layer build upon a common base that is one of two predominant static analysis frameworks: *Soot* [131] or *Wala* [92]. This shared foundation allowed the community not only to reproduce and compare past results, but also to build on top of them to avoid re-inventing the wheel and focus their contribution on the new analysis.

The second line of work, which focuses on applying dynamic analysis approaches to those problems, tries to overcome known disadvantages of static analysis, such as over-approximation, by rooting analysis results in actual executions. However, in contrast to the static analysis branch, these works often create highly-customized toolchains that are optimized for the current use-case only and are not made available openly. Consequently, the community struggles to re-use or even evaluate those artifacts and their reported results, which leads to the unsatisfactory situation of duplicate work and incomparable results. While the employed dynamic analyses are promising approaches, they cannot realize their full potential due to the lack of a shared foundation. We argue that a more principled approach is needed to steer the community towards comparable and reproducible results, created from re-usable components on a shared platform.

This thesis aims at advancing this line of work by introducing two frameworks to realize dynamic analyses for Android's middleware and apps.

First, we propose ARTist — the Android Runtime Instrumentation and Security Toolkit — as a compiler-based instrumentation framework that allows to create dynamic analyses that can efficiently instrument its targets to, e.g., receive runtime feedback or patch obstacles in the code. In contrast to existing approaches, ARTist preserves the application signature, allows for fine-grained instrumentation in contrast to method hooking, and is fully automated at the same time.

Second, based on a thorough requirements analysis of existing work in this area, we introduce Troop as a common framework that can re-instantiate and unify dynamic analyses of the Android middleware by combining them with the modular solutions we created to overcome the shared challenges in the literature. Additionally, evaluating these analyses on a unified platform — for the first time — allows the community to properly reproduce and compare the corresponding results, and in the next step have a solid foundation to discuss and move towards *optimal* solutions for the mentioned challenges.

Summary of Contributions:

ARTist

ARTist is an instrumentation framework for Android apps and middleware components that allows for efficient modifications with instruction-granularity. It is designed to preserve Android's application signatures and can be deployed on the application layer with minimal invasiveness towards the operating system, thereby filling a gap in the current design space of instrumentation frameworks. It is built as an extension to the *dex2oat* ahead-of-time compiler that was introduced with the Android Runtime in Android 5, and deployed either via a simple app or as a part of a custom Android fork. Using its module SDK, researchers can use it to, e.g., track information flow (e.g., taint tracking), provide coverage information for feedback-driven test generators (i.e., fuzzers), enforce fine-grained dynamic permissions, and to reverse-engineer and introspect applications. We further created a tool to automate large-scale app instrumentation and testing, which we used to evaluate the ARTist framework itself and two use cases we implemented. The whole toolchain is open source.

Troop

Troop is a full-fledged dynamic analysis platform for Android's middleware and apps that, based on a thorough study of related work, provides primitives to re-instantiate, evaluate and compare dynamic analyses from the literature. It serves as a common foundation that allows to solve recurring problems of dynamically analyzing core Android components in a modular and re-usable way in order to streamline and support research on this topic. Our platform allowed us to implement two completely different use cases from related work. The first case study, a large-scale evaluation of vulnerability discovery using three different fuzzers, led to the discovery of 11 flaws and provides valuable insights into whether known strategies from other fields can be applied effectively in the Android middleware scenario. The second case study serves as a proof-of-concept that dynamic permission mapping, which has been abandoned in favor of static analysis in the literature, is nonetheless an invaluable tool for coping with the over-approximation of those static approaches. The whole platform and toolchain will be open sourced to help establish reproducibility and comparability in this field.

Outline

The remainder of this thesis is structured as follows. In Chapter 2 we introduce the technical concepts underlying this dissertation. Chapter 3 introduces ARTist, our compiler-based instrumentation framework for Android. The Troop dynamic analysis platform is described in Chapter 4. We conclude this dissertation in Chapter 5 and give an outlook.

2 Technical Background

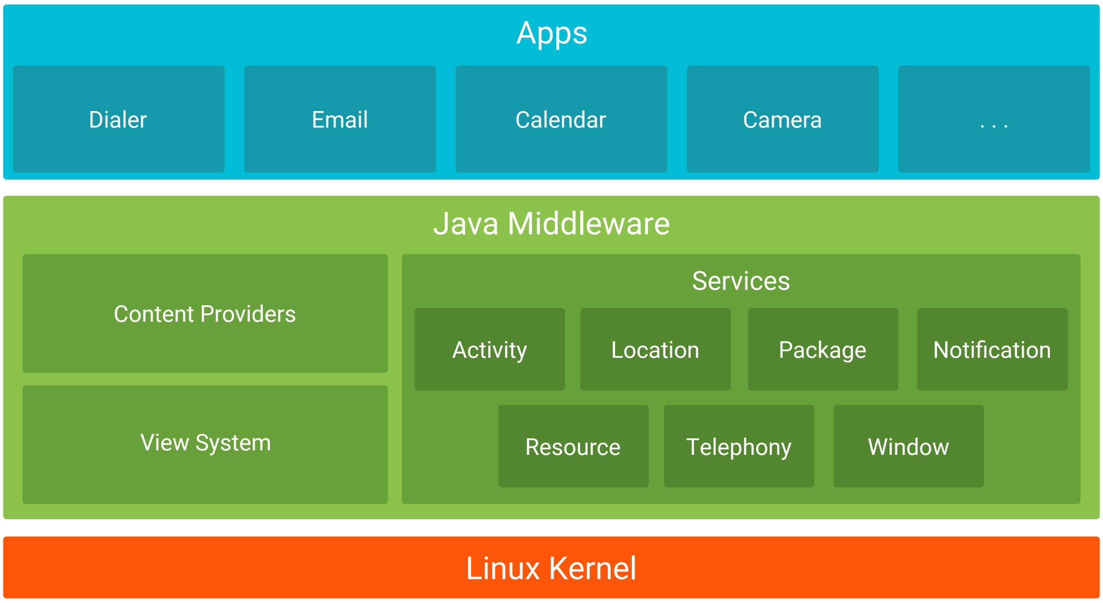


Figure 2.1: Overview of the Android software stack (adapted and simplified from [13]).

2.1 Android Background

The Android open source operating system is the center of a large ecosystem that powers a multitude of smart devices, ranging from smartphones and tablets to wearables, TVs, cameras, IoT devices, and soon even automotives. In 2020, Android leads the market for smartphones and other mobiles with a global share of ~74% [103] (July 2020). It is continuously developed and improved by Google through the *Open Handset Alliance* that releases new Android major versions once a year via the Android Open Source Project (AOSP). At the time of this writing, the most recent version is Android 10, which corresponds to API level 29. Given its vast popularity in the market and its open ecosystem, there is a large body of research that focuses on different aspects of security in the context of Android. This section provides the common technical background required for both of the following Chapters 3 and 4.

2.1.1 Software Stack

As depicted in Figure 2.1, Android consists of three major layers: the kernel, the middleware and the application layer. On the lowest level, it utilizes a customized Linux kernel that handles low-level tasks such as talking to device drivers, handling inter-process communication (IPC) or managing native Linux users. However, Android's core functionality is mostly implemented by the libraries and middleware on top. On the one hand, the Android Runtime, formerly Dalvik Virtual Machine [61] (DVM), compiles and executes all Java-based middleware components and apps, which provide most of the functionality that is exposed to third-party applications, such as access to location information, telephony services, notifications, and many more. For example, `ActivityManagerService` is handling the complete application lifecycle and

PackageManagerService manages their installation process. On the other hand, the middleware also contains a set of core libraries and natively implemented services and daemons, such as the media framework. On top of the middleware is the application layer that consists of system apps as well as third-party apps. These applications extend the platform's capabilities to the user by utilizing its APIs to build their own functionality, thereby forming the primary interaction channel of the user with their device. Apps are usually installed and updated from central software repositories called app stores (e.g., Google Play store or Amazon Appstore) but app packages can also be installed directly from the file system (side-loading).

Systemserver. The so-called Systemserver is a central component of the middleware that hosts most of the system services exposed to applications (e.g., location, notifications). During the boot process, it is the first Java-based process that is started with the runtime. When launched, the Systemserver forms a large process with every service being instantiated as a thread that exposes its functionality via IPC. Overall, the Systemserver exposes more than 2000 APIs[1]. Its utmost importance for the system's functionality combined with this large attack surface makes it a prime target for researchers to analyze for potential bugs and vulnerability. A restart of the Systemserver triggers a so-called soft reboot where the whole application layer is shut down until the Systemserver process is back online. In order to protect against deadlocks and other situations that prevent core services from responding properly, it comes with a dedicated Watchdog thread that connects to all hosted services every 30 seconds and kills the whole process if one does not answer after a minute.

Mediaserver & Native System Services. Besides the Systemserver, the so-called mediaserver and other native services support the application and middleware layer with, e.g., media related functionality such as access to audio, camera, and codecs. While those have proven to be notorious sources of high-severity vulnerabilities, one of the more popular examples being the stagefright exploits [82, 81], Google has since reduced the potential impact of such attacks by sequentially splitting those privileged entities into smaller processes using increasingly restricted SELinux [76] policies[2].

2.1.2 Inter-Process Communication (IPC)

In contrast to stock Linux that provides multiple ways for IPC, Android primarily uses light-weight and fast IPC via its *Binder* component. A client can utilize it for remote procedure calls by directly interfacing with the binder driver via the /dev/binder device[3]. The binder driver then copies the payload from client to server and executes the requested functionality in a thread there. Once the result is ready, it is transmitted back again via the binder driver so that the client thread receives it. Unless explicitly

[1] The exact number varies depending on Android version, vendor and whether the device is physical or emulated.

[2] The evolution of mediaserver and the corresponding sandboxing efforts are documented in a blog post [75] from the Android team.

[3] Starting from Android 8, there are now multiple such endpoints depending on whether the client is a framework, vendor, or hardware service [62].

flagged as one-way, this operation blocks the caller until a result is ready, which allows for synchronous procedure calls across the process boundaries. A central entity called the service manager is used to register new services and can be queried for all services in the system that are available via binder. In order to abstract from the binder protocol, Android system components (e.g., middleware services) provide proxy code that can be used on the client side to talk to the service manager and specific services without having to deal with the underlying low-level details of the communication. The Android Interface Definition Language (AIDL [46]) even automates the generation of client-side proxies and server-side stubs for Java-based services. Additionally, Android provides a set of high-level IPC mechanisms on top of binder, such as sending intents, using messengers, or accessing content providers via the content resolver interface.

2.1.3 Applications

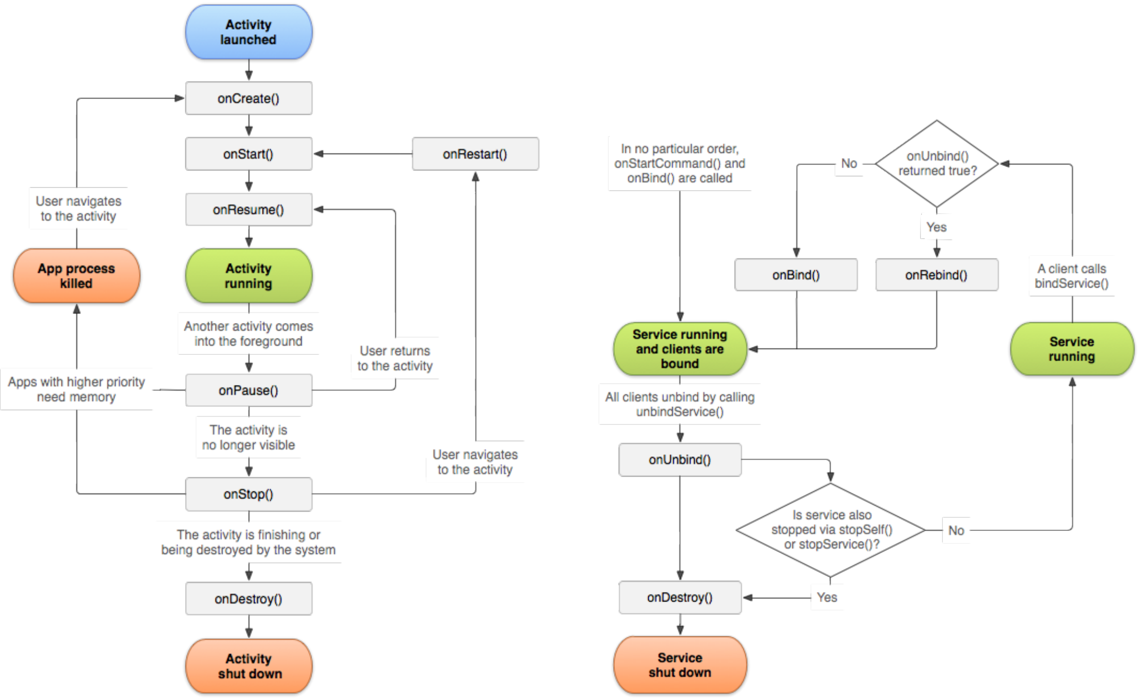
In this *appified* ecosystem, apps are the primary points of interaction with the user. They are responsible for implementing new features and functionality, based on Android's APIs, that the users interact with using, e.g., graphical user interfaces, notifications, or voice commands. In order to create apps, developers nowadays either use a combination of Java, Kotlin, and native compiled code, or employ cross-platform frameworks such as Flutter [67] or Xamarin [102]. Apps are based on four main components:

Activities implement event-driven interaction with users. On most devices, they display a graphical interface that drives the execution of the app through user interaction, but on display-less platforms, such as Android Things for IoT, they rely on other interaction mechanisms. Activities are managed by the Systemserver (i.e., the `ActivityManagerService`) and follow a strict lifecycle, as depicted in Figure 2.2a.

Services perform long-running tasks either in the foreground (e.g., play music) or in the background (e.g., download data). These are often used to avoid heavy computation in *Activities* that would slow down the user interface. Optionally, services can expose functionality to other applications by using the same mechanisms and tools (i.e., AIDL) as services in the Systemserver. The lifecycle of such services is shown in Figure 2.2b.

Content Providers abstract away and shield access to an app's content, such as downloaded files, images, databases, or files on disk. They are mostly used to implement create, read, update, and delete (CRUD) operations in a database-like interface. Similarly to services, they can be exposed to other apps as well. Apps can find system or third-party providers using a centralized content resolver that queries the `ActivityManagerService` for matching providers. Android provides support for fine-grained access control for the data managed by a content provider through permissions (e.g., on a file or database schema level).

Broadcast Receivers implement listeners and callback handlers for events generated by the system or other applications. Receivers declare the type of broadcast they are interested in by means of an intent filter and the system will instantiate them


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down
In no particular order, onStartCommand() and onBind() are called
No
onUnbind() returned true?
Yes
onBind()
onRebind()
A client calls bindService()
Service running and clients are bound
Service running
All clients unbind by calling unbindService()
onUnbind()
Is service also stopped via stopSelf() or stopService()?
No
Yes
onDestroy()
Service shut down

(a) The full lifecycle of Android activities. Source: [15].

(b) The full lifecycle of a bound service. Source: [11].

on-demand when a matching broadcast intent is available. Popular examples of system-wide broadcasts that are consumed by app broadcast receivers are the *boot completed* or *battery low* events.

2.1.4 Permissions

Permissions are an integral part of Android's security architecture that guard access to a multitude of protected resources, such as location information, the user's contacts and calendar, placing phone calls, or opening network sockets. The enforcement happens on a Linux user ID basis (UID), which is unique to each app on a device[4]. At install time, all permissions that an app wants to use are parsed from its manifest and requests for permissions that are not part of this initial set are automatically denied. The only way to add new permissions to an app is to add them to its manifest for the next app update. A particular set of internal and dangerous permissions, however, is only granted to system apps that are included in the Android OS image. Before the introduction of runtime permissions in Android 6, all requested permissions were automatically granted during the installation process and there was no way for the user to remove them again without uninstalling the application. Since Android 6, permissions are requested ad hoc and need to be confirmed by the user via a protected system UI component, and some can even be revoked at any time through the settings app. While the kernel enforces lower-level access, such as for sockets, the middleware comes with an own

[4] With the exception of the `sharedUserId` feature that was deprecated in API 29 [12].

implementation of permissions that is managed by the `PackageManagerService`, `AppOpsService`, and `PermissionController`, depending on the concrete Android version. However, there is no centralized enforcement point within the Systemserver. Exposed service APIs are responsible to enforce their own permissions and while many follow the fail-early principle of checking permissions in the beginning of the function, in some cases the enforcement is context- or parameter-sensitive. Currently, no official mapping from Android APIs to the permissions required to call them is available, hence the current process of determining the set of required permissions for an application is based on trial and error during the development phase.

2.2 Dynamic Analysis Primer

Static and dynamic analysis are two techniques that follow a fundamentally different approach to analyze a target. In general, static analysis refers to techniques that do *not* execute the target but inspect it using offline information. Often, this includes representing the target's code in a format that allows to reason about, e.g., dependencies and control flows. It is often used where the execution of targets is either not feasible (i.e., because of size, speed, or other limitations) or even dangerous (e.g., dissecting malware). A key property of such analyses is the over-approximation of findings, since by definition all possible (and in some cases even impossible) execution paths can be reasoned about. Dynamic analysis, in contrast, ties its results to actual executions of the target. It employs introspection techniques to learn about concrete execution traces as they appear and reasons about them. Compared to static analysis, it trades completeness for soundness. Under certain assumptions[5], results are sound because they actually manifest in real executions. Completeness, however, would require to generate a set of inputs that triggers every single part and combination of a target's functionality in order to catch all possible edge cases. The latter in particular is the problem underlying many well-studied challenges for dynamic analysis, such as maximizing coverage measures while dealing with implementation details of real executions, such as statefulness and parallelism.

In terms of scalability, static and dynamic analysis suffer for different reasons. While static analysis can often benefit more from increased computational resources such as more CPU time, RAM and disk space, the required computation is often based on the possible states of a target. Since the amount of possible states tends to be exponential in the target's code complexity, this is often referred to as the *state explosion* problem. Dynamic analysis avoids this by only reasoning about the current state of an execution trace, but scaling heavily depends on the kind of target under test. In our scenario, apps and middleware components are heavily depending on the Android system so that scaling dynamic analysis requires large amounts of devices, either physical or virtual.

[5] E.g., perfect introspection capabilities that do not change the target's behavior.

3
ARTist

The Android Runtime Instrumentation and Security Toolkit

3.1 Motivation

The Android OS has become a popular subject of the security research community over the last few years. Among the different directions of research on improving Android's security, a dedicated line of work has successfully investigated how instrumentation of the interpreter (i.e., Dalvik virtual machine) can be leveraged for security purposes. This line of work comprises influencing works such as TaintDroid [49] for analyzing privacy-relevant data flows within applications, AppFence [89] for protecting the end-users' privacy, Moses [116] for domain isolation, or Spandex [39] for password tracking, just to name a few.

However, with the release of Android 5 Lollipop, Google made a large technological leap by replacing the interpreter-based runtime with an on-device, ahead-of-time compilation of app and system code to platform-specific native code that is executed in the new Android Runtime. While this leap did not affect the app developers, it broke legacy compliance of all of the previously mentioned security solutions that relied on instrumentation of the DVM and restricts them to Android versions prior to Lollipop. In fact, it has left the security research community with two choices for carrying on work that relies on instrumented runtimes: resorting to binary or bytecode rewriting techniques [43, 86] or adapting to the novel but uncharted on-device compiler infrastructure. While the former has been studied already in the literature, it comes with a set of disadvantages on Android, such as breaking the application signature or being platform dependent, that still leave room for improvement. The latter, however, first and foremost requires a thorough study of the new runtime's internals, more specifically of its on-device compiler *dex2oat*. Furthermore, these observations raise the question whether the new runtime can even support the security community beyond what was possible within the frame of the DVM interpreter. For the first time in the development of Android, having an on-device compiler allows to draw a connection to existing compiler-based security literature, which is a dedicated line of work that has been thoroughly studied on commodity systems but not yet on mobile systems. Therefore, this work also aims at investigating whether, going forward, we can bring compiler-assisted security solutions to Android.

3.2 Exploring the Android Runtime's New Compiler

This section will provide the necessary background on the internals of the Android Runtime as required for this chapter. The analysis is based on Android 6 Marshmallow that represents the first major OS version that made the *Optimizing* backend — our focus for this work[1] — the default. Even though the Android source code is publicly available as part of the Android Open Source Project, little attention has yet been given to ART from a security researcher's perspective. Paul Sabanal had an early look [117] at the Android Runtime right after its silent introduction as a developer option on Android 4.4 KitKat. Besides providing information on the ART executable file formats, the paper discusses the idea of hiding rootkits in framework or app code, assuming root access

[1] Section 3.2.4 explains why we specifically chose to utilize the *Optimizing* backend.

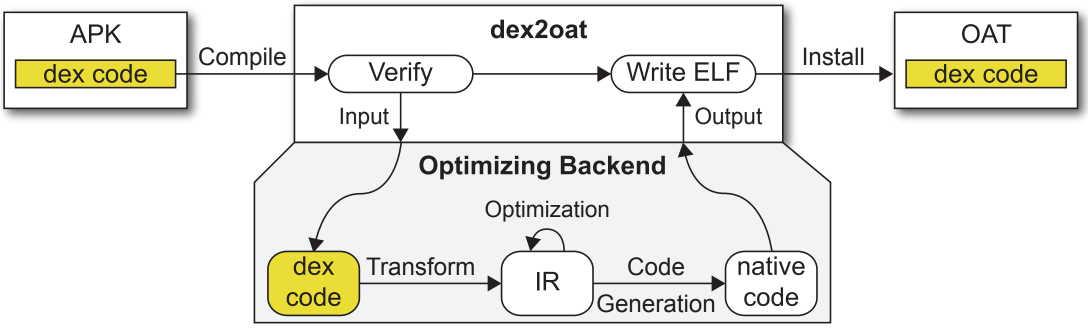


Figure 3.1: A high-level overview of the *dex2oat* compiler using the *Optimizing* backend including the transformation to the IR, optimizations, and native code generation.

has already been granted. However, especially in its early phase, the Android Runtime has undergone frequent changes, which, unfortunately, has made the corresponding documentation outdated by now.[2]

Android's new on-device compiler *dex2oat*, which is of particular interest for this work, is responsible for the validation of applications and their compilation to native code. It was designed from scratch to be highly flexible and of modular structure, providing numerous configuration possibilities, multiple compiler backends, and native code generators for supported Android platforms.

The general workflow of the compiler suite is depicted in Figure 3.1 and its steps will be explained in the remainder of this section.

Providing a full technical documentation of the entire compiler suite and all its intricacies is outside the scope of this work, therefore we only focus on those parts relevant to the upcoming chapters of this thesis.

3.2.1 Input File Format

As an input format, *dex2oat* expects the very same `dex` files that the DVM used to interpret. This strategical decision ensured that neither developers nor app store operators needed to adapt their code to ART. Developers still upload their apps as Android Application Package (APK) files that bundle the app's code with its resources. When a new app is installed on the device, *dex2oat* compiles the app's `dex` bytecode and the ART runtime executes the resulting `oat` file, which is completely transparent for the end user. Using this strategy, ART is still compatible with the old Android app base without enforcing a fallback to interpretation.

3.2.2 Output File Format

`oat` files are Android's new file format for compiled code that is loaded and executed by the Android Runtime. Even though the format was newly created for the Android platform, technically speaking `oat` files are specialized ELF shared objects that are

[2] E.g., there is a large version gap between the documented `oat` version 45 and the version 64 that we investigated.

loaded into processes, i.e., loading a compiled app into an application process resembles loading an (ELF) shared library into the process space of a dynamically linked executable. Besides the native code generated with *dex2oat*, `oat` files contain the complete original `dex` code, which is required to, e.g., fall back to interpretation mode during app debugging. `oat` files are still not officially documented, but they can be analyzed using standard ELF tools like *readelf* and, in more recent Android versions, a tool called *oatdump*.

3.2.3 Compilation

Before the actual compilation is performed, each input `dex` file is checked for validity. These checks are more extensive and stricter than those implemented in the DVM [78]. The compilation itself is done on a per-method base and can be parallelized. *dex2oat* completely delegates the actual compilation to one of its backends and only writes the results of the compilation to an `oat` file along with the original `dex` code. There are three compilation phases shared between all backends:

Transformation. A graph-based intermediate representation (IR) is created from the `dex` code. Depending on the concrete backend, multiple IRs are possible. While nodes in this graph representation typically resemble `dex` instructions, they are interlinked and extended with additional information that aid code analysis.

Optimization. Each backend provides its own set of optimization passes, ranging from very basic techniques to state-of-the-art algorithms. Given a populated IR graph, the code is optimized by running a subset of the implemented optimization passes. These passes require full access to the method graphs since optimizations might involve rewriting major parts of the code (e.g., inlining, dead code elimination). However, while these aim to produce more *efficient* code, the original functionality intended by the developer and therefore also the semantic consistency between the original `dex` code and the code compiled from it needs to be preserved.

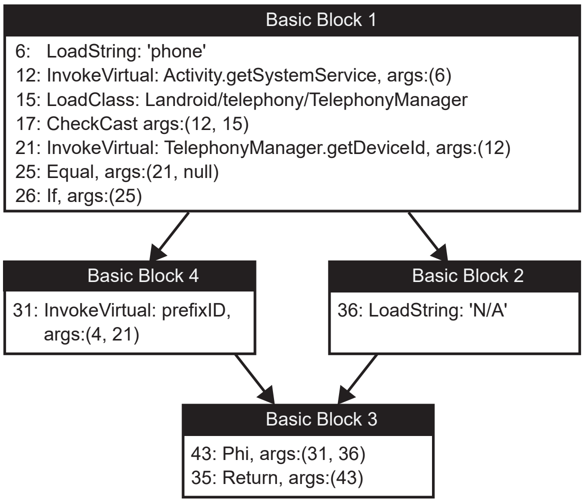
Native Code Generation. The IR nodes are transformed to native code using a code generator for the specific CPU architecture of the current platform, e.g., `arm` or `x64`. Again, there are major differences between the backends, such as the level of sophistication of the register allocation algorithm, which depends on the versatility of the underlying IR.

3.2.4 Backends

On an Android stock device running version 5 Lollipop or higher, *dex2oat* can choose between two different backends, *Quick* and *Optimizing*. Originally, there was a third backend called *Portable* that utilized LLVM [26] to lift dex bytecode to LLVM's bitcode, but it was discontinued before ever reaching a production-ready state. Although *Quick* was *dex2oat*'s default backend until Android 6, we focus on the newer *Optimizing* backend for the remainder of this thesis. This choice is not only motivated by the

```
public String getID() {

  TelephonyManager tm =
    getSystemService(TELEPHONY_SERVICE);

  String id = tm.getDeviceId();

  if(id != null) {
     id = prefixID(id);
  } else {
     id = "N/A";
  }

  return id;
}

public String prefixID(String id) {
  String prefix = "ID: ";
  String result = prefix + id;
  Log.d(TAG, prefix + id); // leak id!
  return id;
}
```

(a) Example code leaking the device's phone number.


Basic Block 1
6: LoadString: 'phone'
12: InvokeVirtual: Activity.getSystemService, args:(6)
15: LoadClass: Landroid/telephony/TelephonyManager
17: CheckCast args:(12, 15)
21: InvokeVirtual: TelephonyManager.getDeviceId, args:(12)
25: Equal, args:(21, null)
26: If, args:(25)
Basic Block 4
31: InvokeVirtual: prefixID, args:(4, 21)
Basic Block 2
36: LoadString: 'N/A'
Basic Block 3
43: Phi, args:(31, 36)
35: Return, args:(43)

(b) Generated IR in SSA form for the `getID()` method.

Figure 3.2: Transformation of Java code to the *Optimizing* IR.

fact that *Optimizing* is the default backend since Android 6, but also because *Quick* is essentially derived from the DVM's just-in-time compiler and lacks a sophisticated IR that can support state-of-the-art optimizations. However, *Optimizing* was designed completely from scratch and its internal structure and design are still not officially documented. In Figure 3.1 the compilation steps of *Optimizing* are depicted.

3.2.5 Optimizing: Intermediate Representation

Our insights into *dex2oat*'s *Optimizing* backend are mainly derived from manual analysis of the AOSP source code of the ART project. *Optimizing*'s intermediate representation is essentially a control flow graph on the method level, which the Android developers denote as `HGraph`. The graph is further enriched with structural data about the program and populated with instruction nodes, denoted as `HInstructions`. Figure 3.2a presents an example Java code snipped and Figure 3.2b presents the resulting[3] `HGraph` of the `getID` function in the *Optimizing* IR.

HGraph. The `HGraph` serves as the single intermediate representation of the app methods' code. When the graph is created, `dex` instructions of every method's bytecode are scanned one after another, and the corresponding `HInstructions` are created as the graph's nodes and interlinked with the current basic block and the rest of the graph. In order to allow for complex optimizations, the graph is transformed into *single static*

[3] Presented code is simplified and limited to relevant instructions for the sake of readability.

assignment form (SSA). Pairs of value definitions and usage, so-called *def-use-pairs*, are created during a liveness analysis and explicitly interlinked afterwards. To deal with cases in which the exact value cannot be determined statically, special *phi* nodes are inserted. *phi* represents a pseudo-function that will "decide" which of the inputs will become the output, thereby providing a useful tool for static analyses to cope with situations where a variable assignment depends on a decision at runtime (e.g., branching behavior). Using this workaround, the *phi* node can be used as a surrogate input to all operations that depend on this runtime decision.

In this form, the graph is amenable to a multitude of possible optimizations. The available optimizations include algorithms such as *Bounds Check Elimination* to remove redundant bounds checks, *Global Value Numbering* to remove duplicate assignments, *dead code elimination* to delete unreachable code, or *loop invariant code motion* to optimize hotspot code in loops. In Section 3.5.1, we show how we can also leverage this form for security-oriented instrumentation, thus supporting compiler-based security solutions on Android, such as dynamic taint tracking (see Section 3.6.2).

HInstructions. The graph's `HInstruction` nodes roughly correspond to `dex` instructions. Besides this transformation, nodes in the `HGraph` have additional attributes that have no equivalent in `dex` bytecode (e.g., an SSA index). The `HInstructions` distinguish between arguments and inputs. While the former correspond to the arguments given to an operator or method, the latter encode additional dependencies that may not be immediately observable given only the underlying `dex` code. Static method invocations, for example, additionally have an `HLoadClass` or `HClinit` instruction node as their input, both of which encode auxillary information for the compiler. All `HInstructions` share a basic set of information: (Primitive) type, inputs, def-use relations, id, and further data are attached to each node in order to ease the creation of and working with the `HGraph`. Each node is uniquely identified within the graph by its id that is assigned and incremented continuously during node creation.

In SSA form, every instruction node also represents the value it *outputs*, i.e., its return value for a method invocation or the actual value for a constant. This way, instruction nodes can be provided as inputs for other instruction nodes. Consequently, `HInstruction` nodes need to be typed. The attached primitive type can be `Void` for methods that have no return value, `Not` for strings and object types of any kind, and additionally any Java primitive type (e.g., `float`, `int`, `long`). Since those only represent primitive types properly, a backreference to the original `dex` file is required to obtain the actual object type from a `Not`-typed instruction. This coupling between `HInstructions` and `dex` instructions as well as the presence of a method local `dex` program counter in each node show that the IR is not completely independent of the original `dex` file.

Semantic consistency. In addition to the instructions that represent the original application logic, the `HGraph` also contains meta-instructions to preserve the semantic consistency between the original Java code of the developer, the `dex` bytecode shipped with APKs, and the compiled code executed by ART. First, additional instructions are inlined into the graph to support meaningful debugging (e.g., to map from segmentation faults in ART to actual stack traces) and to conduct various forms of runtime checks (e.g., type casting, bounds checking, division-by-zero checks, or null pointer exceptions). Second, instructions to represent so-called *suspension points* are added, which effectively subdivide the application code into multiple chunks. Each suspension point between two chunks acts as a synchronization point between native code and original `dex` bytecode in the program execution and also serves as an entry point for garbage collectors or debuggers. Those chunks may be optimized, but as soon as a suspension point is reached, several prerequisites need to be fulfilled. First, the thread that is currently executing the code must be ready to stop its execution. This is important because certain types of garbage collectors require to hold mutator locks. Second, the suspension points provide points of memory consistency between the native code and the interpreted bytecode, so at those points the registers are spilled to the stack for example. Third, in case a debugger is attached, the next suspension point may mark a transition from executing the native code to falling back to bytecode interpretation in order to support breakpoints and other debugging features. Also, at those points dynamic deoptimization may happen in order to ease debugging for developers.

3.3 Problem Description

A dedicated line of work, including the TaintDroid project [49] and its derivatives, relied on instrumentation of the now abandoned Dalvik virtual machine. As a consequence, the research community faces the dilemma on how to continue this line of work and is left with two choices (see Figure 3.3): Either compensating the missing runtime instrumentation through app rewriting techniques — `dex` bytecode ($\mathsf{Instr}_{\mathrm{APK}}$) or binary ($\mathsf{Instr}_{\mathrm{OAT}}$) — or by taking advantage of Android's new compiler suite ($\mathsf{Instr}_{\mathrm{DEX}}$, $\mathsf{Instr}_{\mathrm{OPT}}$, and $\mathsf{Instr}_{\mathrm{BIN}}$). Although `dex` bytecode rewriting is well-established in contexts such as inline reference monitoring [95, 43, 42, 29] and taint analysis [120], [138], and ART now supports porting binary rewriting techniques from commodity systems, this work builds on compiler-based instrumentation to not only re-instantiate previous approaches that relied on Dalvik VM instrumentation, but also to explore novel security solutions that leverage the compiler features.

In the following, we analyze the concrete requirements that an instrumentation solution should provide and discuss, for each of the above approaches (i.e., bytecode rewriting $\mathsf{Instr}_{\mathrm{APK}}$, binary rewriting $\mathsf{Instr}_{\mathrm{OAT}}$, and compiler-based instrumentation), their respective benefits and shortcomings in fulfilling those requirements. Table 3.1 provides a summary of our requirements analysis.

R1. Enforceable security policies. Each of the three approaches operates on one of the different representations of the same app code, i.e., bytecode, IR, or binary. Hence, all three approaches are identical in their capabilities of instrumenting the code and none of the solutions addresses any security policy alone.

R2. Strong security boundary. Both rewriting and the compiler-based approach rely on injecting monitoring code into the app's process space and can therefore not provide a strong security boundary between monitoring and (potentially) malicious app

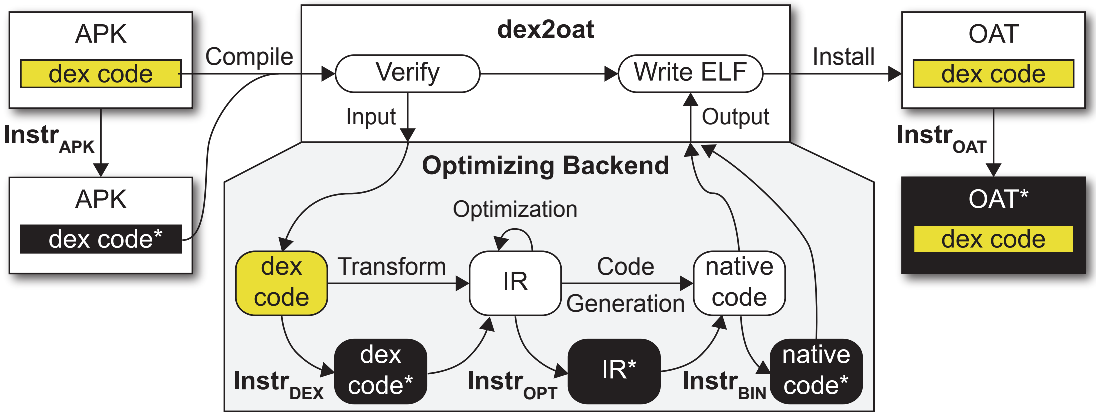


Figure 3.3: The code instrumentation points before, during, and after the compilation for different representations of the app code. Instrumented code is depicted in black boxes.

	Bytecode rewriting	Compiler-based	Binary rewriting
R1. Enforceable security policies		identical	
R2. Strong security boundary	✗	✗	✗
R3. Application layer only	✓	✓	✓
R4. User privilege only	✓	(✗)	(✗)
R5. Platform independence	✓	✓	✗
R6. Signature preservation	✗	✓	✓
R7. Robustness against optimization	✗	✓	✓
R8. Integrated approach	✗	✓	✗
R9. Supported versions	all	6+	5+

✓= fulfilled; ✗= not fulfilled

Table 3.1: Comparison of security and deployment features between bytecode rewriting, compiler-based instrumentation, and binary rewriting.

code (✗), e.g., native code. Thus, all of them can only provide security guarantees for at most honest-but-curious apps.

R3. Application layer only. All approaches can be implemented purely on the application layer (✓). Deploying bytecode rewriting techniques Instr$_{\text{APK}}$ in the form of separate apps has been presented in the literature [95, 43, 42, 29]. On Android, a compiler-based solution can be deployed as a separate app that ships and controls the security-instrumented compiler suite (see also Section 3.5.4.1). For both the compiler-based approach and the binary rewriting, the main requirement is access to the storage location of applications' `oat` files, which does not require system modification.

R4. User privilege only. While `dex` code is freely available for non-forward locked apps[4], accessing applications' `oat` files makes elevated privileges necessary. However, in Section 3.7.1.3, we discuss approaches that would allow both binary and compiler-based rewriting to circumvent this problem without requiring elevated privileges.

R5. Platform independence. Bytecode rewriting $\mathsf{Instr}_{\mathrm{APK}}$ (✓) and compiler-based instrumentation (✓) can be applied on all platforms supported by Android, since they modify the code before platform-dependent native code is generated. Binary rewriting $\mathsf{Instr}_{\mathrm{OAT}}$, in contrast, depends on the actual hardware architecture of the platform (✗), thus requiring extra effort to support different hardware platforms.

R6. App signature preservation. App signatures are the foundation of Android's same origin model that governs the app update policy and sharing of resources between apps, like a common process or UID. Consequently, modifying bytecode $\mathsf{Instr}_{\mathrm{APK}}$ and the resulting obligation to resign and repackage apps breaks this same origin model (✗). In contrast, compiler-based instrumentation (✓) and binary rewriting $\mathsf{Instr}_{\mathrm{OAT}}$ (✓) do not modify the original app package and therefore do not invalidate the signature because the code modification happens outside of the signed APK file.

R7. Robustness against code optimization. The instrumentation point determines whether any instrumented code will be subject to optimization at compile time. Applying optimization algorithms to instrumented code has the potential to interfere with the semantics of the modification through, e.g., instruction reordering, inlining, or similar techniques of state-of-the-art compilers. On the one hand, current bytecode rewriting approaches $\mathsf{Instr}_{\mathrm{APK}}$ are applied before compilation; thus any instrumentation has to be robust against optimizations—an aspect not yet further investigated by contemporary research (✗). On the other hand, binary rewriting $\mathsf{Instr}_{\mathrm{OAT}}$ is restricted to instrumenting optimized code (✓), but misses the chance to reuse the rich optimization frameworks of modern compilers to also optimize added security code. The sweet spot occupied by compiler-based instrumentation provides full control over which optimizations are applied when and in which ordering (✓). Furthermore, this enables creating optimizations that are specifically tailored towards improving the instrumented code by utilizing the static program information that is present in the compiler.

R8. Integration into toolchain. Integrating an instrumentation system into an existing toolchain ensures continuous development and maintenance by the community as well as access to established and well-tested tools and frameworks. In this case, even though the ART project is open source and therefore open to the community, the compiler is mostly maintained by Google itself. Consequently, compiler-driven solutions that do not break with the toolchain's regular functionality benefit from the continuous improvements (✓). In the case of ARTist, the amount of code that needed to be changed is minimal and therefore easy to adapt for newer versions of the toolchain. Bytecode

[4] Section *Forward Locking* in *Android Security Internals* [48] explains Android's copy protection in more detail.

rewriting $\mathsf{Instr}_{\mathrm{APK}}$ and binary rewriting $\mathsf{Instr}_{\mathrm{OAT}}$ are developed separately from the toolchain and do not reap those benefits (✗).

R9. Version support. While bytecode instrumentation $\mathsf{Instr}_{\mathrm{APK}}$ can be applied to *all* Android versions, compiler-based approaches and binary rewriting $\mathsf{Instr}_{\mathrm{OAT}}$ depend on ART and therefore can only be applied since Lollipop (*5+*), where a compiler-based solution (as presented here) should utilize the *Optimizing* backend on Android *6+* in preference to *Quick*.

Sweet spot. In conclusion, comparing the security and deployment features that the three available instrumentation approaches provide, following a compiler-based approach for designing ARTist has very appealing properties and occupies a sweet spot among all approaches.

3.4 Contribution

In this chapter, we present a compiler-based framework that can be used to study the feasibility of re-instantiating previous instrumentation-guided approaches such as dynamic, intra-application taint tracking and dynamic permission enforcement, and that provides a more robust, reliable, and integrated application layer instrumentation approach than previously possible. More precisely, we make the following contributions.

Uncovering the Uncharted ART Compiler Suite. Since the novel ART compiler suite, *dex2oat*, has still not received much attention in the academic community, we uncover its applicability for compiler-based security solutions to form expert knowledge that facilitates independent research on the topic. In particular, we provide a deep-dive into its most recent backend called *Optimizing* that became the default in Android 6 Marshmallow, and expose its relevance for ARTist as well as future work in this area. We envision this to be a first step towards establishing compiler-based security research on Android, which comprises already a popular line of work outside the mobile domain (see *LLVM* [26]).

Compiler-based Instrumentation. We design and implement a novel approach called ARTist for instrumenting Android components, based on an extended version of ART's on-device compiler *dex2oat*. Our system leverages the compiler's rich optimization framework to safely optimize the newly instrumented code. Our system supports the instrumentation of apps as well as Java-based system components, such as the Systemserver. The instrumentation process is guided by static analysis that utilizes the compiler's intermediate representation of the app's code as well as its static program information in order to efficiently determine instrumentation targets. A particular benefit of our solution, in contrast to alternative application layer solutions (i.e., bytecode or binary rewriting), is that the application signature is unchanged and therefore Android's signature-based same origin model and its central update utility remain intact. We thoroughly discuss further benefits and drawbacks of security-extended compilers on Android in comparison to bytecode and binary rewriting. Our results provide compelling arguments for preferring compiler-based instrumentation over alternative bytecode or binary rewriting approaches. We open sourced and documented ARTist and its companion tools at `https://artist.cispa.saarland`. See Appendix A.1 for a list of code repositories.

Feasibility Study for Instrumentation Modules. To demonstrate the benefits of a solution such as our ARTist, we conduct multiple case studies of instrumentation modules that have been suggested in the literature. We show that compiler-assisted instrumentation can be utilized to realize complex dynamic analysis systems, such as intra-application taint tracking at runtime or inline reference monitoring for dynamic permission enforcement. We thoroughly evaluated our taint tracking case study by using microbenchmarks and confirmed its operational capability using an open source test suite with known ground truth.

Large-scale Evaluation of ARTist. In order to properly benchmark the robustness of ARTist, we created *monkey-troop* to automate the instrumentation and evaluation of apps using ARTist. Our tool automatically downloads and installs apps, instruments them using ARTist, and exercises their features using Google's *monkey* [14] software. Using *monkey-troop*, we conduct a large-scale evaluation with top ranked apps from the Google Play store to show that ARTist can successfully instrument most third-party applications.

3.5 Design & Architecture

Figure 3.4 provides an overview of the ARTist ecosystem. The main component is a security-instrumented compiler (*sec-compiler*) that is deployed either as a part of a custom ROM or on rooted stock ROMs using a regular Android app (*ArtistGui*) that deploys and manages the compiler and modules. Instrumentation logic is encapsulated by so-called modules that we can create with architecture-specific module SDKs.

3.5.1 ARTist compiler

ARTist's *sec-compiler* enhances Android's *dex2oat* with additional instrumentation routines. This section will highlight design challenges and decisions we encountered while building *sec-compiler*, such as the placement of our instrumentation code within the compiler, the modification capabilities of our approach and the inclusion of custom libraries into target applications.

Choice of instrumentation point. Given *dex2oat*'s modular design, multiple possibilities for the placement of app-modifying code are immediately apparent. For instance, *dex2oat*'s design would easily allow porting bytecode and binary rewriting approaches ($\mathsf{Instr}_{\text{DEX}}$ & $\mathsf{Instr}_{\text{BIN}}$) into the compiler infrastructure (see Figure 3.3). Both techniques, although thoroughly studied in the literature, would benefit from instantiation within the compiler infrastructure by improving upon known shortcomings such

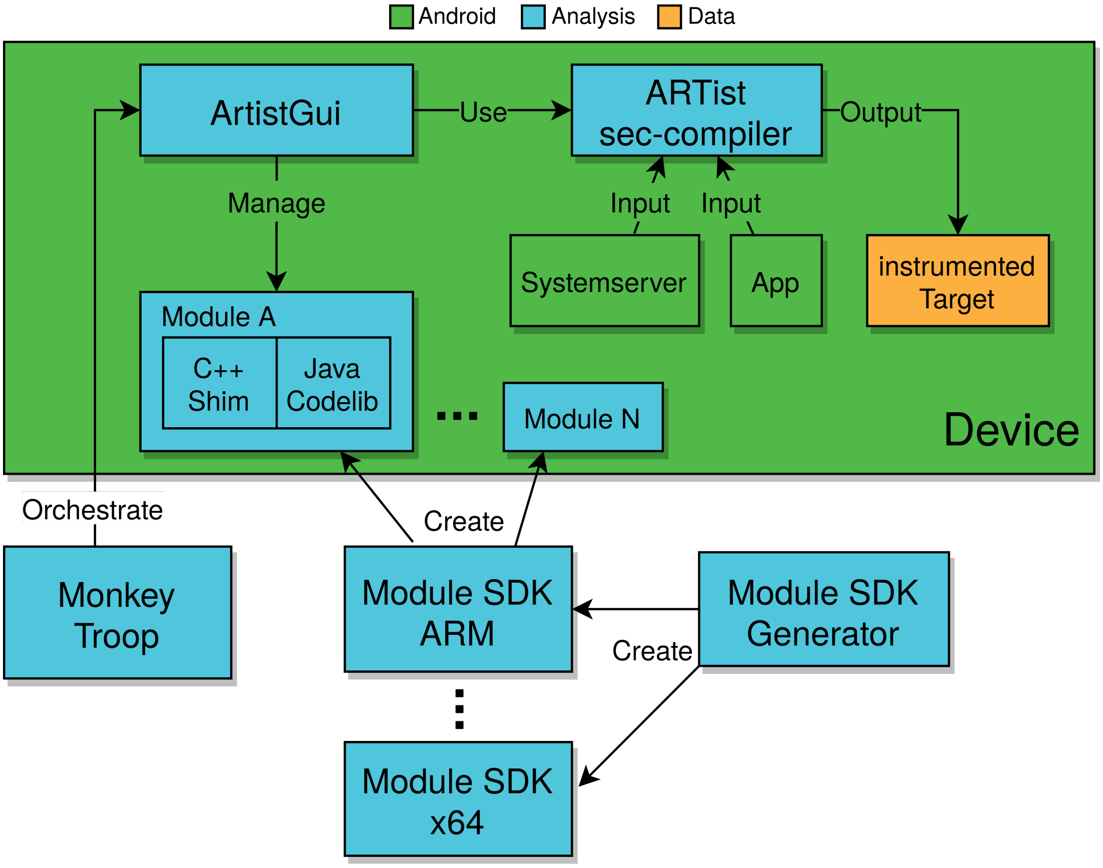


Figure 3.4: Overview of the ARTist ecosystem.

as the requirement to re-sign modified applications. However, of the different choices, ARTist's *sec-compiler* is explicitly designed to operate on the intermediate representation of *dex2oat*'s *Optimizing* backend ($\text{Instr}_{\text{OPT}}$), where the existing optimization infrastructure and static code information in the *Optimizing* IR allow for efficient and precise code modification. More precisely, our app instrumentation code is realized using the `HOptimization` class that represents optimization passes over the compiler's intermediate representation of the target app. As a consequence, instrumentation passes created for ARTist properly integrate into the compiler's optimization framework, including automated execution and access to the currently compiled method's `HGraph`. Implanting our instrumentation routines into the optimization workflow additionally grants us full control over the ordering and execution of optimizations in general, which opens up the opportunity for arbitrary reordering or even introduction of own optimization passes. Multiple such passes can combine different instrumentation routines or specifically crafted optimizations can improve the performance of our security code within target apps.

Generally speaking, the `HOptimization` interface's loose coupling allows to integrate new functionality into the compiler while — at the same time — keeping the effort for adaption of patches and maintainability to a minimum.

Spotting instrumentation targets. `HGraph` supports the visitor pattern [57] that enables us to iterate over, inspect, and modify each single `HInstruction` of the app's code. In contrast to method hooking techniques, we can therefore operate at the instruction level. We use `HGraphVisitors` primarily to identify instrumentation targets and apply the desired modification. However, they can also be utilized to bootstrap static analysis. We will see concrete implementations using a visitor to collect instrumentation sites for our dynamic permission enforcement system in Section 3.6.1 and starting points for backward slicing in our taint tracking case study in Section 3.6.2.

Modification capabilities. With full access to a method's `HGraph`, ARTist can arbitrarily modify the target application's code, e.g., to change inputs, types, or even remove, add, or replace instructions. ARTist even provides a dedicated API to inject arbitrary method calls into `HGraphs`, which, combined with the capability to inject whole libraries (see Section 3.5.4.1), allows to inject arbitrary code into target applications. Developers simply declare the instrumentation location, the method to be invoked and the inputs to be passed. An example can be found in Section 3.6.1 where our dynamic permission enforcement use case module make extensive use of this feature.

All instrumentation routines operate on `HInstruction` nodes, meaning the `dex` frontend and native code generators stay agnostic towards our changes and can therefore be used as is. The result of this integrated solution is that we still take advantage of the robustness of *Optimizing*'s code generators, which are well-tested, constantly improved, and in productive use on every stock Android phone running version 6 and above.

3.5.2 Modules

In the ARTist ecosystem, modules represent instrumentation tasks that are executed by the *sec-compiler*. They typically combine a code library, which is injected into the target and represents the runtime logic of the module, with instrumentation logic, which is utilized by *sec-compiler* to modify the target at compile-time. Modules are built using our module SDK and can be loaded and executed independently of each other.

Code Libraries. In order to ease the process of developing instrumentation logic to a minimum, ARTist modules can embed so-called *CodeLib*s, full code libraries that are injected into the target prior to the instrumentation through *sec-compiler*. While theoretically, all modifications in the target could be done from within *sec-compiler*, it is more efficient to implement the module's own logic in a Java library[5] and use the actual instrumentation code only to connect the target's existing functionality with the injected code. One example is the inline reference monitoring approach that we discuss in Section 3.6.1, where the whole monitoring and state management is implemented in such a library and the instrumentation passes are only used to modify the target to call into this library at well-defined places in the target code. Following this design, we keep the compiler-dependent code to a minimum, thereby not only increasing the efficiency of the module developer but also the robustness against changes in the *dex2oat* compiler suite.

Instrumentation Passes. A module's instrumentation pass acts as the thin shim between the target's original code and the injected code of the module developer. Its sole purpose is to either directly modify the target on the instruction level or add method invocations to the target that call into the injected code library. This glue code is typically written in C++ and the resulting shared library is loaded into and used by the *sec-compiler*. Since module developers can implement the core runtime logic in a *CodeLib*, the actual instrumentation code is very focused and kept to a minimum. ARTist already provides a range of boilerplate functionality, such as a declarative injection API that allows to specify where to inject which method call. Furthermore, modules can declare filters that decide on a per-target method basis whether the module is executed, which allows for pinpointed instrumentation.

Manifest. The manifest carries a module's meta information and contains fields such as a unique name, current version, author information, and verbose description. While this information is currently used for module management and serving a user interface, we envision a module store where developers can upload modules and users can directly download them into ARTist.

3.5.3 Module SDK

The integration of instrumentation passes into *dex2oat*'s optimization framework comes at the price of strong dependencies on ART code. When developing new instrumentation

[5] In fact, anything that eventually compiles to `dex` files is supported (e.g., Kotlin).

passes, certain headers and libraries from AOSP need to be present at build time. This, however, would require potential module developers to work from within a fully built AOSP source tree, which introduces a major barrier of entry. We overcome this obstacle by creating the so-called module SDK, an installable (zip, debian, or rpm) package that bundles all necessary dependencies for building modules locally. After installation, developers can build and distribute their modules, in particular their instrumentation passes, with the help of our Makefile-based helper scripts.

SDK Generator. Bundling Android components into our SDKs introduces dependencies on the concretely used AOSP source tree. First, the header files required to interact with the compiler change over the course of multiple Android version upgrades, hence we need at least one SDK per Android major release (e.g., Marshmallow, Lollipop). Second, since native compiled libraries are also part of the SDK, we further introduce a dependency on the concrete hardware platform (e.g., `arm64`, `x86`), so we also need per-ABI SDKs. This inspired the creation of the *SDK Generator*, which fully automates the creation of an SDK from a built AOSP tree by pulling the correct files (i.e., headers and libraries) and combining them with the correct Makefiles and scripts for deployment. The result is a tool that creates installable SDKs for specific combinations of Android versions and hardware platforms.

3.5.4 Deployment Strategies

There are two major ways to apply *sec-compiler* to app or system targets on an Android device. Either we run *sec-compiler* selectively from a management app or we replace the compiler and its companion libraries as a part of an OS modification (custom ROM) so that it is automatically executed by Android itself.

3.5.4.1 Application Layer Deployment

In order to avoid modification of system components, we implemented a pure application layer deployment solution for ARTist. Based on the fact that *dex2oat* and its libraries are regular dynamically-linked ELF binaries, we can ship our custom compiler and its libraries as assets in our app called *ArtistGui*, the ARTist graphical user interface. Furthermore, using a regular Android application allows for simple deployment and updates.

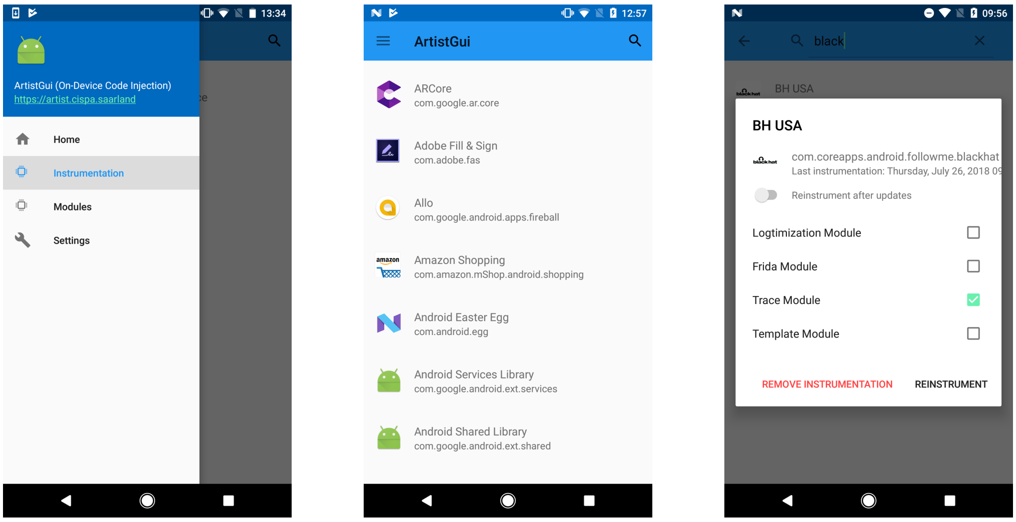
Instrumentation. When executing *dex2oat*, we utilize `LD_LIBRARY_PATH` to enforce the usage of our custom libraries that include the ARTist logic. As an input, we provide a prepared copy of the target file — app APK or middleware JAR[6] — as well as all modules that the user wants to apply. We denote the output as `oat'` (*oat prime*) to differentiate between the instrumented and the original compiled file. In order to complete the instrumentation and make the modified target runnable, it suffices to swap the original `oat` with `oat`'. The changes will take effect after the user either restarts the

[6] The Systemserver's code is stored in a JAR file instead of an application APK since it is not shipped through an app store and therefore requires less meta information.

app, in case the target was an application, or reboots Android if the target was a system component. The user stays agnostic to this change since she is still able to interact with the instrumented target as usual. However, `oat` files reside at protected locations that are not accessible by third-party apps such as our *ArtistGui*. A naïve solution to this problem would be to require extended privileges for our *ArtistGui* (e.g., a dedicated SELinux type or access to the root user) to replace the `oat` file. We discuss alternatives to the naïve approach in Section 3.7.1.3, which abstain from extended privileges by using app virtualization or reference hijacking.

CodeLib Injection. For targeting installed third-party apps, the first problem we face when injecting arbitrary code libraries before the re-compilation by *sec-compiler* is that this breaks the application's signature that essentially aims to verify that the app's `dex` code is unchanged. However, our approach focuses on replacing the compiled `oat` file, *not* the APK. Our toolchain therefore automatically unpacks the target app's APK and creates a copy that additionally includes the custom module *CodeLib*'s. The resulting extended APK is then provided as an input to *sec-compiler* that creates `oat'`. The original APK is not modified and therefore its signature remains valid. We additionally patch checksums and paths in the `oat'` header so that they refer to the original APK instead of the copy.
The second problem we observe is that adding the *CodeLib* to the target, i.e., injecting another `dex` file into the app package, does not make the new symbols (e.g., classes, methods and strings) available to other parts of the target's code. Method calls across `dex` files require extra information on, e.g., how to locate the invoked method. When the `dex` code is initially built, this is automatically done for all interactions between `dex` files, but since our *CodeLib* is added afterwards, we have to manually fix this by merging symbols of our new functionality into the existing `dex` files. However, we do not know which new functionality is invoked from which part of the target code since the merging of symbols needs to happen *before sec-compiler* is executed, hence we need to make all new symbols available to all existing `dex` files. In our toolchain, we created a program called *Dexterous* that extends the original `dex` symbol merging logic from the Android SDK, so that the whole merging process is automatically executed before the instrumentation and from within *ArtistGui*. Using this approach, *sec-compiler* will always be able to inject calls from any part of the target code to any *CodeLib* functionality.

Module Management. Another responsibility of *ArtistGui* is the management of instrumentation modules. First, it loads, parses and manages module zip files that are created with our module SDK. Second, it allows the user to select the instrumentation targets (i.e., apps and system components) and the set of modules that will be used for the instrumentation. Figure 3.5 depicts the user navigation flow of instrumenting an application with a subset of installed modules. Since app updates (e.g., via Google Play) will create new `oat` files and therefore overwrite our instrumented versions, *ArtistGui* can also track these updates and automatically re-instrument the corresponding targets with the original set of modules to maintain the instrumentation status even while targets evolve over time.



BH USA

com.coreapps.android.followme.blackhat
Last instrumentation: Thursday, July 26, 2018 09

Reinstrument after updates

Logtimization Module ☐
Frida Module ☐
Trace Module ☑
Template Module ☐

Figure 3.5: Navigation flow of a user instrumenting a target with a certain set of modules.

3.5.4.2 Custom ROM

In addition to an application layer-only approach, ARTist supports a system-centric deployment model that requires source code access to the AOSP tree and is therefore only applicable to platform developers that are shipping their own custom Android ROMs. Instead of tricking the loader at runtime (e.g., as done with `LD_LIBRARY_PATH` in *ArtistGui*), we directly replace the `ART` repository with our extended version that incorporates ARTist's *sec-compiler*, which ensures that the *dex2oat* compiler for this compiled Android version includes the ARTist functionality. Also, this requires to run the *Dexterous* tool for symbol merging before the actual instrumentation, which was taken care of by *ArtistGui* in the application layer scenario. The result is a custom ROM that can automatically instrument every single target on the device, which is particularly useful for dynamic analysis setups where, e.g., we want to instrument all targets to provide coverage information, allow debug access, or increase the logging verbosity. In Chapter 4, we will see how this can be applied to instrument the Systemserver to allow for large-scale dynamic analysis.

3.6 Case Studies

We demonstrate the applicability and usefulness of our system by exemplarily discussing two use cases implemented as ARTist modules: First, we implemented an Inline Reference Monitor (IRM) injection module to allow for dynamic permission enforcement. Second, we conduct a case study on realizing intra-app taint tracking through inlining of taint tracking code. In addition, we discuss further applications of ARTist and their implementations.

3.6.1 Inline Reference Monitoring for Dynamic Permission Enforcement

In the literature, Inline Reference Monitoring (IRM) on Android is mostly implemented by modifying the bytecode before the installation [95, 43] or by hooking into an application's method at the caller or callee side at runtime [29]. By utilizing a security-instrumented compiler, IRM can be implemented without the need to resign and repackage apps as it is required by established approaches. Moreover, when implemented with ARTist, IRMs can operate at instruction granularity instead of employing method-level hooks. Those capabilities are showcased by our IRM injection module that allows for dynamic permission enforcement, as shown by [29, 95, 137] on Android versions before Marshmallow.

The module is split into two parts: the code injection routine that will inline permission enforcement code and the *CodeLib* that acts as a policy decision point. While the former directs the instrumentation process at installation time, the latter enforces the user's policy at runtime.

Code injection. We first utilize *Optimizing*'s method graphs to locate the call sites of permission-protected SDK methods that are defined in a policy configuration file. Afterwards, ARTist injects additional calls to our companion library right before the call sites to check whether the critical method invocations should be allowed. This ensures that the control flow is diverted to our policy decision point before the execution of permission-protected methods.

Policy decision point. The library that our module injects into targets provides an API to check the app's current state of permissions. Based on a policy configuration file, the library either allows or rejects the execution of a protected SDK method. In the former case, the app's execution is continued without further interruptions. In the latter case, the monitoring code, e.g., logs an alarm, interrupts the app by raising an exception, or even terminates the app completely to avoid unwanted usage of enforced permissions. The most straightforward case for such a user permission policy is a simple list of whitelisted or blacklisted permissions. However, more complex policies can be enforced by, e.g., restricting the number of allowed permission requests in total, within a timeframe (e.g., at most five SMS per hour), or even to a certain time of the day (e.g., block location access during the night). Furthermore, another application or system component could expose a graphical interface to the user that allows to fine-tune this behavior, similar to the permission settings in Android versions newer than Marshmallow, or as observed in related work [29].

3.6.2 Taint Tracking

Established approaches for dynamic taint tracking on Android [49] rely on instrumenting the now superseded DVM for intra-application taint tracking or directly rewrite bytecode [120], [138]. In this case study, we explore the applicability of ARTist to re-instantiate intra-app taint tracking for applications on Android version 6 and higher. Through a prototypical implementation, we want to investigate whether inlining taint tracking logic into the application code base with ARTist can be a surrogate for solutions prior to Android version 5. Note that this case study does not aim to be a full replacement of existing solutions like TaintDroid, but demonstrates a new potential foundation for future taint-tracking on Android.

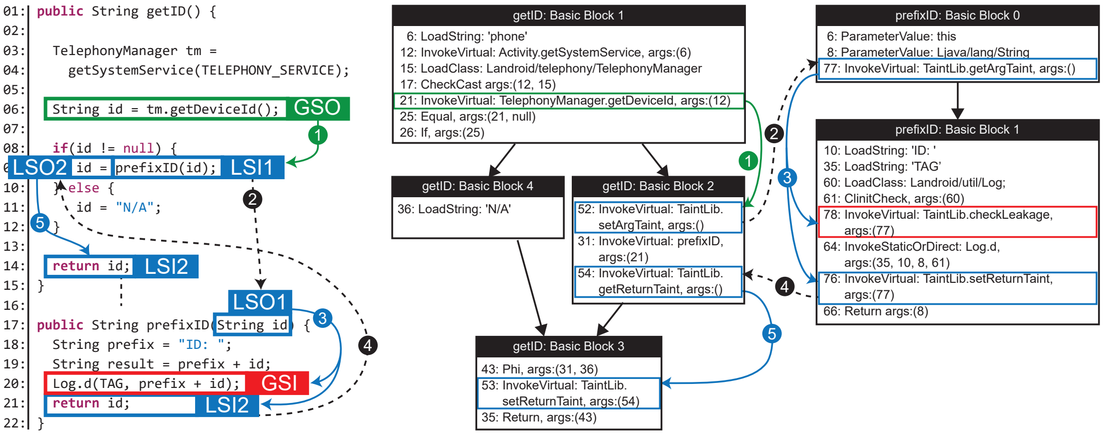
3.6.2.1 Module Design

Using tracking logic inlined by a new `HOptimization` pass via ARTist, we want to track information as it flows through the code. However, simply assigning a taint tag to each single value that should be tracked and updating this tag for each single instruction operating on it would incur a major performance penalty. To minimize the runtime impact, we split our approach into two phases: *analysis* and *instrumentation*. During the *analysis* phase, we identify flows of tainted information between *sources* and *sinks*. By restricting ourselves only to those relevant flows of the values we are interested in, we avoid generating irrelevant but costly taint tracking code for parts of the method that never actually influence the data that is observed and gain noticeable performance improvements over more naïve taint tracking. During the *instrumentation* phase, code will be inlined that creates, propagates, and checks the taint values along the identified data flows. Our combined analysis and instrumentation achieves flow-, path-, object-, and context-sensitive taint tracking. While [120] and [100] also utilize static analysis to optimize and guide the instrumentation process, both assume a holistic view on the application in the form of a control or data flow graph. In contrast, the *Optimizing* backend and therefore ARTist operate on a per-method level, which leaves inter-method taint tracking as a major challenge. A naïve solution to this problem would be to retrofit the compiler suite to provide an application-wide view and instrumentation. However, our prototype demonstrates how we can still achieve taint tracking for the whole application while restricting ourselves to a per-method view and instrumentation. To this end, we introduce in the following a new design for storing and propagating taint tags, in particular we have to refine the definitions of *sink* and *source*.

3.6.2.2 Analysis Phase

In order to optimize the instrumentation with taint tracking code, we exploit the processing features (e.g., `HGraph`'s visitor [57] pattern support) of the *dex2oat* compiler to detect the data flow sources and sinks, and afterwards use its built-in analysis features to identify the relevant data flows and the operations along those flows that have to be instrumented.

Refining Source and Sink Definition. The literature on taint tracking for Android defines sources and sinks as the API methods that introduce privacy-sensitive information into the application process (e.g., framework functions that return sensitive data, such as the location or telephony API) or, respectively, leak privacy-sensitive information from the application process (e.g., file handles, Internet sockets, or logging facilities). Since ARTist is operating on a per-method level, we cannot assume that our analysis is able to always connect a sink and a source (e.g., when they are located in different methods). To address this problem, we have to connect the data flows of tainted variables across the different methods while maintaining the per-method analysis.



```
01: public String getID() {
02:
03:   TelephonyManager tm =
04:     getSystemService(TELEPHONY_SERVICE);
05:
06:   String id = tm.getDeviceId();
07:
08:   if(id != null) {
0      id = prefixID(id);
10:   } else {
11:     id = "N/A";
12:   }
13:
14:   return id;
15: }
16:
17: public String prefixID(String id) {
18:   String prefix = "ID: ";
19:   String result = prefix + id;
20:   Log.d(TAG, prefix + id);
21:   return id;
22: }
```

```
getID: Basic Block 1
 6: LoadString: 'phone'
12: InvokeVirtual: Activity.getSystemService, args:(6)
15: LoadClass: Landroid/telephony/TelephonyManager
17: CheckCast args:(12, 15)
21: InvokeVirtual: TelephonyManager.getDeviceId, args:(12)
25: Equal, args:(21, null)
26: If, args:(25)

getID: Basic Block 4
36: LoadString: 'N/A'

getID: Basic Block 2
52: InvokeVirtual: TaintLib.
      setArgTaint, args:()
31: InvokeVirtual: prefixID,
      args:(21)
54: InvokeVirtual: TaintLib.
      getReturnTaint, args:()

getID: Basic Block 3
43: Phi, args:(31, 36)
53: InvokeVirtual: TaintLib.
      setReturnTaint, args:(54)
35: Return, args:(43)

prefixID: Basic Block 0
 6: ParameterValue: this
 8: ParameterValue: Ljava/lang/String
77: InvokeVirtual: TaintLib.getArgTaint, args:()

prefixID: Basic Block 1
10: LoadString: 'ID: '
35: LoadString: 'TAG'
60: LoadClass: Landroid/util/Log;
61: ClinitCheck, args:(60)
78: InvokeVirtual: TaintLib.checkLeakage,
      args:(77)
64: InvokeStaticOrDirect: Log.d,
      args:(35, 10, 8, 61)
76: InvokeVirtual: TaintLib.setReturnTaint,
      args:(77)
66: Return args:(8)
```

Figure 3.6: Tracking tainted variable *id* from the example in Figure 3.2. All discovered sinks and sources are marked. Solid lines indicate intra-procedural data flows of tainted variables, dashed lines inter-procedural data flows between local sink-source pairs. The right-hand side depicts the inlined taint tracking code to propagate taint tags.

To this end, we introduce the concept of *method-local sinks/sources* in addition to the *global* sinks and sources defined in the literature. More precisely, HInstructions that represent such local sinks and sources form the entry and exit points for inter-procedural data flows. Thus, global sinks and sources are points of interest for taint tag creation and checking, respectively, while local sinks and sources are for inter-procedural tag propagation. For local sinks and sources, we differentiate between three categories each: *Local sources* include parameters provided to the current method (LSO1), return values from method invocations (LSO2), and values read from fields (LSO3). Conversely, *local sinks* are method invocations that leak values through their arguments from the current method (LSI1), return statements of the current method (LSI2), and field setting instructions (LSI3).

Creating Intra-Procedural Data Flows.

For each global and local sink collected in the current method, we create a backward slice by tracing back the sink's inputs until a source or constant is reached. The found sources define which data can potentially leak through the sink, so our slice is fully defined by a sink and all its influencing sources appearing in the currently analyzed method. For instance, Figure 3.6 continues the example code from Figure 3.2. In the Java code on the left-hand side, all sinks have been identified (i.e., the parameter id passed to function prefixID in line 9 is a local sink of type LSI1, the return statements in lines 14 and 21 form local sinks of type LSI2, and in line 20 the call to Log forms a global sink). Using backwards slicing (solid lines ①, ③, and ⑤) the local sources in lines 17 (LSO1) and 9 (LSO2) as well as the global source in line 6 (getDeviceID call to retrieve device's phone number) have been identified. Each resulting backward slice is defined by its starting point (i.e., the sink) and all found endpoints (i.e., the sources). Constants cannot be tainted and are therefore explicitly omitted as sources. Together, those backward slices form the input for the instrumentation phase.

Because the backward slice dictates the targets for the instrumentation phase, high precision (i.e., avoiding over-approximation) is desirable for improved runtime performance, but not strictly necessary. Soundness, however, is crucial since missed data flows result in false negatives. Consequently, our slicing algorithm over-approximates to compensate for known shortcomings of static analysis like missing runtime information, e.g., when encountering *phi* nodes, reflection, or native method invocations. While over-approximating *phi* nodes by tracing back *all* its inputs is sound, handling, e.g., native code is more involved. Our heuristic assumes that all data provided as an argument to a native function will influence its result, hence the return value taint is the combined taint of all inputs. However, this is sound for side-effect-free (pure) functions only. Native code in general, as well as reflection, are limitations we share with similar approaches as ours.

3.6.2.3 Instrumentation Phase

During the instrumentation phase, we inline code that creates taint tags for global sources and that checks taints at global sinks at runtime. Additionally, we inline code that inter-procedurally propagates taints at runtime from a local sink to a local source, ensuring the data flow of a tainted value across multiple methods correctly propagates the taints.

TaintLib. Making use of ARTist's modular design allows us to deploy the taint tracking logic in form of a *CodeLib* called *TaintLib*. *TaintLib*, in turn, relies on a policy file that defines the global and local sources/sinks as well as the sources' taints tags. *TaintLib* provides source type-specific *taint-get* functionality that we inline at source locations, and sink type-specific *taint-set* functionality that we inline at sink locations. By injecting *TaintLib* method calls instead of concrete taint tracking logic, we decouple the instrumentation from the taint management code. For global sources, *taint-get* retrieves and sets the taint tag according to the policy and *taint-set* at global sinks checks[7] the taint tag. In contrast, for local sinks *taint-set* propagates the tag together with the tainted value to the next local source, where it is retrieved with *taint-get*. By instrumenting all methods alike, an implicit contract between all methods is established and fulfilled: Every time a *taint-get* tries to obtain the taint value of a method parameter on the callee side, we know the corresponding *taint-set* has been executed in the calling method to provide the taint data. In case the slice contains multiple `sources`, the output of their corresponding *taint-gets* is combined by injecting a call to a merger *TaintLib* method that combines the taint tags.

To continue our running example, the right hand side of Figure 3.6 presents the IR of the code snippet with *taint-set* and *taint-get* calls inlined. For instance, the `setArgTaint` call for `LSI1` in basic block 2 of `getID` (`HInstruction` 52) precedes the local sink in `HInstructions` 31 that invokes the `prefixID` function. The `setArgTaint` instruction transfers the taint of `id` inter-procedurally to the `getArgTaint` instruction in `HInstruction` 77 of basic block 0 of `prefixID` (dashed line ❷), from where it is

[7] While a naïve check halts the program when tainted data is about to leak, invoking a sanitizer as suggested by [100] is straightforward.

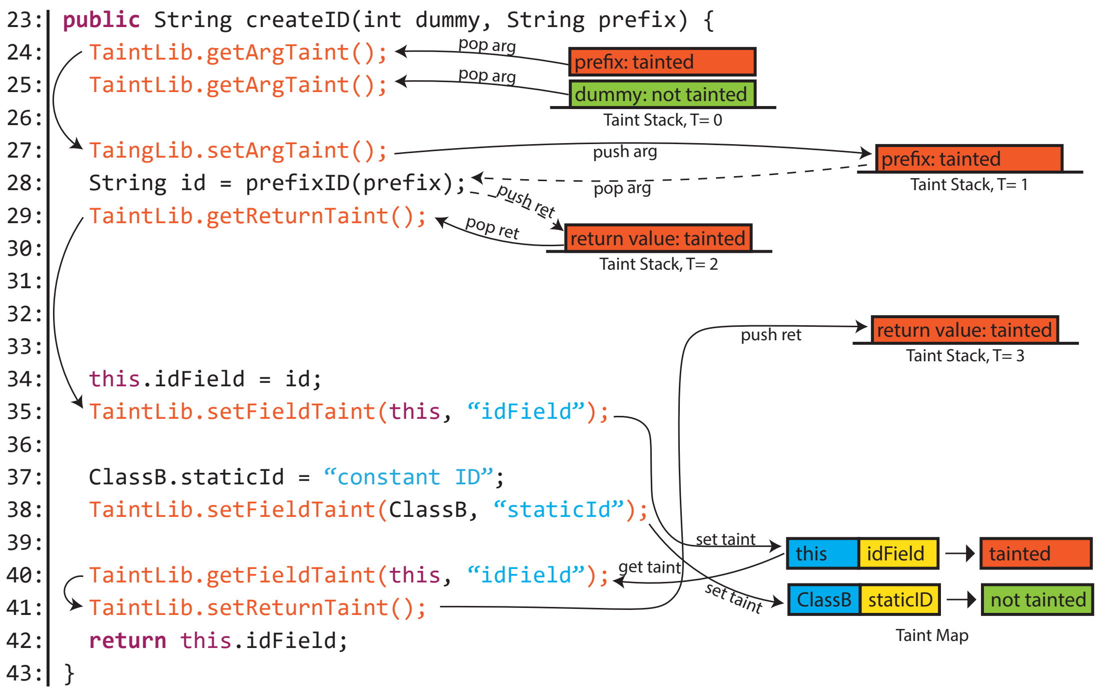
intra-procedurally propagated using the backwards slicing information (solid line ③). Similarly, the taint is propagated back from `prefixID` to `getID` through the return statement and variable assignment (dashed line ❹).

Inter-Procedural Taint Tag Propagation Channel. In the case of parameters (`LSO1` and `LSI1`) and method returns (`LSO2` and `LSI2`), there are always pairs of *taint-sets* and *taint-gets* present at runtime, due to the fact that for each callee method, there is a caller method that also has been instrumented[8]. Combining this with the observation that a caller-callee method pair is always executed in the same thread, the taint propagation can be realized using *thread local storage* for a *taint stack*. At the caller side, the taint information is pushed onto a per-thread stack and at the callee side it is popped again, vaguely resembling the x86 calling convention for passing arguments to methods. Keeping in mind that almost every injected *TaintLib* method call accesses the taint information, replacing more straightforward approaches for taint storage (like a single `HashMap`) with cheaper stack operations also benefits the overall performance of our taint tracking solution.

In the case of field operations (`LSO3` and `LSI3`), we can neither assume them to appear in pairs nor to always be executed on the same thread and therefore employ a thread-safe mapping in the form of a `ConcurrentHashMap`. This, however, raises the challenge of providing easily computable, stable and unique keys. If we consider our taint tags not to store the taint value of a certain value, but of a certain location, we can compute stable identifiers for fields and use them as keys. For `static` class fields, identifying the specific class and field is sufficient and can be pre-computed during compilation. The current implementation injects the computed key as a constant into the `HGraph` and provides it as an argument to a field *taint-set* or *taint-get*. For object fields, we do not only need to identify classes but concrete objects, which requires runtime information. In this case, we only inject the field identifier as a constant and provide it together with the field's concrete object to a *TaintLib* function. The returned key is robust to object aliasing such that we do not lose track of objects in, e.g., collections. Afterwards, we can use this key in a *taint-set* or *taint-get* for the object field. Figure 3.7 provides a step-by-step example of how taint information is updated on the thread-local taint stack and the global taint map.

It is important to note that our approach to taint tracking depends not only on the entity for which we store taints (i.e., variable locations instead of values), but also on the type of data to which we assign taint values. In our model, we track taints only for primitive types and the taint tag of objects is transitively given by their field's tags. In case of non-primitive fields, the rule applies recursively because eventually all objects can be decomposed to primitives. This design decision is motivated by the fact that tracking all *taint-set* and *taint-get* operations on fields and on all method invocations is more fine-grained than storing taint information at the object level.

[8] While this is true for the majority of cases, some exceptions like event callbacks are also considered and handled gracefully.



```
public String createID(int dummy, String prefix) {
  TaintLib.getArgTaint();
  TaintLib.getArgTaint();

  TaingLib.setArgTaint();
  String id = prefixID(prefix);
  TaintLib.getReturnTaint();




  this.idField = id;
  TaintLib.setFieldTaint(this, “idField”);

  ClassB.staticId = “constant ID”;
  TaintLib.setFieldTaint(ClassB, “staticId”);

  TaintLib.getFieldTaint(this, “idField”);
  TaintLib.setReturnTaint();
  return this.idField;
}
```

Figure 3.7: An illustration of the taint propagation, based on an extension of Figure 3.2. The taint stack is displayed at four points in time (T0-T3).

3.6.3 Outlook: Compartmentalization

> This outlook refers to the paper *The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android* [S3] that is **not** a part of this thesis. This paragraph only serves as a example for a full-fledged system built on top of ARTist.

An active line of work has identified libraries, in particular those focused on delivering advertisements, as a major source of vulnerabilities and privacy problems [50, 85, 32, 122, 127, 27]. To cope with this unsatisfactory situation, multiple countermeasures have been proposed to compartmentalize these libraries to, e.g., selectively revoke permissions or contain vulnerabilities to reduce the attack surface. While existing solutions either require modification of the operating system [122, 108, 124, 142] or violate Android's same origin model by breaking app signatures [128, 98], we suggest a new approach that uses ARTist to provide an application layer-only solution that can preserve target application signatures. Our system called CompARTist instantiates libraries in dedicated application processes and utilizes an ARTist module to detect interaction points between the app and advertisement libraries to replace them with IPC calls to the remote version of the library. CompARTist introduces a lightweight protocol on top of the binder IPC mechanism to keep the object state in sync and preserve visual fidelity for advertisements displayed within the app's user interface layout. Using this approach, we can effectively *carve out* an advertisement library, move it to a different process where we can enforce completely independent security policies (e.g., permissions), and reconnect both transparently. This work showcases how to use ARTist to create compiler-based security solutions that occupy a sweet spot among existing systems by tapping into the benefits of ARTist's app layer deployment model and its fine-grained instrumentation capabilities.

3.6.4 Further Modules

We will have a quick look at three example modules that inject debugging capabilities into target apps.

Stetho. Facebook's Stetho [52] is a debugging library that, when included in an application, connects this app at runtime to the Chrome Developer Tools [63] running on a connected desktop PC. Once connected, the Developer Tools allow to inspect and modify the application's files, databases, visual layout, and even traffic via a plugin system. Our stetho module [111] injects the required library into target applications and inserts function calls to provide a proper setup, including the interception of network traffic in case the `okhttp` library is used. Figure 3.8 shows how the connected Chrome Developer Tools can be used to inspect, e.g., traffic and databases. Both examples show the execution of the official reddit app [115].

Trace. The *trace* module is a simple helper that we inject into targets to test their compatibility with ARTist. All target methods in scope are modified to include a *CodeLib* method call in their preamble. In the *CodeLib*, we use stack inspection to find and log the name of the calling method. The module logs this information to *logcat*, making it easy for an analyst to, first, ensure that ARTist is running properly and, second, have a first impression of the target's code, in particular the external code and libraries used. Figure 3.9 shows an example *logcat* snippet generated by such an implementation. A variation of this module is also used for evaluating the robustness of ARTist in Section 3.7.1.1.

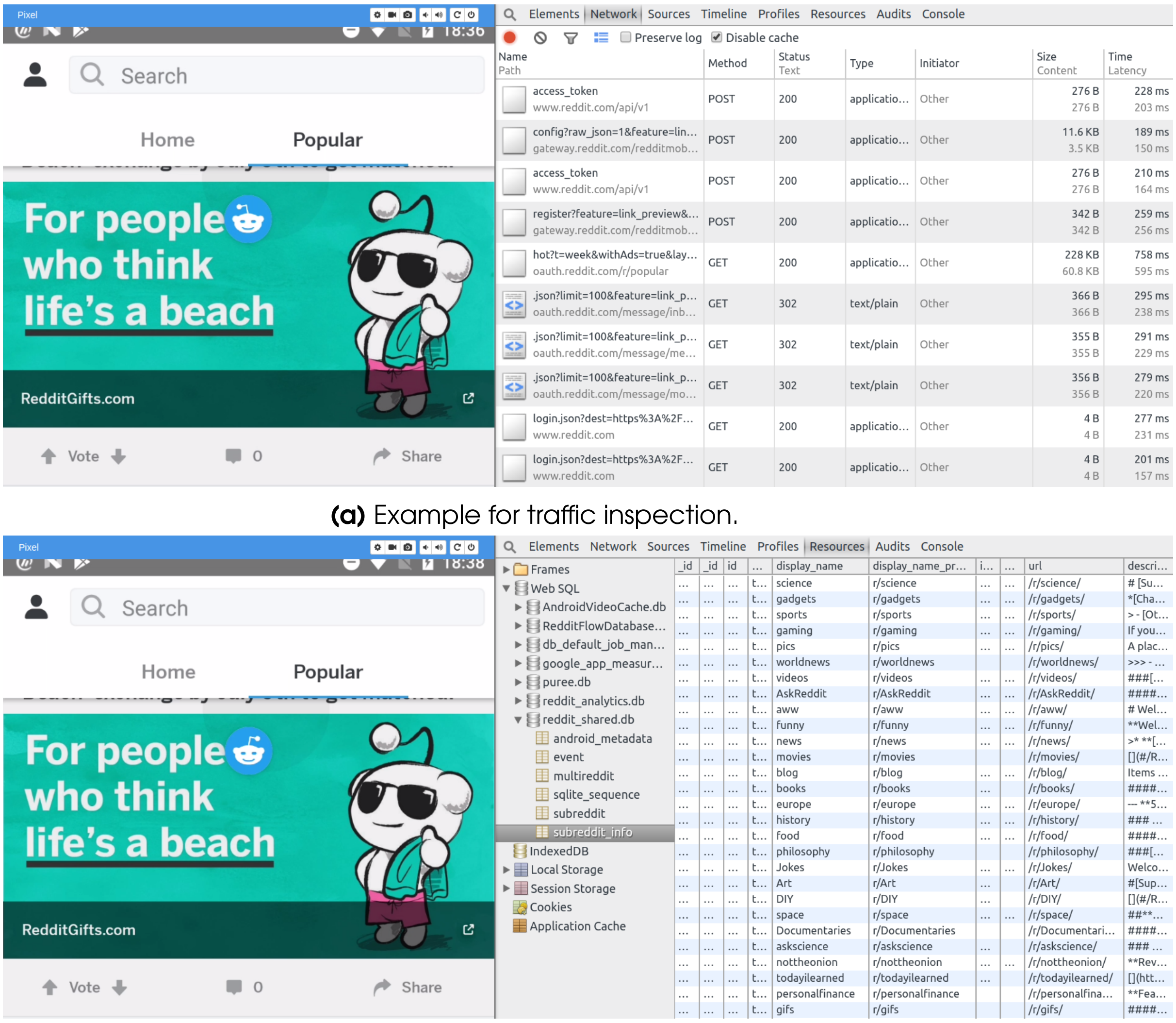
Gunshop (Community Contribution). In order to solve an Android-based CTF challenge called *Gunshop*, Alexander Fink created a specialized ARTist module to leak the AES key used for encryption and intercept `HTTPS` traffic by bypassing a custom certificate pinning implemented in the challenge [9].

3.7 Discussion

This section evaluates ARTist and its modules in terms of robustness, performance, inherent and implementation-specific limitations, and discusses ideas for future work.

3.7.1 ARTist

We first evaluate ARTist in terms of its robustness and discuss general limitations of the approach that are inherited by all modules.



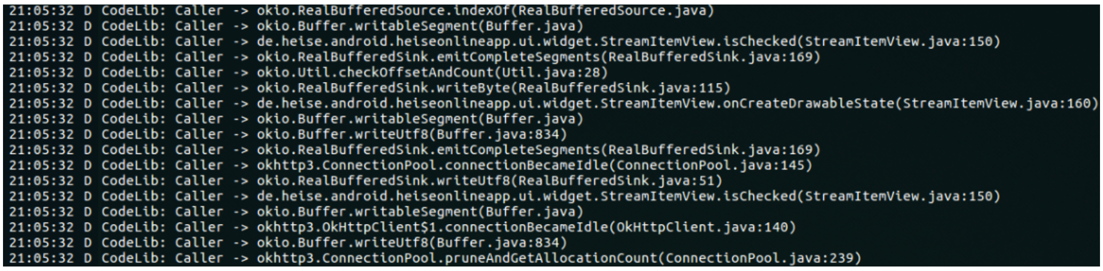
Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency
access_token www.reddit.com/api/v1	POST	200	applicatio…	Other	276 B 276 B	228 ms 203 ms
config?raw_json=1&feature=lin… gateway.reddit.com/redditmob…	POST	200	applicatio…	Other	11.6 KB 3.5 KB	189 ms 150 ms
access_token www.reddit.com/api/v1	POST	200	applicatio…	Other	276 B 276 B	210 ms 164 ms
register?feature=link_preview&… gateway.reddit.com/redditmob…	POST	200	applicatio…	Other	342 B 342 B	259 ms 256 ms
hot?t=week&withAds=true&lay… oauth.reddit.com/r/popular	GET	200	applicatio…	Other	228 KB 60.8 KB	758 ms 595 ms
.json?limit=100&feature=link_p… oauth.reddit.com/message/inb…	GET	302	text/plain	Other	366 B 366 B	295 ms 238 ms
.json?limit=100&feature=link_p… oauth.reddit.com/message/me…	GET	302	text/plain	Other	355 B 355 B	291 ms 229 ms
.json?limit=100&feature=link_p… oauth.reddit.com/message/mo…	GET	302	text/plain	Other	356 B 356 B	279 ms 220 ms
login.json?dest=https%3A%2F… www.reddit.com	GET	200	applicatio…	Other	4 B 4 B	277 ms 231 ms
login.json?dest=https%3A%2F… www.reddit.com	GET	200	applicatio…	Other	4 B 4 B	201 ms 157 ms

(a) Example for traffic inspection.

_id	_id	id	…	display_name	display_name_pr…	i…	…	url	descri…
…	…	…	t…	science	r/science	…	…	/r/science/	# [Su…
…	…	…	t…	gadgets	r/gadgets	…	…	/r/gadgets/	*[Cha…
…	…	…	t…	sports	r/sports	…	…	/r/sports/	> - [Ot…
…	…	…	t…	gaming	r/gaming	…	…	/r/gaming/	If you…
…	…	…	t…	pics	r/pics	…	…	/r/pics/	A plac…
…	…	…	t…	worldnews	r/worldnews			/r/worldnews/	>>> - …
…	…	…	t…	videos	r/videos	…	…	/r/videos/	###[…
…	…	…	t…	AskReddit	r/AskReddit	…	…	/r/AskReddit/	####…
…	…	…	t…	aww	r/aww	…	…	/r/aww/	# Wel…
…	…	…	t…	funny	r/funny	…	…	/r/funny/	**Wel…
…	…	…	t…	news	r/news	…	…	/r/news/	>* **[…
…	…	…	t…	movies	r/movies			/r/movies/	[](#/R…
…	…	…	t…	blog	r/blog	…	…	/r/blog/	Items …
…	…	…	t…	books	r/books	…		/r/books/	####…
…	…	…	t…	europe	r/europe	…	…	/r/europe/	--- **5…
…	…	…	t…	history	r/history	…	…	/r/history/	### …
…	…	…	t…	food	r/food	…	…	/r/food/	####…
…	…	…	t…	philosophy	r/philosophy			/r/philosophy/	###[…
…	…	…	t…	Jokes	r/Jokes	…	…	/r/Jokes/	Welco…
…	…	…	t…	Art	r/Art	…		/r/Art/	#[Sup…
…	…	…	t…	DIY	r/DIY	…		/r/DIY/	[](#/R…
…	…	…	t…	space	r/space	…	…	/r/space/	##**…
…	…	…	t…	Documentaries	r/Documentaries			/r/Documentari…	####…
…	…	…	t…	askscience	r/askscience	…		/r/askscience/	### …
…	…	…	t…	nottheonion	r/nottheonion	…	…	/r/nottheonion/	**Rev…
…	…	…	t…	todayilearned	r/todayilearned	…		/r/todayilearned/	[](htt…
…	…	…	t…	personalfinance	r/personalfinance			/r/personalfina…	**Fea…
…	…	…	t…	gifs	r/gifs			/r/gifs/	####…

(b) Example for database access and modification capabilities.

Figure 3.8: Screenshots of the Chrome Developer Tools connected to an instrumented target app via our ARTist stetho module.



```
21:05:32 D CodeLib: Caller -> okio.RealBufferedSource.indexOf(RealBufferedSource.java)
21:05:32 D CodeLib: Caller -> okio.Buffer.writableSegment(Buffer.java)
21:05:32 D CodeLib: Caller -> de.heise.android.heiseonlineapp.ui.widget.StreamItemView.isChecked(StreamItemView.java:150)
21:05:32 D CodeLib: Caller -> okio.RealBufferedSink.emitCompleteSegments(RealBufferedSink.java:169)
21:05:32 D CodeLib: Caller -> okio.Util.checkOffsetAndCount(Util.java:28)
21:05:32 D CodeLib: Caller -> okio.RealBufferedSink.writeByte(RealBufferedSink.java:115)
21:05:32 D CodeLib: Caller -> de.heise.android.heiseonlineapp.ui.widget.StreamItemView.onCreateDrawableState(StreamItemView.java:160)
21:05:32 D CodeLib: Caller -> okio.Buffer.writableSegment(Buffer.java)
21:05:32 D CodeLib: Caller -> okio.Buffer.writeUtf8(Buffer.java:834)
21:05:32 D CodeLib: Caller -> okio.RealBufferedSink.emitCompleteSegments(RealBufferedSink.java:169)
21:05:32 D CodeLib: Caller -> okhttp3.ConnectionPool.connectionBecameIdle(ConnectionPool.java:145)
21:05:32 D CodeLib: Caller -> okio.RealBufferedSink.writeUtf8(RealBufferedSink.java:51)
21:05:32 D CodeLib: Caller -> de.heise.android.heiseonlineapp.ui.widget.StreamItemView.isChecked(StreamItemView.java:150)
21:05:32 D CodeLib: Caller -> okio.Buffer.writableSegment(Buffer.java)
21:05:32 D CodeLib: Caller -> okhttp3.OkHttpClient$1.connectionBecameIdle(OkHttpClient.java:140)
21:05:32 D CodeLib: Caller -> okio.Buffer.writeUtf8(Buffer.java:834)
21:05:32 D CodeLib: Caller -> okhttp3.ConnectionPool.pruneAndGetAllocationCount(ConnectionPool.java:239)
```

Figure 3.9: Excerpt from the trace output when running an instrumented version of the *heise online* app (88).

Category	Tested	Success	Percentage
Books And Reference	44	39	88.64%
Business	37	33	89.19%
Comics	44	41	93.18%
Communication	45	38	84.44%
Education	38	35	92.11%
Entertainment	34	32	94.12%
Family	30	28	93.33%
Family?age=age Range1	41	40	97.56%
Family?age=age Range2	40	39	97.5%
Family?age=age Range3	29	27	93.1%
Family Action	33	31	93.94%
Family Braingames	43	43	100.0%
Family Create	47	44	93.62%
Family Education	46	45	97.83%
Family Musicvideo	52	49	94.23%
Family Pretend	40	38	95.0%
Finance	37	30	81.08%
Game Action	34	32	94.12%
Game Adventure	25	23	92.0%
Game Arcade	31	29	93.55%
Game Board	48	47	97.92%
Game Card	38	33	86.84%
Game Casino	25	22	88.0%
Game Casual	25	25	100.0%
Game Educational	36	34	94.44%
Game Music	41	37	90.24%
Game Puzzle	31	30	96.77%
Game Racing	32	30	93.75%
Game Role Playing	20	19	95.0%
Game Simulation	21	21	100.0%
Game Sports	40	40	100.0%
Game Strategy	25	23	92.0%
Game Trivia	42	38	90.48%
Game Word	50	46	92.0%
Health And Fitness	36	31	86.11%
Libraries And Demo	20	18	90.0%
Lifestyle	42	41	97.62%
Media And Video	36	33	91.67%
Medical	42	41	97.62%
Music And Audio	40	32	80.0%
News And Magazines	43	38	88.37%
Personalization	47	44	93.62%
Photography	44	38	86.36%
Productivity	34	31	91.18%
Shopping	39	34	87.18%
Social	33	25	75.76%
Sports	32	30	93.75%
Tools	48	44	91.67%
Transportation	49	46	93.88%
Travel And Local	41	36	87.8%
Weather	41	38	92.68%
51 Categories	1911	1761	92.15%

Table 3.2: Robustness evaluation results for the Google Play app store categories. Note that the *Family* categories contain apps from other categories as well. Filtered apps are omitted.

3.7.1.1 Robustness

In order to prove its applicability, we conducted an evaluation on top apps from the Google Play Store to show that ARTist-based instrumentation does not incur major impairments in the target app's robustness.

Evaluation Infrastructure. In order to scale our evaluation to thousands of apps, we created a dynamic app testing infrastructure called *monkey-troop*. It allows us to automatically test apps with ARTist modules on an arbitrary number of connected devices in parallel, thereby heavily reducing the time required for large-scale evaluations. The pipeline works as follows:

1. Setup. Before we start the actual testing, all devices need to be prepared for the evaluation. This, in particular, requires a working installation of *ArtistGui* as well as our special ARTist evaluation module that is based on *trace*.

2. Filter. In order to restrict ourselves to only test meaningful targets and avoid spoiling the evaluation results, we employ a pre-filtering on target application candidates. Some apps, even though they are in our list, could not be downloaded[9] and are therefore removed from the test set. The remaining ones that could also be installed are then tested using the *monkey* UI exerciser tool [14] *before* we apply our instrumentation.

[9] There is no official API for automatically downloading massive amounts of APKs from the Google Play Store and the current workarounds fail in some edge cases.

This way, we can rule out apps that are already crashing without any modification from our side.

3. Test. We apply the instrumentation by recompiling the target applications. Afterwards, we test them again using the *monkey* UI test automation tool and monitor the execution. Using the same *monkey* seed during filtering and testing, we can ensure that the app is tested with identical inputs during both phases. The module we are using is a customized version of *trace* that, instead of naïvely instrumenting *all* methods of the target, places the logging calls into certain well-defined event callbacks that have a high probability of being called by the Android OS when the application starts (e.g., `onCreate` methods of all `Activities`). This approach increases the chance of instrumented code being executed without requiring any knowledge of the target app's semantics. Consequently, if the application does not crash while being exercised by *monkey*, we count a success.

4. Collect. We collect and store all data generated by the testing scripts, as well as *logcat* dumps from the device to allow for further analyses after the evaluation.

Evaluation. We used *monkey-troop* to test the robustness of ARTist-based instrumentation on the most popular apps from the Google Play Store categories. As automated app testing with high coverage is still an open problem, the ARTist module created for the evaluation ensures the execution of our custom code by injecting it into each single `onCreate` method in any developer-written class. Thus, starting an application through its launcher activity (as done by *monkey*) always triggers our injected code.
The injected tracking code implements method-call tracing by utilizing stack inspection to print the current method's name to the log. Using this setup, we can evaluate the robustness of our instrumentation on real-world applications. The significant performance overhead incurred by the expensive stack inspection routine is only of secondary interest since we solely focus on robustness testing here. Table 3.2 shows the results of our evaluation. Out of 1911 tested apps, 1761 (92.15%) were successfully instrumented and tested, clearly showing the robustness of ARTist's instrumentation capabilities.
We conducted a manual investigation on the remaining applications in order to find the root cause of their failure during the evaluation. We detected that a lot of errors are seemingly not a result of our instrumentation but false positives. Even though *monkey* delivers the same touch events to the app before and after instrumentation, there are still unexpected errors that are seemingly unrelated to our modifications. For example, we encountered apps failing with a `SecurityException` because of missing permissions or `IllegalStateException` because the application is already initialized. We can rule out deviating behavior caused by statefulness since we make a clean install of the app before instrumenting it, thus removing any state potentially created during the first *monkey* test. While we expected our filtering approach to catch most of these cases, there are environmental factors such as relying on randomness or other kinds of non-determinism that are inherently hard to account for. Our findings therefore indicate that the filtering failed in those cases.

We conclude that our large-scale evaluation approach meets its purpose by providing an estimation on the robustness of instrumenting real-life applications. However, the degree of automation and the lack of a more sophisticated UI testing tool for Android apps introduces a certain imprecision that needs to be taken into account when working with those results.

3.7.1.2 Performance

The actual runtime overhead induced for apps instrumented with ARTist largely depends on the concrete module, e.g., taint tracking requires more injected code than permission enforcement. Therefore, we provide concrete measurements for instrumented applications in the context of our implemented use case modules in Sections 3.7.2.1 and 3.7.3.1.

3.7.1.3 Limitations

Native Code Support. *Optimizing* operates by design on `dex` input only. Targets' bundled native libraries (e.g., compiled from C/C++) that are connected via JNI are not compiled on the device and therefore neither instrumented nor inspected by ARTist. Native code components are a not only a limitation of the attacker model for our concept but are indeed an open challenge for most of the solutions in the Android security research community, e.g., code analysis as well as IRM solutions in particular. However, ARTist can be combined with existing binary instrumentation approaches [125, 56, 94, 106] to fill this gap.

Fallback to Dex. The `oat` files produced by *dex2oat* still contain the original `dex` byte code of the app to allow fallback to interpretation mode. Naturally, fallback to interpretation would render our instrumentation of the compiled `dex` byte code futile. Android 7 Nougat introduced profile-guided compilation that allows for a mixture of just-in-time compilation, interpretation, and ahead-of-time compilation based on which parts of the app are used more often (i.e., hot code paths). These profiles are even bootstrapped by delivering generic per-app profiles via Google Play that are based on averaged results from past users of the application (see Google Play cloud profiles [73]). However, our experiments confirm that, as explained in the official ART documentation [72], if there is an `oat` file available, it will be used for execution because it implies that the app was at some point explicitly compiled (e.g., because many hot code paths occurred or the device was idle and enough space was available). For ARTist, this means that we can still enforce the execution of our instrumented `oat`' files by placing them in the correct location, thereby completely bypassing the profile-guided compilation approach.

Deployment strategy. In order to create a pure application layer solution to avoid OS-level modifications, our prototype currently relies on the naïve approach of requesting elevated privileges to replace the installed app `oat` file with the instrumented version. We can eliminate this requirement by integrating ARTist with an application layer-only sandboxing solution that provides file system virtualization, such as *Boxify* [S1] or *NJAS* [30], or by resetting the execution environment and replacing loaded libraries

Tested Method	Permission	Baseline	Instrumented	Penalty
WifiManager.getConfiguredNetworks()	ACCESS_WIFI_STATE	0.681 ms	0.742 ms	8.89%
WifiManager.isWifiEnabled()	ACCESS_WIFI_STATE	0.071 ms	0.072 ms	1.18%
WifiManager.getScanResults()	ACCESS_COARSE_LOCATION	0.452 ms	0.591 ms	30.65%
BluetoothAdapter.startDiscovery()	BLUETOOTH_ADMIN	0.910 ms	0.940 ms	3.32%

Table 3.3: Microbenchmarks averaged over 60.000 runs. The *baseline* benchmarks measure the pure execution time of the permission-protected call while the *instrumented* benchmarks measure the protected call and the additional permission check.

using reference hijacking [138]. Both approaches enable the manipulation of file paths from the original to the instrumented `oat`' file at application startup time.

3.7.2 Dynamic Permission Enforcement

We briefly evaluate the performance impact of our dynamic permission module and discuss limitations.

3.7.2.1 Evaluation

The additional security checks by our module are only inserted before permission-protected SDK method calls. Thus, we cannot rely on benchmark apps because they rarely trigger the injected extra functionality. Therefore, we evaluate the performance impact of our permission-checking code using custom microbenchmarks. Table 3.3 depicts the results of our measurements for calls that are protected by three distinct permissions. The overhead encountered in the microbenchmarks ranges between 1.18% and 30.65%, showing the feasibility of our prototype if we consider that only permission-protected method calls suffer from this overhead.

3.7.2.2 Limitations

Restriction to Synchronous Calls. In order to demonstrate the straightforward implementation of an ARTist module, we opted for a simple instrumentation strategy that only covers synchronous permission-protected method calls. As a result, the current prototype does not support callbacks or asynchronicity, and its implementation should therefore be considered a proof-of-concept only.

Best Effort Permission Map. In order to direct our module to the instrumentation targets, i.e., the application's permission-protected method calls, we utilize the method call to permission mapping provided by the state-of-the-art tool *Axplorer* [28]. However, as stated on their website [51], the map is incomplete because many permission checks are not covered by their current analysis technique. Our module inherits this limitation from the used permission map.

3.7.3 Taint Tracking

We evaluate our taint tracking module in terms of feasibility, performance, and the limitations of its current prototypical implementation.

Test	Baseline	Taint-Aware	Penalty
CPU	32521	22526	30.73%
Disk	24893	20777	16.53%
Memory	3627	3346	7.74%

Table 3.4: *Passmark* results averaged over 5 runs, higher is better.

3.7.3.1 Evaluation

Runtime Overhead. We leverage an Android microbenchmark application to evaluate the performance of our prototype. Since our taint-instrumentation only affects the performance of Java code, we specifically chose the *Passmark* benchmark, which does not contain native libraries and implements all benchmarks in Java. Table 3.4 compares the results of the baseline benchmark with a non-instrumented *Passmark* app to those of an instrumented and taint-aware version. The results show an overhead ranging between 7.74% and 30.73%, which is within an acceptable range for a taint tracking approach that is not fully tuned for performance. This result is also roughly comparable to microbenchmark results of TaintDroid's interpreter-based approach, but in our case without relying on system modifications. However, as stated in contemporary literature [99], microbenchmarks are not very representative in user-driven scenarios such as Android apps. Hence, we take this result with a grain of salt.

Overall performance could be enhanced by introducing custom optimizations specifically tailored towards improving taint tracking code. One approach would be to eliminate *taint-sets* and *taint-gets* that are based on stack operations and cancel each other out, e.g., alternating *pushs* and *pops* of the same tag as seen for methods that return the return value of a method invocation. Moreover, the analysis phase allows to abstain from instrumenting apps that do not contain any global taint sinks in order to not impact performance at all in this case.

Functional Evaluation. We conducted this case study to investigate whether intra-application taint tracking can be achieved with ARTist; thus our functional evaluation focuses on detecting different kinds of data leaks in apps. However, to the best of our knowledge, there is no standardized test suite specifically tailored towards evaluating dynamic taint tracking systems for Android apps, and testing real applications is not feasible because they lack the required ground truth. In order to overcome this unsatisfactory situation, we decided to leverage an open-source suite called *Droid-Bench* [20, 121] that was initially created to benchmark static taint tracking systems. Even though this does not immediately apply to a dynamic system such as ours, we can still leverage the fact that it provides us with an assortment of applications with different but well-defined leakage behavior. Table 3.5 summarizes our module's results for those tests and categories within scope. Tests for implicit flows, inter-component communication, and reflection are omitted because they currently exceed the scope of our proof-of-concept taint tracking. As we are *abusing* the benchmark suite, we need to be careful which conclusions we draw from the test results. The first insight we gain, however, is that our case study succeeded in showing that intra-app taint tracking

Category	Successful Tests	Ratio
Callbacks	14/15	93%
Lifecycle	13/14	92.9%
General Java	14/20	70%
Aliasing	1/1	100%
Android Specifics	5/9	55.6%
Field & Object Sensitivity	7/7	100%
Overall	54/66	81.8%

Table 3.5: Results for the DroidBench taint tracking evaluation. Broken tests and categories not applicable to our system are omitted.

can be implemented as a pure application layer solution with ARTist. The second insight we derive is that, as indicated by lower results such as those for the *Android Specifics* category, our proof-of-concept does not yet catch up with previous works such as TaintDroid. Nonetheless, our work shows the feasibility of the approach and thereby lays the foundation for creating a full-fledged taint tracking system for Android versions above Android 6 Marshmallow that utilizes compiler-based instrumentation and does not require operating system modification.

3.7.3.2 Limitations

No Tracking of Implicit Flows. Like TaintDroid, our system currently does not track implicit flows (i.e., data leakage using control flow dependencies) and malevolent apps could exfiltrate data in a way that is unnoticeable by our prototype. As the TaintDroid authors discuss, mitigating leakage through control flows would require static analysis and access to the app's source code — both of which TaintDroid could not provide. ARTist, however, is already provided with the full app code and we expect interesting results to be yielded by investigations to which extent the structural program information of the IR and analytical features of the compiler backend (i.e., *Optimizing*) can help to remedy the limitations of customary taint tracking solutions on Android.

Taint Tracking Boundaries. The compiler is restricted to the app's codebase, which introduces imprecision when leaking information through SDK methods, where a *taint-set* at the caller side (developer code) but not the *taint-get* at the callee side (SDK) can be inlined. In particular, and in contrast to object types, storing primitives or strings in collections or sharing them across threads are corner cases where the taints will not be propagated appropriately. This shortcoming can be solved by using pre-computed control-flow models for framework methods [35] to generate corresponding *taint-set* and *taint-get* pairs that model the transition of data through the framework. A preferable technical solution in the future is the instrumentation of the *core.oat* image, a pre-compiled `oat` file of the framework classes that is pre-loaded into every application process. Such an approach could remove the potential over-approximations of SDK internal states in control-flow models, which could be of interest beyond taint tracking. Since the core image is created with *dex2oat* during the device startup once after each system update, it can be instrumented using *sec-compiler*. However, in either case and as in the original TaintDroid, data that already left the phone (e.g., through a network socket) cannot be tracked.

Inter-Application Communication. Our prototype is currently limited to *intra*-application tracking and lacks support for *inter*-application tracking, for instance, through the file system or Binder IPC. This opens the possibility of confused deputy [53, 41] or collusion attacks [118, 33] to exfiltrate data. Assuming that all installed apps are instrumented, a fix to this problem would be the instrumentation of the I/O method calls in order to write out taints together with the data (e.g., into a file or Binder parcel) and restore taints at the receiver side. When abandoning the requirement for a pure application layer solution, our system could also be complemented with the original TaintDroid file system and IPC infrastructure, which is unaffected by the loss of the DVM, or deploy ARTist's *sec-compiler* as the system's compiler in order to track taints across applications.

3.8 Conclusion

In this chapter, we presented ARTist, a *dex2oat* compiler-based instrumentation solution for Android that does not require system modifications. In order to be able to design and implement ARTist, we first and foremost had to thoroughly study the yet uncharted internals of the new compiler suite and in particular of its *Optimizing* backend. A deeper understanding of this compiler suite and the new ART runtime is of interest for the security community insofar as ART and *dex2oat* replaced the interpreter-based runtime (DVM) of OS versions prior to Android 5 Lollipop and hence also voided applicability of any security solution that relies on interpreter instrumentation (e.g., TaintDroid [49] and its derivatives [89, 80]). We studied the feasibility of our approach by implementing two distinct use cases. Furthermore, our case study highlights the capability of a compiler-based instrumentation framework to re-instantiate basic taint tracking for Android apps at the application layer. In general, our results provide compelling arguments such as higher robustness and better integration for preferring compiler-based instrumentation over alternative bytecode or binary rewriting approaches. We open sourced and documented the results of this work [19, 112] to allow for independent research on the topic.

4

Troop

Towards a Common Platform for Principled Dynamic Analysis of the Android Middleware

4.1 Motivation

The Android ecosystem consists of a multitude of customized Android versions (ROMs) that are all based on the official releases of the Android Open Source Project maintained by Google. This not only includes official vendor ROMs, such as those running on Android devices from, e.g., Samsung or HTC, but also ROMs that ship without the full line of services from Google (i.e, building upon the Google Play Services [71]), such as Lineage OS [130], Copperhead/GrapheneOS [37, 40], or Amazon's FireOS [10]. In consequence, Android is quickly growing due to the addition of new features for developers and users, thereby increasing the size and complexity of the system code base with every new release. This also affects Android's middleware and in particular one of its core components, the Systemserver, an always-running, privileged process that is crucial to Android's security and robustness. The Systemserver implements the bulk of Android's application framework, which provides core functionality to apps via a set of APIs, such as package management or location services. Between Android 4.1 (released 2012) and Android 10 (released 2019) the number of middleware services has more than doubled from 65 to 166, most of them being hosted by the Systemserver. As a consequence, the growing code complexity also increases the system's attack surface by adding more highly-privileged code and APIs while simultaneously raising the bar for holistic analysis of the whole code base. Over the last years, research has identified new flaws and vulnerabilities in the Systemserver and other middleware components that allow apps to elevate privileges [60] or mount denial-of-service (DoS) attacks due to, e.g., input validation errors [133, 54], concurrency bugs [90], or inconsistent permission enforcement [5, 123, 136, 83, 84]. Given its central role in the Android software stack, the increasing amount of discovered problems emphasizes the need for a more thorough robustness and security testing. However, a number of Android-specific problems and the corresponding high barrier of entry for this field have yet precluded the community from establishing a line of research that particularly aims at alleviating those problems.

4.2 Problem Description

Static and dynamic analysis have both been successfully applied to this problem domain because they come with complementary advantages and disadvantages. Static analysis provides better scalability and more formal guarantees by considering all possible execution traces, but suffers from an increased false positive rate because it considers hard-to-reach or even impossible code paths. Dynamic analysis avoids false positives by tying its results to actual execution traces, which, however, leads to scalability issues and requires a thorough code coverage of the test subject. Hybrid approaches exist where either static analysis pre-filters interesting candidates to restrict the amount of test subjects for dynamic analysis, or dynamic analysis is used to verify potential findings from static analysis. Despite the shown efficacy of both kinds of analysis in certain cases, there is a major asymmetry in the literature in terms of reproducibility and comparability. On the one hand, most static analyses on Android build upon a set of common base frameworks, such as Soot [131] or Wala [92], which allows related work to compare against and build upon each other. Or in other words: there exists a principled approach to statically analyzing Android's middleware. On the other hand, dynamic analysis is lacking such a common foundation. Instead, work on dynamically analyzing Android's middleware created a large set of mostly incompatible and highly specialized solutions. While they succeeded in implementing a multitude of analyses, such as vulnerability detection and permission mapping, their results are hard to compare or reproduce, and their solutions to shared challenges cannot be re-used as building blocks for future analyses. As a consequence, without a more principled approach to dynamically analyzing the Android system, it is unnecessarily hard to learn from those prior results, to identify more promising solutions and strategies, or to find and address shared challenges.

We believe that the driving factor for this discrepancy is the particular set of hurdles that comes with dynamically analyzing the middleware. In particular, the complex interdependencies with other Android core components prevent restarting or resetting the Systemserver in isolation. This is exacerbated by its statefulness and asynchronous lifecycle design that make it hard to isolate execution traces and requires frequent restarts and cleanups. As a result, existing off-the-shelf components from related fields, such as fuzzers or instrumentation frameworks, are often not directly applicable to testing Android's middleware, which leads to the creation of the custom toolchains that fragment this research area.

4.3 Contributions

In order to base these efforts on a common foundation, we propose a more principled approach to dynamic analysis on Android that favors reproducible results and interoperable solutions that allow other researchers to build their solutions on top of each other and compare with prior work. To make an important step in this direction, we propose a full-fledged dynamic analysis platform for Android's middleware that allows to combine and implement modular solutions to the challenges we identified from related work. This further removes the burden for future dynamic analyses of implementing the surrounding infrastructure and instead allows to focus on the actual analysis part. We showcase the benefits of our platform by implementing two use cases from related work, vulnerability detection and permission mapping, which we utilize to study and evaluate different strategies. In our comparative evaluation, we find that our more generic building blocks already uncover vulnerabilities from multiple cases that were targeted only by specialized approaches in the past. For permission mapping, we find that proven strategies, such as coverage-guided fuzzing, fail to deliver superior results in comparison to much simpler black-box alternatives. In summary, we make the following contributions:

Requirements Analysis. First and foremost, we conduct a thorough study of related work to deduce a set of requirements for a shared foundation for dynamically analyzing Android's middleware. Our results indicate that prior work repeatedly restricts itself to particular sub-areas and solves the corresponding problems in a mostly non-reusable way. While they succeed to fulfill their particular goals, e.g., detect vulnerabilities of a certain kind, often their results and toolchains do not generalize to become a foundation for subsequent research on this matter. In order to avoid these problems in the future, we compile a list of shared challenges that occur for the majority of dynamic analysis approaches, discuss available and potential solutions, and show how to solve them in a re-usable way.

Common Platform for Dynamic Analysis. Based on the requirements analysis, we propose a common platform for implementing dynamic analyses on top of re-usable building blocks for the identified challenges. We strive to create a framework that allows to evaluate different approaches against each other, which makes it possible to find *canonical solutions* instead of repeatedly solving the same problems. Furthermore, this approach can steer the community towards more reproducible results that allow future research to refer to and base upon each other. As a first step in this direction, we implemented said platform and will open source the whole ecosystem around it, currently consisting of 15 projects (see Appendix A.2 for the full list).

Case Study: Vulnerability Discovery. Following the example of prior work [91, 38, 133, 90, 60, 54, 34, 87, 97], we implemented a vulnerability discovery case study on top of our platform. We show how modular building blocks aid the creation of a full-fledged dynamic analysis-based vulnerability detection system by abstracting away many Android-specific challenges and allowing to focus on the analysis task at hand. While prior work often specialized on detecting particular classes of bugs, we succeed in implementing a more generic approach that can find instances of these bugs from multiple of those categories. We responsibly disclosed our findings to Google (see 4.6.2).

Case Study: Permission Mapping. Creating a complete mapping from Android APIs to the set of permissions required to use them is still an active area of research in the Android community. After the initial work of Felt et al. [110] that utilizes a test generator to dynamically create such a permission map, subsequent approaches primarily turned to static analysis [22, 28, 5] to alleviate many of the problems we have identified in our requirement analysis, such as the need to map the target device's attack surface or scale to thousands of APIs. However, these approaches suffered from the drawbacks of static analysis, such as the general over-approximation due to missing runtime information that, on Android, is often exacerbated by the prevalent use of statefulness and asynchronous programming patterns within the middleware.

In this case study, we show how our platform aids to alleviate those problems to an extent that—for the first time—we can properly scale dynamic analysis to the Android middleware and make an important step towards continuous and holistic testing of this crucial component of Android's architecture. Considering the advantages and disadvantages of both, static and dynamic analysis, we do not aim to replace static analysis but rather envision to combine it with dynamic components to, e.g., confirm the results of prior work and extend them.

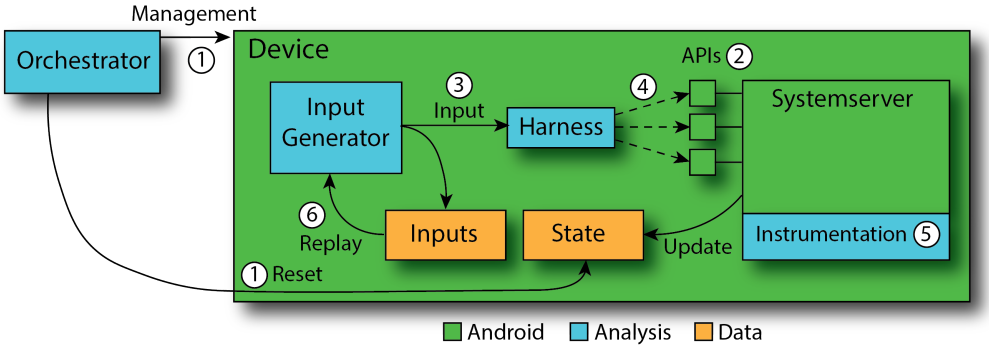


Figure 4.1: The typical structure of Systemserver dynamic analysis solutions with commonly used components.

4.4 Requirements Analysis & Taxonomy

As a first step towards a more principled approach to dynamic analysis of the middleware, we need to understand the core challenges that are shared among related work. We are in particular studying the crucial Systemserver component because certain characteristics make it a hard target for dynamic analysis. For example, the Systemserver hosts multiple inter-connected services within a single process in contrast to many single-process services in the middleware. Furthermore, a Systemserver crash triggers a soft-reboot of the device, hence re-starting is an inherently slow operation. Therefore, our work primarily focuses on overcoming the particular challenges of the Systemserver and only discusses other parts of the middleware if necessary.

Based on our survey of existing work, we extracted the critical components that are utilized to build dynamic analysis systems for the Systemserver. Figure 4.1 depicts the resulting generalized structure that can be mapped to concrete solutions by instantiating a subset of the *Analysis* components. Numbers ① to ⑥ mark different key parts that come with a particular set of challenges, which we will discuss in the corresponding Sections 4.4.1 to 4.4.6. We found that major design decisions underlying existing work in this field can be explained and categorized through examining their concrete design choices regarding these building blocks. In this section, we will introduce these six key components in more detail, compare their implementation in existing work, and discuss whether the underlying problem is solved already or still an open issue. Table 4.1 gives an overview of both academic and industry work and details a subset of their design decisions, which we will refer to in the upcoming discussions.

Table 4.1: Overview of existing systemserver dynamic analyses.

	Analysis	Devices	Freshness	Mapping	Input	Harness	Coverage	Crash Detection	Transformations	Auto-Verification	Exploit Generation
Chizpurfle (BB) [91, 38]	VD	physical	✗	SM, RE	custom	Java CLI tool	✗	LC, BD	✗	✗	✗
Chizpurfle (Evo) [91, 38]	VD	physical	✗	SM, RE	custom	Java CLI tool	✓	LC, BD	coverage	✗	✗
ExHunter [133]	VD	physical	✗	SM, RE	custom	App (Parcels)	✗	LC, RD	✗	✓	✓
ASVHunter* [90]	VF	physical	-	SA	only replay	App	✗	?	manual: add logging to Binder	✗	✓
Gong [60]	VD	physical	✗	SM (native)	custom	Binary (Binder)	✗	?	✗	✗	✓
Stowaway* [110]	PE	?	✗	SM, RE	Randoop [25]	App	✗	-	manual: log permissions	✗	-
BinderCracker (BB) [54]	VD	physical	✗	SM (native)	custom	App	✗	?	✗	✗	✗
BinderCracker (Ctx*) [54]	VD	physical	✗	SM (native)	custom	App	✗	?	manual: log binder transactions	✗	✗
Buzzer [34]	VD	?	✗	SM (native)	custom	App + native	✗	LC	include hidden SDK APIs	✗	✗
He [87]	VD	?	✗	SP, IDA (static)	custom	native (?)	✗	?	binder parcel interception	✗	✓
FANS [97][1]	VD	physical	✗	SP	own	Binary (Binder)	✗	LC,TS	ASan [17]	✗	?
Our Case Study Prototypes											
Vulnerability & Bug Hunting	VD	emu	per API	SM (native), RE, SA	AFL [139],Gong* [60], Chizpurfle [91, 38]	Binary (Binder), Java CLI tool	✓	LC,PM,IN	coverage, crash detection	✓	✓
Permission Mapping	PE/VF	emu	per API	SM (native), RE, SA	Chizpurfle [91, 38]	Java CLI tool	✓	LC,PM,IN	coverage, log permissions	✓	✓

Legend:
Analysis: *VD*: vulnerability detection, *VF*: verification, *PE*: permission mapping
Freshness: How frequently device state is reset to avoid influencing the testing of another target.
Mapping: *SM*: Service Manager, *RE*: Reflection, *SA*: static analysis, *SP*: source parsing, *IDA*: IDA Python plugin
Coverage: Whether coverage feedback is utilized.
Crash Detection: *LC*: Logcat parsing, *BD*: binder death notifications, *RD*: app-based reboot detection, *PM*: process monitoring, *IN*: instrumentation, *TS*: Tombstone monitoring

[1] FANS explicitly targets native services only but shares many problems with Systemserver dynamic analyses.

4.4.1 Target Instance Management

One of the aspects that sets Android apart from other platforms is the requirement for proper target instance management. In the domain of binary analysis, dynamic approaches, such as concolic execution or fuzzing, are most efficient when aiming at targets that adhere to a basic set of properties:

1. Re-starting the binary, which is required to keep executions independent, needs to be fast.

2. The binary should be stateless to avoid dependencies between different inputs.

3. The execution should be synchronous to avoid code being executed because of an old input while a new input is processed.

4. Running the binary multiple times in parallel should be supported to speed up the process.

Even if these properties do not apply, previous work has shown how to overcome these problems, e.g., by running the binary in a restricted environment [119], such as an emulator, that can be reset quickly, or forcing the execution environment to behave more deterministically [59].
Android's Systemserver, in contrast, does not fulfill these requirements. First, it cannot be started independently because of complicated interdependencies to other Android components. It can only run on an actual device or full emulator, therefore restarting it requires to reboot large parts of the Android software stack. Second, because the Systemserver keeps a lot of state to ensure the proper functioning of the whole application layer (e.g., state of app components like `Activities` or `Services`), cleaning the current state requires a whole reset of the device or emulator. Third, the Systemserver makes heavy use of asynchronous communication patterns [28], which complicates isolating single executions. And fourth, running multiple instances of the Systemserver only scales with more physical devices or emulators where available, which results in poor performance in comparison to running multiple copies of a binary. Therefore, managing the Systemserver as a target for dynamic analysis requires specialized solutions.
With these differences in mind, we notice that working with the Systemserver roughly resembles API testing with multiple entry points (see also Section 4.4.4) where target re-starts are slow, executions are asynchronous, and tracking the state in order to reset it requires involved and dedicated analyses.

4.4.1.1 Existing Work

To the best of our knowledge, all existing work in Table 4.1 uses physical Android devices for their evaluations (column **Devices**). Our survey of existing works indicates two main reasons why the authors could not use emulators despite superior scalability: First, Chizpurfle [91, 38] explicitly targets vendor ROMs that typically do not run on emulators because of vendor-specific closed-source patches. Second, ASVHunter [90], Stowaway [110], BinderCracker [54] and Buzzer [34] require manual AOSP patches to work properly, hence only open-source ROMs are supported. While those could have been launched in custom emulators, we speculate that the lack of a tool for cloning and managing emulators built from AOSP might be the driving force behind relying on physical devices in this scenario. Furthermore, the column **Freshness** in Table 4.1 shows that no previous work isolates test cases from each other by cleaning the device or even discusses state management, which can be implemented completely independent of the actual analysis. The FANS paper [97] even reports cases where the testing was interrupted by a bricked device so that the authors had to factory reset the OS and restart the experiments.

Our Assessment

The current state-of-the-art fully relies on physical devices for testing. While these are indispensable for analyzing vendor-specific code, we strongly recommend to additionally support emulators, if only for testing the common AOSP code base. Emulators directly scale with computational power so properly equipped servers can run 30 or more instances in parallel (see Section 4.6.1.2). Using emulators that are based on QEMU [113] further allows for convenient and fully automated resets of the Systemserver's state so that we can force a reboot with a clean filesystem. Resetting a physical device in contrast requires a time-consuming factory reset.

In order to automate the management of emulators, i.e., creating Android images from built AOSP trees, creating clones with these images, and resetting them to a clean state, we created a tool called *avd-tool* that will be open sourced as a part of our evaluation platform (see Appendix A.2). For newer Android versions[2], however, Google created the cuttlefish [65] emulator that runs images created from AOSP trees. Our platform supports both types of emulators, classical as well as cuttlefish.

> To cover different use cases, our platform supports both, physical and virtual devices, and exposes their configuration via APIs (e.g., number of instances, frequency of refreshs).

4.4.2 Attack Surface Mapping

The concrete attack surface exposed by the middleware completely depends on the currently running ROM. There are at least two dimensions to consider: First, in addition to AOSPs base set of system services and APIs that are required for Android to work properly (enforced by Google through their Compatibility Test Suite [64]), vendors further extend Android with new services and APIs, which led to an increase in flaws and vulnerabilities [134, 143] in the past. Second, the number of system services of AOSP increases steadily with the introduction of new major Android releases. For this reason, dynamic analyses of the middleware requires a mapping of which APIs are accessible for the ROM under test to decide which ones to target.

Table 4.2: Categorization of different API surface mapping approaches in terms of the used analysis method and resulting API description.

	Transaction IDs	**Method Signatures**
Dynamic	Native Service Manager [60, 54, 34]	Java Service Manager [91, 38, 133, 110]
Static	Source Code Parsing [87, 97]	Class Hierarchy Analysis [90, 97]

Table 4.3: Example of a fully represented method in the Android middleware including high-level Java signatures (interface *and* implementation) and low-level binder service and transaction ID.

Identifier	**Example**
Interface	android.webkit.IWebViewUpdateService
Implementation	com.android.server.webkit.WebViewUpdateService$BinderService
Method	changeProviderAndSetting
Parameters	java.lang.String
Binder Service	webviewupdate
Transaction ID	3

Existing Work

The row **Mapping** in Table 4.1 lists the different approaches to API mapping by existing work. Table 4.2 further categorizes these approaches by two dimensions: the collected method identifier (**Transaction IDs** and **Method Signatures**) and the required kind of analysis to obtain these identifiers (**Dynamic** and **Static**). In addition, Table 4.3 shows all representations of an example method in the Systemserver.

1. Native Service Manager. Gong [60], BinderCracker and Buzzer bootstrap their service discovery by first acquiring a handle for the global service manager—the central system service registry—and querying it for the list of all currently registered system services. The returned list contains binder service names and, if present, an interface descriptor. Based on this information, it is now possible to collect the list of all eligible low-level transaction IDs (i.e., for all accessible APIs on the device) that are required when directly talking to the binder driver in order to invoke the target API. However, neither the Java method signature including parameters nor the implementing class are known at this abstraction level.

2. Java Service Manager. Similarly to the native approach, Chizpurfle, ExHunter [133], and Stowaway make use of the service manager, this time in the Java domain. The obtained list of service interfaces is utilized to enumerate all API methods and their fully-qualified signatures. In contrast to the native service manager approach, we do

[2] The declared goal is to support all API levels after 28 [65].

not need transaction IDs because we can directly issue remote procedure calls to service methods using reflection.

3. Source Code Parsing. He [87] shows how to heuristically find the correct source code files that implement system services and parse them for transaction IDs. In a quick experiment (see *transaction-id-mapper* tool in Appendix A.2), we were able to replicate this with a simple python script that derives the name and path of the AIDL-generated [46] stub code from the interface name. While this works great for Java-based services, native services require different heuristics, but the online material from He does not provide any details on this. Unfortunately, this part would be most beneficial since the native service manager can only provide transaction IDs but no method names or parameters. FANS solves this problem with a Clang [23] plugin that analyzes the code of native services at compile time to derive the transaction IDs. However, the mapping is restricted to native services and therefore excludes services hosted by the Systemserver.

4. Class Hierarchy Analysis. ASVHunter uses the Soot framework to statically analyze the Systemserver's bytecode and enumerate exposed methods of system services including fully-qualified signatures, which has been demonstrated before in the literature [22, 28, 84]. FANS' Clang plugin uses a similar approach to find the full signatures of native non-Systemserver APIs.

Our Assessment

We compare the four established strategies by two aspects: which method representation they are collecting and under which circumstances they are applicable.

The low-level binder representation is typically used by native tools that directly talk to the binder IPC driver and is available for Java-based and native system services. However, at this level all typing information is lost, causing a semantic gap between the transaction IDs and their corresponding service API signatures, which leaves these tools either with blindly creating byte streams or requiring auxiliary information about the structure of their target. Java method signatures, in contrast, come with full package names and typing information that can be utilized to create structured inputs, but are only available for Java-backed system services.

This higher semantic insight further allows to use the signature information with, e.g., instrumentation tools, and to invoke Systemserver Java APIs via reflection. Applicability of these mapping approaches, however, depends on whether dynamic or static analysis is used to gather the API methods. The dynamic approaches **Native Service Manager** and **Java Service Manager** are always available because they utilize APIs that are used by the Android system itself and therefore guaranteed to be present on every Android device through the Compatibility Test Suite [64]. ASVHunter's approach to statically analyze the Systemserver for entry points (**Class Hierarchy Analysis**), which is also used in other Systemserver-related work [22, 28, 84], is broadly applicable because it operates on compiled bytecode, which is also available on Android by default, but it can only cover Java-based APIs. He's **Source Code Parsing** approach is

only feasible if the code is available, so it is restricted to AOSP versions and its open source derivatives, which explicitly excludes vendor ROMs like those employed by, e.g., Samsung or Huawei. In cases where it is applicable, however, it can also provide higher semantic insight by mapping transaction IDs to method signatures.
In conclusion, the high-level approaches **Java Service Manager** and **Class Hierarchy Analysis** work equally well for analyses targeting only Java APIs and the low-level approach **Native Service Manager** is only suitable when directly talking to binder. The approach **Source Code Parsing** could in theory provide both, but only heuristically and for open source ROMs.

> We provide both mappings, reflection-based high-level signatures and low-level binder transaction codes, to support tools on the Java and the native layer.

4.4.3 Input Generation

Input generators are an integral component of dynamic analyses because they generate the test input that drives the actual executions. We examine existing work with a focus on two particular aspects: First, we want to determine possible goals for middleware dynamic analysis to pursue. Second, we are interested in which input generation strategies are most beneficial for these goals.

Existing Work

According to column **Analysis** in Table 4.1 existing work is partitioned into three groups: vulnerability detection, dynamic verification, and permission mapping.

1. Vulnerability Detection. The largest group with nine different solutions from existing work focuses on detecting bugs and vulnerabilities. We notice that all systems rely on generating and detecting crashes in order to trigger abnormal behavior. Gong, both versions of BinderCracker, Buzzer, He, and FANS target APIs that are backed by native code, which is prone to a large set of low-level flaws that can lead to severe vulnerabilities as demonstrated by the amount of CVEs in this area [132]. ExHunter and both versions of Chizpurfle, however, focus on Java-based APIs, where most crashes rather lead to denial-of-service flaws due to Java's memory safety. As depicted in column **Input** in Table 4.1, all vulnerability detection analyses ship their own custom fuzzers as input generators, which we categorize in two dimensions as depicted in Table 4.4. The first dimension, structural awareness, describes whether an input is generated according to certain structural information about the target API. Typical examples are grammar-based fuzzers, where grammars are either generated automatically or encode expert knowledge. The second dimension describes whether the generator utilizes a feedback loop. Coverage-guided greybox fuzzers, for example, are commonly built by measuring code coverage at runtime and feeding it back to the fuzzer that can base the generation of new inputs on how *successful* previous ones uncovered yet unknown code segments. Black box fuzzers, in contrast, do not utilize feedback such as coverage information from the program, so the generation of new inputs is independent of the *success* of previous ones.

Table 4.4: Categorization of evaluated fuzzers by structural awareness and utilization of coverage feedback.

	structure-aware	**structure-unaware**
coverage-guided	Chizpurfle Evo	AFL Evo
black box	Chizpurfle BB	Gong/RandFuzz, AFL BB

2. Dynamic Verification. ASVHunter utilizes static analysis to produce a set of candidate findings that are subsequently verified dynamically. Static analysis is often prone to generating false positives because it resorts to over-approximation when runtime information is required[3], so a dynamic complement is created to test whether these candidate inputs are actually triggering the expected behavior. In contrast to utilizing fuzzers, there is no need for massively generating inputs as we are working with a known set of candidate inputs from static analysis.

3. Permission Mapping. Permission mapping, as done by Stowaway, focuses on triggering as many permission checks as possible to be able to enumerate and map them. Their approach of using a test generator is comparable to employing a fuzzer. However, while vulnerability detection aims for *deep* coverage of the target under test, permission checks are often placed at the beginning of API methods (*fail early* principle) [5], so thorough code coverage is not required. Even though Stowaway opened this line of research by following a dynamic approach, follow-up work [22, 28, 5] was exclusively conducted using static analysis.

Our Assessment

Goals. When comparing Systemserver dynamic and static analysis, we notice a large discrepancy between the set of pursued goals. While we found three dominant use cases for dynamic analysis in the literature, static analysis covers a much broader scope. Goals of Systemserver static analysis also include, e.g., detecting confused deputies in the codebase [83] and inconsistent policy enforcement [123, 84, 6, 4]. Besides one case where dynamic components are added to complement the static analysis [90], hybrid or pure dynamic analyses have not been applied to those topics yet. In the future, we hope to see more of these use cases also pursued using dynamic analysis.

Strategies. Related work in Table 4.1 that focuses on vulnerability discovery exclusively employs fuzzers as their input generators. They are easy to implement and experience has shown that they work well for producing crashes in other fields [125, 126, 119, 21, 114, 96]. However, there are no prior studies on whether fuzzers are superior to alternative input generation approaches for analyzing the Android middleware, or which strategies or design decision are the best fit for certain goals.
Whether more advanced strategies or completely different input generators can provide major benefits is an open but promising research question that has not yet been explored

[3] E.g., in case of the asynchronous *Handler* pattern in the Systemserver [28, 83].

in depth. However, a common platform would allow future work to evaluate these approaches under the same conditions in order to produce comparable and reproducible results.

To satisfy different analyses' input generation requirements, we integrated multiple kinds of input generators into our platform as re-usable building blocks (see Section 4.5.6) and provide APIs to integrate more.

4.4.4 Target Communication & Harness

In other domains, such as binary testing, dynamic analysis systems directly feed input to a target by providing command line arguments, setting environment variables, using `stdin`, or executing code from within the target's context. However, step ④ in Figure 4.1 illustrates that in the middleware analysis scenario we have to resort to techniques from API testing to combine input generators with so-called test harnesses. These harnesses have to match inputs to the API's required format and are responsible to bridge the process boundary between input generators and actual targets by transparently forwarding all generated inputs.

Related Work

Similarly to the problem of mapping the attack surface in Section 4.4.2, there are approaches on at least four different abstraction levels:

1. Native Binder Interfaces. Gong, Buzzer, He and FANS directly talk to the binder driver through the exposed native binder interface and therefore the test generator can immediately feed marshaled parameter structures to the API under test. While this approach provides superior performance because many abstraction layers are simply skipped, the semantic insight is rather low because auxiliary information such as types and structure of the target API's expected parameters are unknown.

2. Android SDK Managers. App developers usually interact with the middleware by using SDK managers that provide stable APIs while handling communication with the corresponding system services in the background. This abstraction allows to input actual Java objects instead of marshaled byte buffers at the cost of additional marshaling of these inputs. The main caveats, however, are that the SDK implements client-side checks that prevent apps from common mistakes, which can be bypassed by directly talking to the system services, and that the APIs exposed by the SDK do not necessarily match those of the corresponding system services. Therefore, the SDK API does not reflect the actual attack surface of the underlying system services and should therefore be avoided in most scenarios.

3. Reflection on Service Binders. Similar to the native binder interface, there are also Java-based APIs that allow to query for all registered binder services in the system and interact with them. Chizpurfle, BinderCracker, Stowaway, and ExHunter use the resulting service binders in order to operate on the semantic level of Java objects, which, however, adds the increased complexity and overhead of reflection and is only supported for Java-backed services. Using reflection bypasses the above mentioned SDK managers and allows to directly talk to the system services while transparently handling the inter-process communication.

4. Intra-Process Communication. Communicating with middleware services without going through an IPC protocol requires to run the input generator in the target's process space. This is the fastest alternative since it completely skips the marshaling procedures on both ends of IPC calls. However, it comes with robustness issues (generator crashes with target), needs invasive changes to the OS to be deployed, and might incur an increased false positive rate because there is no guarantee that candidate inputs that expose unexpected behavior would have made it through the marshaling step.

Our Assessment

Utilizing native binder interfaces and reflecting on binder objects have both been implemented by existing work. With respect to the discussed problem of additional checks and a different attack surface, the SDK should not be used for communicating with system services. Reflecting on service binders avoids these problems and is therefore better suited for testing the actual attack surface. Intra-process communication could be implemented by, e.g., integrating *Libfuzzer* [24] into the middleware by either changing the source code[4] or injecting it using an instrumentation framework. However, it might be required to collect and feed back the coverage information by unusual means because *Libfuzzer's* default way of collecting coverage — recompiling the target with the LLVM [26] compiler infrastructure — is not always possible here.

> Since related work operates on the native *and* the Java layer, we implement harnesses for our integrated input generators to support both layers.

4.4.5 Instrumentation & Introspection

Dynamic analysis is typically supported by instrumentation or introspection tools, e.g., to provide coverage feedback, detect crashes, patch out obstacles, or any other code transformations that improve analysis results or performance. Often, these tools are used to compensate for shortcomings of dynamic analysis, such as non-determinism when executing code paths with randomness or dependency on accurate coverage information. In order to provide this functionality when targeting involved systems, such as the Systemserver, the following requirements need to be fulfilled:

1. Low overhead. Maintaining a certain level of performance ensures that transformations are not considerably slowing down the testing process. Fuzzers, in particular, are optimized for high throughput of inputs and most transformations incur extra costs per execution trace.

[4] There is restricted support for building *parts* of AOSP with *Libfuzzer* support [68].

2. Fine granularity. Some instrumentation frameworks allow for surgical changes instead of replacing whole blocks of code. In contrast, popular hooking frameworks, such as XPosed [135], can only redirect method calls to custom functions. This approach falls short in case we want to change a minor detail in a targeted method, such as inverting a single condition or changing a constant. Those use cases require more fine-grained instrumentation capabilities.

3. Scalability. When working with a large codebase, scalability is a major issue because many frameworks are mainly used for instrumenting single applications or smaller binaries. Instrumentation and introspection systems need to scale to the Systemserver without drastically reducing robustness or performance.

Existing Work

To which extent instrumentation and introspection capabilities are used by prior work is visible from the columns **Crash Detection**, **Coverage**, and **Transformations** in Table 4.1.

Crash Detection. Any dynamic analysis that specifically targets vulnerabilities and other robustness issues requires solid crash detection mechanisms to be able to observe potentially interesting behavior. Chizpurfle uses the binder death notification mechanism [101] to receive notice as soon as its currently targeted system service crashes so that the offending input can be flagged accordingly.

An alternative approach to detecting crashes and other unwanted behavior is to parse Android's logcat output, as done by Chizpurfle, ExHunter, and Buzzer. The logcat facility can be accessed directly from on-device code or from the outside by using the Android Debug Bridge (adb) [45]. While it is simple to implement and, when done outside of apps, robust to Systemserver crashes, it requires heuristics for parsing potentially interesting log output and is incomplete because it misses silent failures that do not produce log entries.

Furthermore, FANS uses the ASan support of more recent AOSP versions [17] to detect crashes in native services and ExHunter uses an installed application to detect reboots as communicated through a system-wide broadcast, which indicates a past Systemserver crash.

Coverage Feedback. Chizpurfle uses the Frida [106] instrumentation framework to deploy its patches. Frida's stalker component [107] is utilized to implement coverage tracking for each produced input and the resulting execution trace by rewriting code blocks at runtime. The injected code reports all basic blocks and branches taken by the last input back to Chizpurfle where the information is used for generating new inputs. The capability to deploy code instrumentation at runtime allows to run Chizpurfle on arbitrary rooted Android versions and devices, but it also incurs a heavy slowdown of 1291% [91, 38].

Further Introspection. There is a whole category of code transformations for introspection that are applied to assist dynamic analysis. Most existing works have resorted to patching AOSP by hand to obtain the runtime information required for their analyses to function. ASVHunter, BinderCracker, and He patch the binder framework code to intercept IPC messages to allow for logging and mutation. Stowaway modified AOSP to record checked permissions in order to create a permission mapping and Buzzer re-compiled the SDK to include APIs normally hidden from developers. The major problem with hardcoding changes into AOSP by hand is that it explicitly ties every solution built on top to a particular Android version that additionally needs to be open source. Porting this to newer versions results in additional work and closed source vendor ROMs cannot be supported at all.

Our Assessment

Given the large range of transformations seen in other fields of dynamic analyses (in particular binaries), such as transformational fuzzing [109] or de-randomization [126, 59], related work in the field of Systemserver analysis has barely scratched the surface of the current state-of-the-art. While coverage-guided input generation and greybox fuzzing are well-established in the binary domain, most existing work in our scenario uses black box approaches with the only exception being Chizpurfle, which comes with a significant performance overhead. We see a lot of potential in this field if we are able to reduce the technological gap to the state-of-the-art on other platforms. The major blocker we identified is the lack of an easy-to-use, well-established introspection and instrumentation framework that allows to implement the above mentioned transformations. In the binary domain, there are multiple well-known frameworks that support these kinds of instrumentations, such as LLVM at compile-time, *angr* patcherex [125], pwntools [56], and many others for static code patching, or Intel PIN [94] for dynamic instrumentation. On Android, a lot of tools have focused on in-process method hooking to support inline reference monitors for apps, such as Dr. Android & Mr. Hide [95], I-ARM-Droid [43], retroskeleton [42] or AppGuard [29], but these do not fulfill the granularity requirement. Another popular option on Android is to transform bytecode to the so-called smali representation with a tool called *ApkTool* [93] where a human analyst can manually change bytecode instructions to a certain extent, which does not fulfill the scalability requirement. To the best of our knowledge, the only frameworks that scale to the Systemserver and at least to some extent support fine-grained instrumentation capabilities are Frida and ARTist (see Chapter 3). While the former's main advantage is its support for native code, which ARTist does not cover at all, robustness issues and the large performance overhead make it a less-then-optimal alternative for implementing performance intensive tasks, such as computing code coverage. Since instrumentation support is crucial for a common platform because systems such as Chizpurfle and AFL [139] depend on it, and in an effort to reduce the gap to other ecosystems, we implemented the following two-fold approach: First, we make use of ARTist's Systemserver instrumentation support (see Chapter 3) to provide low-overhead modification and introspection capabilities. Second, we designed, implemented, and evaluated multiple re-usable instrumentation and introspection modules commonly used by existing work for our platform (see Section 4.5.5.2). Similarly to how *angr* [125] provided a unified platform to implement binary analyses, future work on Systemserver dynamic analysis can be built on top of our platform to re-use existing solutions to known problems, thereby reducing the barrier to entry and duplication of effort.

Our platform provides a set of re-usable introspection modules that cover a large set of requirements from related work, and additionally integrates with ARTist for more specialized requirements.

4.4.6 Verification of Results

In contrast to static analysis that argues about every possible execution trace, dynamic analysis derives its results from actual executions and is therefore less prone to false positives due to impossible code paths. However, often false positives cannot be avoided completely when working with concrete execution traces in real-world environments. Typically, dynamic analysis tends to report false positives when the environment is highly dynamic and non-deterministic, which includes the usage of randomness, other actors within the same system (e.g., other apps changing state in the middleware), or occasional background operations being triggered automatically. In order to pre-filter results for human analysts or to even achieve fully autonomous analysis systems, a set of solutions has been proposed to programmatically deal with these problems. Typical examples are automated confirmations for crashes found by fuzzers through systematically replaying previous inputs and applying test case minification (as done by, e.g., AFL). While this has been well-studied for robustness testing and vulnerability detection, other dynamic analyses have more specialized requirements for such automated verification systems. We examined existing work to learn whether such techniques are currently used for Android middleware dynamic analysis and how far these could be automated.

Existing Work

As depicted in the column **Auto-Verification** in Table 4.1, most related work describes their result verification process as manual, with two notable exceptions. ExHunter fully automates the process of finding so-called `UncaughtException` denial-of-service attacks [133] against the Systemserver by detecting device reboots and replaying the offending input to verify if the system reboots again, thereby creating proof-of-concept exploits for these inputs. Similarly, the dynamic part of ASVHunter focuses on verifying potential denial-of-service flaws by triggering the candidate input and observing the result.

Our Assessment

We again observe a gap between the state-of-the-art in other ecosystems and our middleware scenario. Currently, the more advanced techniques, such as minification, bug triaging, or exploitability analysis, and the straightforward result verification technique of directly replaying candidate inputs are only deployed in two systems.

We argue that a common platform to implement and share those building blocks could improve the current situation by facilitating the integration of automated verification into dynamic analyses, which would be beneficial in terms of reproducibility. Therefore, our platform provides out-of-the-box support for reproducing interesting behavior by persisting all generated inputs and automatically replaying them on fresh Android instances.

4.5 Architecture

Given the outlined challenges and the (partial) solutions provided by related work, we base the design of our dynamic analysis platform on these observations and show how to overcome the corresponding challenges in a modular and reusable way. This allows us to directly compare different building blocks and their strategies. We can then answer questions, such as whether the approach used by related work was the *best-possible* one, or if re-using off-the-shelf components like AFL, which gave outstanding results for binary analysis, will provide similar results in the domain of Systemserver analysis.

4.5.1 Overview

Figure 4.2 gives an overview of our platform. The clients request tasks ① (in this case sets of APIs to be analyzed) from a backend component and distribute them among their workers. An arbitrary number of clients running on different machines with varying amounts of workers attached to them are supported so that computational resources can be added and removed easily. The client's core is the manager component called Troop that distributes the work to locally running worker processes and takes care of monitoring all workers and exposing information about their current state ②. Each worker is associated with one task that is analyzed at a time. In our current design, each worker is responsible for exactly one (emulated or physical) device instance where it performs the analysis task at hand ③ and creates corresponding artifacts and results ④. Additionally, a result analyzer component can interpret the worker's artifacts to detect findings, compute success rates, and provide further insights ⑤. The whole system is specifically designed for modularity, because its components are loosely coupled, and a high degree of automation to ensure scalability.

4.5.2 Troop

Troop, the namesake of the overall platform, is the central entity that runs on each client and takes care of bootstrapping the whole process. Troop functions as a central observer that spawns all worker processes, continuously feeds them with new work from the backend, monitors their execution, and takes care of other issues, such as logging and management of generated test artifacts. The live data collected from its workers are exposed via a simple flask-based [55] webserver that describes the workers' current states in both a human-readable HTML format and a machine-readable JSON dump. The management code is decoupled from the actual worker implementation that is used

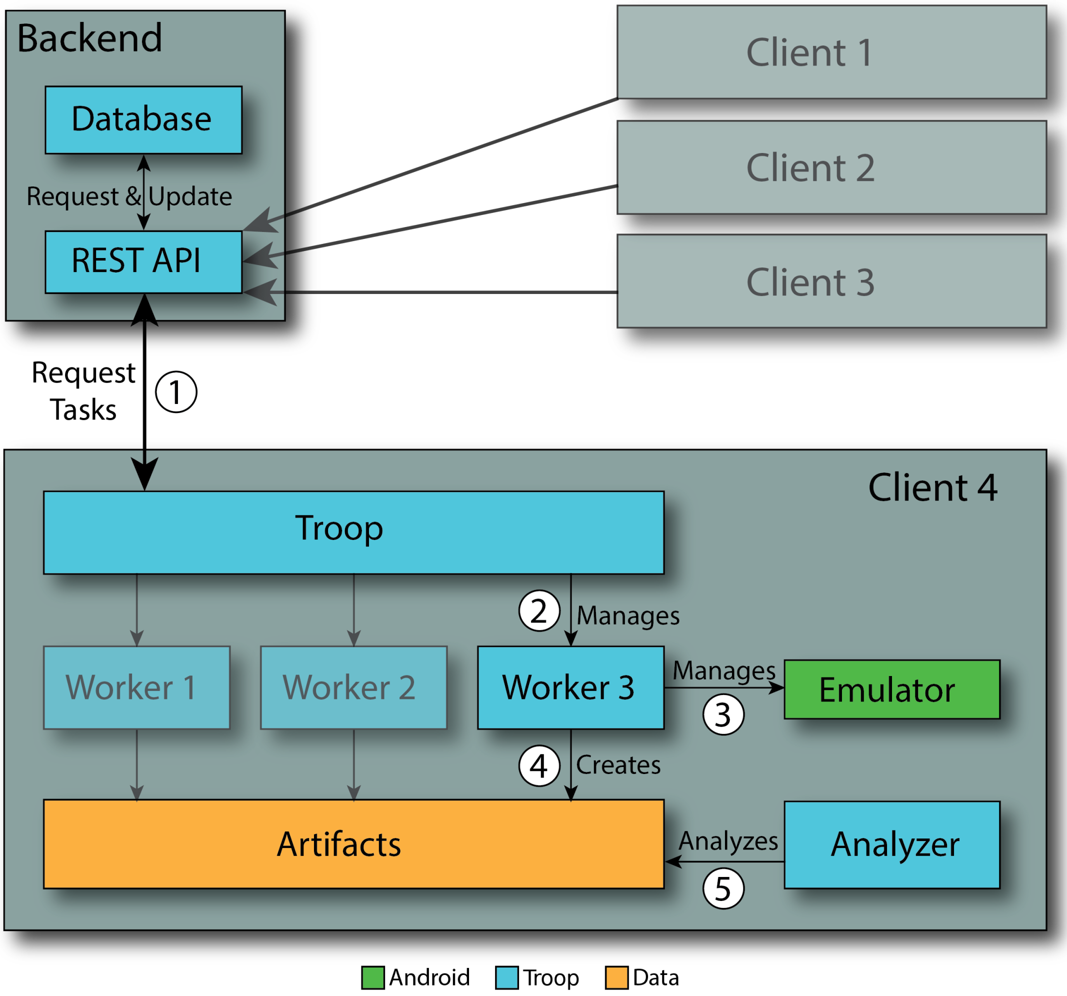


Figure 4.2: Overview of the platform's architecture.

for the current evaluation, so Troop operates independently of whether its workers are fuzzing their targets, confirming bug candidates, or performing any other analysis task.

4.5.3 Result Analyzer

Our framework ships with a set of result analysis components that can be used and extended to reason about the current state of dynamic analysis evaluations and their results. Our result analyzer modules scan through the artifacts generated by workers to produce summaries of the results. These analyses can be invoked from the command line to get an idea about the current state of the evaluation but they are also used by Troop and the workers themselves for decision making. Typically, during the design of a new evaluation, interesting observations and bugs are encoded as new analyzer commands to track them as the system evolves. All results presented in Section 4.6 are based on those result analyzer outputs.

4.5.4 Backend

The backend is a small web server that exposes the list of tasks stored in its database via a REST API. The most common cases are clients requesting new tasks for their workers to test. The backend remembers tasks currently under test (i.e., which ones have been finished) and a list of open tasks, thereby serving two main purposes:

1. Synchronizing all existing clients and workers so that we can ensure no task is executed by multiple workers at the same time (unless we explicitly want that).

2. Controlling the actual definition of a task, i.e., deciding on the subset of middleware APIs that are tested at a time by a single worker. This could be a single API or all APIs of a particular service.

4.5.5 Workers

Workers implement the central logic of an evaluation because they directly perform the required analysis tasks. After being started by Troop, they continuously consume tasks from their input queues and, depending on the kind of task, execute their functionality. There are no restrictions imposed by the framework on what kind of analysis or testing can be applied, so whether the worker runs a fuzzer, replays input, or uses a completely different input generator, is up to the concrete evaluation at hand. Workers alleviate the test case isolation and statefulness problems from Section 4.4.1 by launching *fresh* instances of prepared emulators and managing their lifecycle including setup and teardown, so dynamic analyses only need to decide *how often* to reset the testing environment, e.g., for each new tested API or after a fixed time budget is exhausted.

4.5.5.1 Workflow

Figure 4.3 depicts the typical worker pipeline for an input generator-driven analysis, in this example a fuzzer. The worker starts by launching a *fresh* emulator instance ①. After the system booted successfully, the input generator is launched to continuously

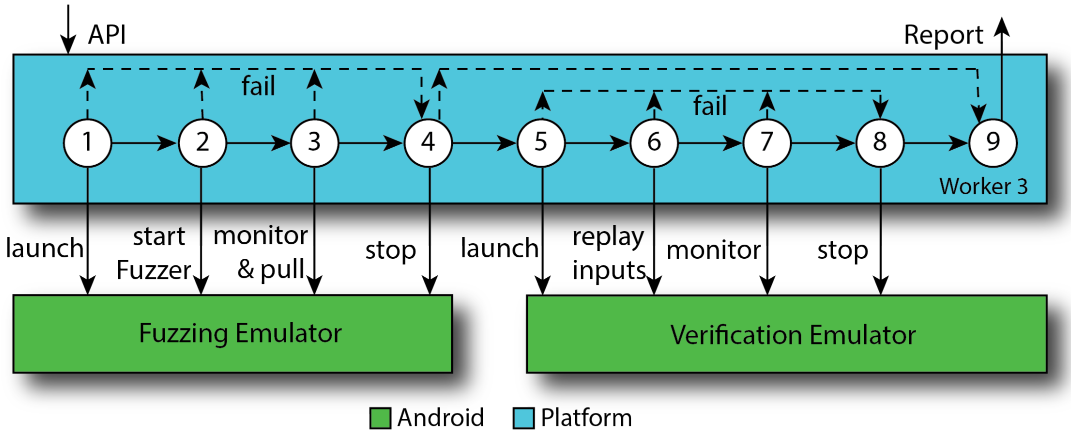


Figure 4.3: Worker processing an API task.

send inputs to the current API[5] on the emulator ② while the worker regularly checks the status of all involved parties ③, e.g., if the input generator is still running and the system is behaving as expected. Furthermore, it keeps pulling snapshots of the current state of all platform components running on the emulator. After a predefined timeout or as soon as we detect a potential finding, a snapshot is pulled and the whole emulator is shut down ④. In the next step, we try to verify the potential problem on vanilla emulators ⑤. Since our goal is to confirm whether the collected input list reliably reproduces the observed behavior, we minimize changes to this emulator by only using non-intrusive monitors (e.g., off-device monitors). The list of previously generated inputs is replayed ⑥ and the system observes the emulator to decide whether the behavior is reproducible or a false positive ⑦. Our verification workflow immediately outputs test cases that reproduce the finding. Finally, emulators are closed ⑧, files are backed-up, a report is written ⑨, and the worker starts all over with the next API.

4.5.5.2 Monitoring

In order to fulfill the requirements from Section 4.4.5, we employ a set of introspection modules for monitoring the system's current state to, e.g., detect crashes or collect coverage information. These modules check for interesting behavior as the result of the currently executed dynamic analysis and confirm that all framework components are still alive. To this end, we use five different techniques:

1. **Logcat** is constantly parsed for crashes, stack traces, and other markers of unwanted behavior.

2. **Binder death notifications** are utilized to stay informed about dying system services.

3. **Checking the list of running processes** ensures we see crashes of critical Android services and platform components.

[5] For the sake of simplicity we analyze one API at a time for now.

4. **Uncaught exception handlers** for all vital Systemserver threads are injected via instrumentation.

5. **Coverage tracking** is implemented by instrumenting target methods to compute basic block and edge coverage[6].

These five monitors effectively ensure that we can keep track of the target's current state to reason about the dynamic analysis' progress.

4.5.6 Integrated Input Generators

In Section 4.4.3 we learned that most middleware dynamic analyses from related work use custom fuzzers tailored to their particular case. This not only leads to duplicate work but also misses out on utilizing improvements from the active community that has formed around advancing and optimizing general-purpose fuzzers (see Section 4.4.3). Therefore, we decided to utilize the modularity of our platform to integrate a collection of different fuzzers that can be used in dynamic analyses, effectively freeing future work from re-inventing the wheel over and over again. This particularly allows to compare the fuzzer's performance in different dynamic analysis scenarios to be able to pick the best-fitted input generator. In the following, we will discuss the three different fuzzers and how they are integrated into our platform.

4.5.6.1 Chizpurfle

We decided to integrate Chizpurfle into our platform because it not only implements a single specialized fuzzer but intends to be a generic boilerplate for experimenting with different fuzzing techniques, such as black box or evolutionary algorithms. Since it already targets Systemserver APIs, we do not need a harness and can directly instruct it to test particular APIs.

Coverage Channel Chizpurfle utilizes the Frida framework to trace executed code at runtime and provide coverage feedback to the fuzzer. However, we decided to use a more lightweight instrumentation framework instead for two reasons: First, the authors report an overhead of 1291% introduced by Frida [91, 38], which tremendously slows down the fuzzer. Second, in our experiments, we could not make Chizpurfle's Frida code run on the `x64` emulators we are utilizing, hence running the original version would restrict us to real ARM devices in contrast to a large number of emulators that we can run in parallel (see evaluations in Section 4.6.1.2). Since our coverage monitor from Section 4.5.5.2 is implemented using the ARTist instrumentation framework, we extend our ARTist module by additionally opening a local socket that the fuzzer can connect to and re-implement Chizpurfle's protocol to establish compatibility. The feedback channel is then used to transparently send the collected coverage back to the fuzzer in the expected format.

[6] Implementing other coverage metrics discussed in the literature (e.g., path sensitive [58]) is also straightforward.

4.5.6.2 RandFuzz (Gong*)

With Chizpurfle as an example of a feedback-directed greybox fuzzer operating on the Java layer, we decided to also integrate a low-level fuzzer that sends marshaled input parcels via native binder APIs. This allows to not only target Systemserver's Java APIs but also other middleware services that are created from native code. Under the name RandFuzz, we chose to re-implement Gong's approach of a random black box fuzzer based on the information from [60].

Implementation & Integration From the available information, we implemented RandFuzz to first list all accessible services by querying the native service manager (see Section 4.4.2). RandFuzz will then generate random payloads for binder transactions to those services. The fuzzer is executed by providing the target services and transaction IDs.

4.5.6.3 American Fuzzy Lop (AFL)

In Section 4.4.3 we learned that, with the exception of Stowaway, no related work makes use of existing input generators but instead creates their own fuzzer from scratch. In order to showcase how to utilize state-of-the-art tools for analyzing the middleware, we integrate the well-known American Fuzzy Lop (AFL) into the platform and describe the necessary steps to adapt it to our domain.

Bridging the Process Gap AFL is a typical example for the challenges described in Section 4.4.4: It forwards generated inputs directly to its targets using `stdin`. However, the middleware contains always-running services that expose multiple APIs running in remote processes, so we need to bridge the process gap to forward inputs to target APIs. In our case, the fuzzing harness that we implemented [7] takes the target service, transaction ID and input data, and crafts valid binder transactions that it transmits to the middleware API. The forkserver that AFL usually employs to have multiple instances of its target running in parallel, in this case its harness, is deactivated for our purposes. All running harness instances are still backed by a single running target process so parallel executions are not improving performance but they increase complexity in terms of isolating executions of generated inputs.

Input Generation AFL typically outputs byte buffers that need to be structured according to the input format of the subject under test. To that end, we translate the argument type list of each targeted API into the harness' input encoding, initialize the argument types with default values, and use them as seeds for AFL. For instance, we extract the parameter types from the `VoiceInteractionManagerService` API `getKeyphraseSoundModel(int, java.lang.String)` and use the corresponding type's default values to generate a zero integer and an empty string as a seed input. Figure 4.4 depicts the initial seed and an excerpt of the inputs generated from it for the `getKeyphraseSoundModel` API. We abstain from leveraging AFL for fuzzing APIs

[7] We extended Android's *service* tool accordingly.

```
Seed:
i64 0 s16 ""

AFL generated inputs:
...
i64 150 s16
i64 0 s16 f6
i64 0 s16 tgEcs
i64 0 s16 08
i64 0 s16 zk8vF
i64 0 s16 5,n
i64 0 s16 6BQm
i64 0 s16 WxXU
i64 0 s16 OIlk
i64 0 s16 IYpuM
i64 0 s16 9pAq0
i64 0 s16 Cihm
i64 0 s16 3uuI
i64 0 s16 Dkm
i64 0 s16 tqb9Rq
i64 0 s16 ??7454
i64 0 s16 LLpH
i64 0 s16 FBn5
i64 0 s16 xAbJ6pIO
i64 0 s16 gy6qez
i64 -1061126006 s16
i64 -2122252032 s16 ja
i64 -2122252032 s16 MSTv
i64 -530563011 s16
```

```
i64 -2122252032 s16 1,3
i64 -2122252032 s16 2oU
i64 -2122252032 s16 QFlC
i64 -1061126005 s16
i64 50463252 s16
i64 -2122252032 s16 hO3W
i64 -2122252032 s16 x7W,
i64 -2122252032 s16 D.j
i64 -2122252032 s16 VsH
i64 -2122252032 s16 oPA
i64 -2122252032 s16 we
i64 -2122252032 s16 L7p
i64 -2122252032 s16 r6x’TI
i64 -2122252032 s16 ,Qn3
i64 201852916 s16
i64 -2122252032 s16 #Oqb
i64 -2122252032 s16 K9u
i64 -2122252032 s16 cOyHFF
i64 -2122252032 s16 ’U
i64 -2122252032 s16 p3yB
i64 -2122252032 s16 lit2G0’
i64 -2122252032 s16 5LUzI
i64 -2122252032 s16 uWjN
i64 -2122252032 s16 wc
i64 -2122252032 s16 sa
i64 100926444 s16
i64 -2122252032 s16 lSjs
i64 -2122252032 s16 nJTdb!
...
```

Figure 4.4: Inputs generated by AFL from an empty seed.

with complex types for now, as this requires careful object modeling in order to reach the target API and to not extensively fuzz the un-/marshaling logic.

Coverage Channel By default, AFL compiles a binary using a customized LLVM to insert feedback instructions into the code such that control-flow information of reachable code is written to a shared feedback buffer of fixed size. This raises two concerns: First, we need to share a memory buffer between the target process and AFL. Second, AFL cannot compile the target, i.e., the Systemserver, so we need another way to write feedback to the shared buffer.

We solve the shared buffer problem by introducing an on-device component called *Uplink*. Once executed, *Uplink* uses Android's shared memory system (*ashmem*) to create AFL's shared buffer. *Uplink* then launches a publicly available service for sharing the corresponding file descriptor so that the shared memory region can be accessed from its clients' process spaces as well. The first client is the Java-based library that ARTist injects into the Systemserver to ensure we obtain a file descriptor for writing coverage feedback. The second client is AFL, which reads coverage feedback from the shared buffer.

Concerning the second problem, we note that LLVM-based instrumentation is not always feasible (e.g., code unavailable, complicated dependencies) and therefore AFL supports a mode of operation where it skips the LLVM-based instrumentation process and just relies on an external tool to write feedback to the shared buffer. Similarly to our integration of Chizpurfle (see Section 4.5.6.1), we can use the coverage monitoring ARTist module to push the collected coverage information back to the fuzzer, this time by writing it to the shared buffer we obtained from *Uplink*.

4.5.7 Device Setup

Combining all aforementioned tools and modules, we obtain the device setup depicted in Figure 4.5. At boot time, ARTist executes our modules that instrument the Systemserver for crash detection and coverage feedback ①. After Android booted successfully, the chosen input generator is started with the current API target ② and generates inputs ③ that are transparently sent to the Systemserver via binder IPC ④. On the server side, the API is executed and the code injected by our coverage module notifies the injected Java library about each basic block that is visited ⑤. Coverage and crash information is persisted ⑥ and coverage feedback is provided back to the input generator if needed ⑦. The generation of new inputs is guided by the feedback received from the coverage module, hence executions discovering new basic blocks directly influence the generation of new inputs. Off-device the responsible worker periodically checks the health of all involved components (e.g., Systemserver, input generator, monitors), collects *logcat* messages, pulls system tombstones (Android crash dumps), and persists the collected artifacts ⑧.

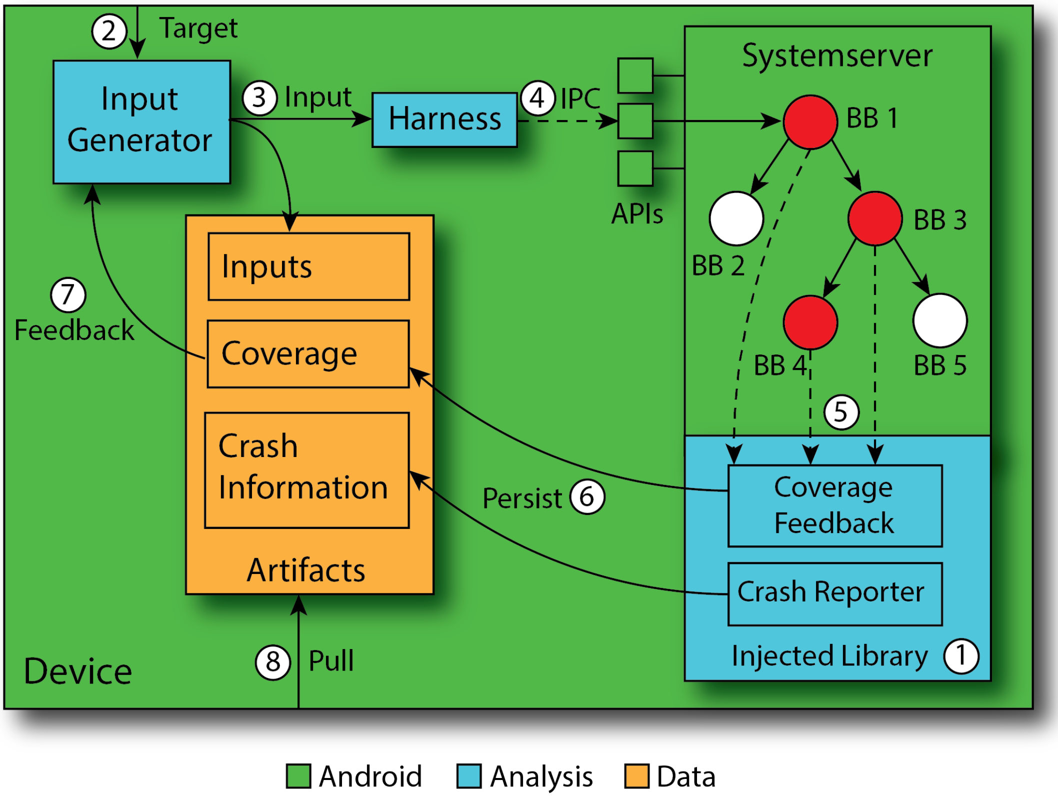


Figure 4.5: The full on-device setup from generating inputs to persisting and feeding back artifacts.

4.6 Case Studies

In this section, we study how to use our platform to approach dynamic analysis tasks from related work in a more principled way. The two case studies we conduct show how our platform can be utilized to explore different design decisions and strategies, which we exemplify by comparing the impact of input generation strategies, such as the choice of fuzzers and whether to utilize coverage feedback. Our results indicate that there are still lots of open questions in order to find *optimal* components for the different dynamic analyses.

4.6.1 Vulnerability & Bug Discovery

Vulnerability and bug hunting is often perceived as a *dark art* driven by intuition and gut feeling. As evident from Table 4.1, there has been a multitude of approaches for hunting bugs in the Android domain, in particular focusing on the Systemserver and other middleware components. While they succeeded in uncovering different kinds of flaws, they rely on customized toolchains that are mostly incompatible and incomparable. This raises questions such as whether the implemented strategies are the most beneficial ones, or where we can still improve in those areas. While these questions currently cannot be answered for the Android domain, there has been a lot of work in related fields recently that focuses on properly comparing and evaluating dynamic analysis approaches and their central components (e.g., input generators) to create benchmarks and evaluation criteria [96, 125, 47]. For dynamically analyzing the middleware, however, we are lacking such a principled approach that places related work and their results on firm ground. We conduct a case study on vulnerability and bug hunting on top of our platform to answer questions such as whether coverage-guided approaches outperform black-box approaches and how the required instrumentation impacts performance.

4.6.1.1 Design

We replicate the vulnerability detection scenario of related work from Table 4.1 by designing our case study to search for crashes in the middleware that lead to, e.g., interrupting users by rebooting the device (potentially in a loop). To be able to replace and compare components easily, we fully utilize all existing building blocks: The three integrated fuzzers (Section 4.5.6) are generating inputs, the monitors (Section 4.5.5.2) feed back coverage and crash information, and Troop (Section 4.5.2) parallelizes the testing across multiple emulators. As soon as a monitor detects a crash and all information is pulled form the device, we use the auto-verification support to replay the collected inputs on a clean vanilla emulator. The input replaying component together with the list of inputs thereby form a proof-of-concept exploit. This platform-driven approach for the first time allows us to conduct a comparative evaluation to assess the impact of different design decisions.

4.6.1.2 Comparative Evaluation

The platform's modular design enables us to evaluate different components under the same conditions while leaving all other components fixed between tests. Since most related work employ fuzzers to drive their concrete executions, we create seven different setups based on our integrated fuzzers to study their impact in terms of performance and coverage.

Setup. All experiments are executed with 30 parallel workers analyzing each target API for 5 minutes. The emulators run a custom AOSP 7.1 image that includes additional tooling required for the evaluation. We compute the target APIs for the experiments by first enumerating all methods from the 89 Java-backed system services. Out of the 2,090 interface-method-pairs, we were able to map 2,005 APIs to concrete implementations and generated coverage information for 1,867 (89%) of them, where we only consider those 1,637 (78%) with a non-empty list of parameters. Subsequently we divide our experiments into two groups based on the APIs they are targeting. The first group contains all 433 target APIs that are currently supported by AFL in black box (BB) and evolutionary (Evo) mode, as well as RandFuzz because all their parameters have primitive or string types and we have binder transaction IDs available, hence we call this group the *primitive APIs*. The second group, which contains all 1,637 APIs, is called the *complex APIs*. It can only be fully tested by input generators that support the creation of complex objects, which is in our case Chizpurfle in black box and evolutionary mode. Running all five fuzzer variants on the primitive APIs as well as both Chizpurfle variants on the complex APIs results in seven setups in total.

Performance. First, we measure the raw performance of all fuzzers by computing the time between executions for generated inputs to evaluate their trade-off between more complex generation heuristics and sheer throughput of inputs. Second, we use the fuzzers to measure the overhead introduced by our instrumentation modules for coverage tracking.
Table 4.5 depicts the results of our measurements. The first observation is AFL's low number of executions per second (between 30-150 as computed from the execution times) compared to the over 500 suggested by the AFL manual [140], which is expected because of the additional overhead added for the IPC round-trip and the execution of harness code. The second insight is that, as expected, RandFuzz is the fastest fuzzer because generating purely random values is magnitudes faster than the complex input generation logic of AFL.
Third, as expected, the black box variant of Chizpurfle is faster than the evolutionary variant. While we would have expected similar results for AFL, we found large outliers in the measurements for the black box variant of AFL (with and without ARTist activated) that can explain the unexpectedly high execution time.
The fourth information conveyed by the table is the acceptable overhead imposed by our instrumentation for the fuzzers. Table 4.5 indicates that our coverage tracking instrumentation is magnitudes faster than the Chizpurfle Frida module's reported 13-fold overhead. However, we take this comparison with a grain of salt because, first, we could

Table 4.5: Performance measurements for time between executions in milliseconds, averaged over the different APIs using mean and median respectively.

Input Generator	Per-API Means		Per-API Medians	
	Mean	Median	Mean	Median
Complex APIs				
Chizpurfle Evo	387.97	22.95	374.24	20.88
Chizpurfle BB	99.15	8.88	101.61	7.68
Chizpurfle BB (no ARTist)	75.92	9.05	60.10	7.79
Primitive APIs				
Chizpurfle Evo	170.47	23.30	166.99	20.98
Chizpurfle BB	15.57	8.74	13.98	7.52
Chizpurfle BB (no ARTist)	15.69	8.84	13.96	7.62
AFL Evo	9.64	7.81	8.33	7.44
AFL BB	32.36	12.10	7.54	7.05
AFL BB (no ARTist)	21.51	8.35	7.21	6.63
RandFuzz	0.42	0.08	0.34	0.07
RandFuzz (no ARTist)	0.36	0.09	0.26	0.08

not evaluate Frida ourselves on our platform, and second, it was not explicitly stated in the papers [91, 38] whether the overhead was an end-to-end measurement.

Coverage. Table 4.6 depicts the results of our evaluation that compares different input generators running in the same environment with identical time budget.

In other domains, coverage-guided greybox fuzzers are commonly known to generate higher coverage than their black box equivalents so we want to evaluate whether this holds in our domain as well. The surprising result of our experiments is that this does *not* seem to hold in our evaluated scenario. As evident from Table 4.6, both setups that utilize evolutionary fuzzers, AFL (Evo) and Chizpurfle (Evo), fail to provide substantial benefits over their black box equivalents and the pure random RandFuzz in terms of achieved coverage during the testing. This finding again reinforces that thorough analysis is needed to make principled decisions about the design of dynamic analyses for the Android system.

4.6.1.3 Discovered Bugs & Crashes

We also evaluated whether the fuzzers were able to uncover bugs similar to what related work targeted. Our results indicate that, in contrast to related work that created specialized toolchains to aim at particular kinds of flaws, integrating available fuzzers with our common platform already finds instances of the same kinds without the need for specialization. In the following, we will discuss the different categories of flaws that were detected by our system and relate them to those from related work.

Table 4.6: Measured coverage for different input generators.

Input Generator	coverage mean	coverage median
	Complex APIs	
Chizpurfle Evo	28.31%	21.66%
Chizpurfle BB	28.53%	21.88%
	Primitive APIs	
Chizpurfle Evo	36.63%	36.36%
Chizpurfle BB	35.65%	33.33%
AFL Evo	34.61%	33.33%
AFL BB	34.50%	29.73%
RandFuzz	33.08%	29.79%

Exception-based Crashes (`UncaughtException`). Similar to ExHunter, our system was able to find multiple `UncaughtException` flaws, which crash the Systemserver and force a soft-reboot. However, our system found these flaws without requiring any kind of specialization and — because it is built on top of Troop — goes the extra mile of confirming and reproducing it automatically. The most surprising results of our evaluation, however, are three vulnerable `Statusbar` and `PackageManagerService` APIs that lead to crashes of the SystemUI app. Given the focus of our analysis, we mainly expected exceptions and native crashes in system services. However, because many problems are not known beforehand, we also created commands for our result analyzer that heuristically checks the generated test artifacts for other interesting behavior by, e.g., collecting crash notifications from logcat, which led to the detection of these three SystemUI crashes. This discovery exemplifies how a more modular approach can lead to new insights, even beyond what was originally targeted.

Systemserver Freeze (ASV) & Bootloop. While ASVHunter was particularly designed to locate *Android Stroke Vulnerabilities (ASVs)*, our prototype was able to uncover an instance of this flaw category in a generic way for the `setOverscan` API in the `WindowManagerService` (see Appendix B.1). Furthermore, the flaw we discovered not only reboots the Systemserver once but permanently soft-bricks the whole device, which no existing work claims to have found.

Resource Exhaustion. We detected the vulnerable API `getSharedAccountsAsUser` in the `AccountManagerService` that, when hammered with fake user IDs for a period of less than a minute, opens a large number of *SQLite* database connections until the file descriptor limit of 1024 is reached and the system crashes. Judging from the limited information that is available about prior work, none of the proposed approaches seems to have found an instance of this flaw category.

Native Crashes. During our evaluation, we surfaced a set of Java APIs that immediately forward all inputs they receive to their corresponding native counterparts via JNI.

We then often saw a native Systemserver crash generated by a tool called *CheckJNI* [66] that prevents, e.g., de-referencing null pointers provided as inputs where complex Java objects are expected. While it is straightforward to misuse these as denial-of-service attacks for crashing the Systemserver and soft-rebooting the device, we believe that these natively implemented methods expose a completely uncharted attack surface that might lead to more serious vulnerabilities (e.g., memory corruption flaws), which we discuss in more detail in Section 4.6.5.2.

Exploitability. The 11 detected flaws require permissions that cannot be obtained by regular apps, thus our attacker model not only considers malicious third-party apps but in addition misused vendor apps. This is motivated by prior research that has shown that, in the past, vendors decreased platform security in their ROMs by, e.g., accidentally exposing a system shell [105], lowering protection levels of system permissions to dangerous or normal [7], or introducing confused deputy components that expose their privileges [134]. Additionally, Xing et al. [136] reported a so-called permission upgrade attack where apps could elevate custom permissions to system permissions due to name clashes. These scenarios illustrate that this type of flaw can still be triggered by regular apps from the Google Play store in situations where the AOSP code is modified by vendors in an insecure way.

4.6.2 Responsible Disclosure

We responsibly disclosed the issues uncovered during our vulnerability case study to Google. While earlier works received CVEs for their findings, according to the wayback machine [79], Google in the meanwhile changed their guidelines on what is considered for their vulnerability disclosure program. The new guidelines explicitly remove application-triggered DoS attacks on the middleware from the scope of the bug bounty program without further notice. In particular, related work, if published today, would also not receive CVEs for their findings according to Google's new guidelines. While the flaws will not be fixed as a part of the security bulletin, the reported issues are still passed on internally to the corresponding Android teams. However, given the crucial role of the Systemserver and the middleware in general, we still see the discovery of these kinds of bugs as an important contribution to the system's overall stability and security.

4.6.2.1 Towards Principled Vulnerability & Bug Discovery

By exploiting the platform's modularity, we were able to compare different input generation strategies in terms of their performance, findings and generated coverage already. Our work provides valuable insights for future iterations of bug hunting dynamic analyses, such as to not focus on a single flaw category alone or to re-use existing building blocks. Furthermore, the modular integration provides additional functionality, such as automated post-analysis of results (e.g., auto-verification or identification of novel bug classes) or execution under different privilege levels for free. There are many more aspects to be considered in the future, such as whether completely different input generators, such as concolic executors, can provide substantial benefits over fuzzers. However, our case study successfully highlights how a more structured approach allows

Table 4.7: Comparing the permissions mapped by different fuzzers.

	Axplorer [28]		Arcade [5]	
	New	Confirmed	New	Confirmed
	Complex APIs			
Chizpurfle Evo	132	515 (42.08%)	287	360 (41.67%)
Chizpurfle BB	144	557 (45.51%)	307	394 (45.60%)
	Primitive APIs			
Chizpurfle Evo	52	206 (64.78%)	110	148 (65.78%)
Chizpurfle BB	56	225 (70.75%)	116	165 (73.33%)
AFL Evo	50	229 (72.01%)	113	166 (73.78%)
AFL BB	50	230 (72.32%)	112	168 (74.67%)
RandFuzz	56	230 (72.33%)	118	168 (74.67%)

to, for the first time in the field of Android middleware dynamic analysis, compare strategies from related work in a common testbed.

4.6.3 Permission Mapping

Our second case study focuses on dynamically creating a mapping of Systemserver APIs to their sets of enforced permissions, which Stowaway implemented by instrumenting the Systemserver to log permission checks and using the Randoop [25] test generator to create inputs. In this case study, we investigate alternative strategies for dynamic permission mapping by using existing fuzzers and comparing the results to state-of-the-art permission maps.

4.6.3.1 Design

The design is driven by the idea of re-instantiating the approach of Stowaway to exploit the capabilities of our platform. We explore alternative input generation strategies by using the built-in fuzzers to test single APIs. Instead of hooking the permission enforcement points, which were manually picked and modified in the original work, we automatically parse the exceptions thrown back to the fuzzer for enforced permissions. The number of observed permission checks is further increased by running the fuzzer from within the context of a zero-permission app.

4.6.3.2 Comparative Evaluation

Since Stowaway is not available for newer Android versions, we have to compare our results to the Axplorer [28] and Arcade [5] state-of-the-art static analysis permission mappings. As an example of how a common platform can help evaluate different design decisions, we ran the permission mapping experiment in all available fuzzer configurations. Table 4.7 depicts the results of our evaluation. While our prototype is already able to uncover new permission mappings missed by related work, it cannot

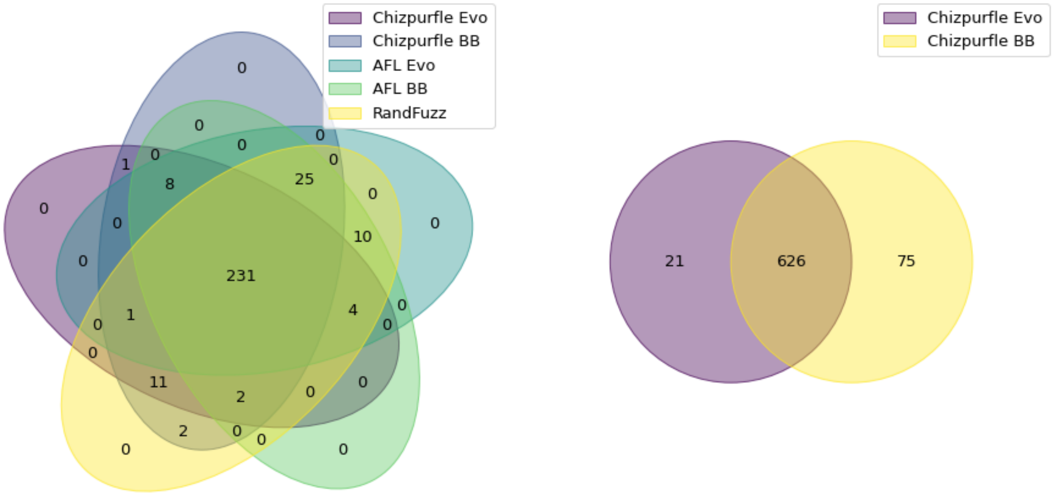


Figure 4.6: Overlap of mapped permissions for primitive (left) and complex (right) APIs.

confirm all previous results. Figure 4.6 allows to further explore how the results relate to each other.

Surprisingly, the fuzzers produce similar results despite their different input generation strategies (i.e., coverage-driven versus black box) and levels of sophistication. In our evaluation, coverage feedback as employed by Chizpurfle (Evo) and AFL (Evo) did *not* lead to consistently superior results as evident from Table 4.7 and Figure 4.6, which might be a distinctive factor of this particular problem domain.

4.6.3.3 Towards Principled Permission Mapping

These findings are indicators that a principled approach to comparing and evaluating dynamic analyses down to their building blocks is required to challenge common beliefs and identify optimal strategies. For example, coverage feedback as employed by Chizpurfle (Evo) and AFL did *not* lead to consistently superior results, as evident from Table 4.7 and Figure 4.6, which might be a distinctive factor of this particular problem domain. We thereby show that further research is required to identify the *optimal* strategy for permission mapping by first understanding and then combining the underlying design decisions of different fuzzers that contribute to the vastly deviating results. Eventually, implementing such an optimal strategy as a core component into different input generators might be able to push the majority of results towards the center of the Venn diagram.

4.6.4 Outlook: Permission Mapping 2.0

This outlook refers to the concurrently developed paper *Bringing Balance to the Force: Dynamic Analysis of the Android Application Framework* [44] (under submission) that is **not** a part of this thesis. However, it follows a principled approach to permission mapping as suggested by our work and therefore acts as an *external* use case example.

While our case study in Section 4.6.3 outlines a naïve design for a dynamic analysis-based permission mapping, it suffers from a set of limitations, such as missing detection of non-permission based access control mechanisms (e.g., UID based) or second-level permission checks that are only observed when passing the first-level checks. The follow-up paper *Bringing Balance to the Force: Dynamic Analysis of the Android Application Framework* improves upon this by, e.g., using selective instrumentation of middleware code to rotate the calling UID of requests, which allows to test for combinations of permission and UID checks.

4.6.5 Further Use Cases

Besides the two use cases we implemented as case studies, there is a large set of possible dynamic analyses that can benefit from being instantiated on top of our platform, out of which we discuss a subset here.

4.6.5.1 Differential Analysis

Given the strong fragmentation of the Android ecosystem due to vendor patches and the resulting risk of introducing vulnerabilities in the process [105, 134, 7], differential analysis allows to argue about differences introduced to security-critical components like the Android middleware. Here, our common platform can be used to analyze multiple ROMs based on the same major Android release and check for differences and inconsistencies, such as removed permission checks or adaption of the core logic of APIs. This can also be used for patch verification, i.e., confirming that an introduced change actually prevents the vulnerability.

4.6.5.2 Targeting Systemserver Native Code

Below the Systemserver lies a set of native libraries that is utilized via JNI, which in contrast to app-facing native code (e.g., `libstagefright`) does not directly consume input from untrusted code and therefore has not been thoroughly studied. However, these libraries are an essential part of Android's application framework and might contain critical bugs and vulnerabilities. In our vulnerability detection case study, we already identified several APIs that are candidates for deeper analysis. Manual analysis of those candidates revealed that this approach indeed seems promising and we consider it highly interesting future work to further explore this. With a three-step approach, one would first need to automatically identify native libraries loaded by the Systemserver and reconstruct all connections between Java code and JNI-accessed native methods. Second, these libraries could be directly analyzed to generate a set of possible candidate defects. In the third step, one could utilize techniques, such as directed greybox fuzzing or concolic execution, to search for paths leading from publicly accessible Systemserver APIs down to the libraries via JNI. While static analysis might help to implement step one, our platform allows to properly implement steps two and three. Troop already provides many core components for this analysis, such as instance management, monitors, and auto-verification. Furthermore, integrating, e.g., a directed greybox fuzzer into the platform will allow to compare it against existing input generators to find the best fit for this particular task.

4.6.5.3 Confirming Static Analysis Results

Static analysis of the Systemserver is prone to false positives as a result of Android's asynchronous communication capabilities and complicated lifecycle. However, as implemented for the evaluation of the permission mapping use case in Section 4.6.3.2, dynamic analysis methods can be utilized to either confirm or refute preliminary results by anchoring them in real execution traces. Our platform provides the testbed to evaluate which combinations of input generators and target patches (e.g., disable permissions, enforce particular system state, directly jump to code locations) is beneficial for this kind of analysis.

4.6.5.4 Analyzing Other Android Components

For the purpose of this paper, our platform is specifically designed to support analyses that target Android's middleware and in particular the Systemserver. However, its infrastructure can also be utilized to test other components of the Android ecosystem. For example, we can re-use building blocks, such as instance management and monitors, to automate dynamic application testing, which has partially been done with the *monkey-troop* tool (see Section 3.7.1.1). Given that the platform evolved from *monkey-troop*, re-introducing the app analysis capabilities is a straightforward task for future work.

4.7 Future Extensions

We can exploit the platform's modularity to integrate more approaches from related work in the future to make them not only available to all existing analyses built on top but also bring more complex dynamic analyses within reach.

4.7.1 Transformational Fuzzing

Transformational fuzzing, as applied in T-Fuzz [109], proposes to uncover new execution paths by skipping program conditions like sanity checks or environmental checks using an instrumentation framework. Fuzzing such a program can efficiently generate a candidate set of program execution traces, which, however, is not guaranteed to trigger the same functionality in the original, non-transformed program. Therefore, symbolic execution is subsequently used to confirm whether there are inputs that generate the candidate traces in the unmodified program. In contrast to hybrid fuzzers like Driller [126] that repeatedly switch between fuzzing and symbolic execution, the idea behind transformational fuzzing is to reduce the usage of costly symbolic execution to candidate verification only. Implementing this idea within our framework could tremendously boost the achieved coverage for all currently integrated input generators because it circumvents many of the checks that are hard to avoid by regular fuzzers, such as privilege checks or assumptions on the runtime environment. Using ARTist and Frida for instrumenting Java and native code respectively, this approach could be implemented for different parts of the middleware. In general, this approach is well suited for dynamic analyses that require a *deep* analysis of the code.

4.7.2 Test Case Minification

Test case minification (also test case reduction, delta debugging) is a known technique [141] in the software testing community to produce minimal and hence more efficient versions of test cases without reducing their utility, e.g., coverage or triggered faults. Similarly to how we implemented the automated verification of candidate inputs, our platform could incorporate standard techniques from this field to ensure the reproduction test cases are as small as possible with respect to size and runtime. Having a shared platform thereby allows to immediately apply this to all the different analyses that could be implemented within our framework.

4.7.3 Alternative Input Generators

While we first focused on the integration of fuzzers into our platform, alternative input generators could be implemented as well. In particular, it would be interesting to combine our platform with frameworks such as angr [125] or DeepState [31] that already implement multiple input generation strategies, i.e., fuzzing *and* concolic execution, in a unified framework. This would allow dynamic analyses to pick their strategy according to the needs of their concrete use case.

4.7.3.1 Continuous Analysis

While OSS-Fuzz [69] automates the continuous testing of popular projects that can be built with *Libfuzzer* [24] support, we envision a similar paradigm for our platform that allows us to plug-in arbitrary fuzzers (and even other dynamic testing approaches) and have them automatically executed on the middleware and other Android components. Once set up, this approach could be applied to test, e.g., patches or new versions of critical components without manual intervention to further improve the platform robustness and security.

4.8 Conclusion

In this chapter, we systematically analyzed common requirements and challenges for dynamically analyzing Android's crucial middleware component with a special focus on the Systemserver as the host of many integral services of the Android OS. Based on our observations, we implement a common platform as a first step towards more generic solutions that allow to integrate, compare, and enhance existing but also novel approaches in this area. We conducted two case studies that instantiate common analyses from related work where we utilized our platform's modularity to evaluate and discuss different implementation strategies and design decisions. We responsibly disclosed our findings and are going to open source the whole platform including both case studies, as well as all underlying modules and supporting tools in a way that allows for reproduction and independent evaluation. We hope that, going forward, our platform allows for a more principled way of approaching new interesting research ideas in the area of dynamically analyzing Android's middleware.

5
Conclusion

In this dissertation, we presented two projects that advance the state-of-the-art of dynamic analysis on Android and provide an important step towards a more principled approach for future research in this area.
Dynamic analysis of the application and middleware layer has been an active area of research on Android. Existing approaches have focused on detecting a multitude of flaws in applications and system components, such as over-permissioning, information leakage, or code execution vulnerabilities. While this prior related work succeeded in pointing out new interesting behaviors, they lack a common foundation to properly compare the results and evaluate the corresponding artifacts, which in addition are often not publicly available. This resulted in the unsatisfactory situation that building upon existing work is often not possible and therefore already solved problems, such as managing target devices to scale up analyses, or implementing certain building blocks for the Android ecosystem, have to be solved again from scratch. The static analysis literature for Android, in contrast, provides a different picture that emphasizes re-usability and comparability by relying on a small set of common base frameworks. Consequently, we see a lot of literature utilizing techniques discovered and published by prior work in order to advance the current state-of-the-art in that direction. Learning from this major asymmetry, the focus and contribution of this dissertation is to provide an important step towards bridging the gap between dynamic and static analysis by providing a common platform to instantiate, evaluate, reproduce, and compare dynamic analyses that target the Android ecosystem. More precisely, our work spreads across two major publications that form the integral building blocks of our approach:

ARTist. With ARTist, we identified and occupied a gap in the design space of instrumentation frameworks for Android. In contrast to existing frameworks, ARTist supports application layer-only as well as system-centric deployment, operates on instruction granularity, and is independent of the hardware architecture at the same time. We provide a module SDK that facilitates the creation of instrumentation modules that can change the behavior of target apps or services arbitrarily. Given the deprecation of the DVM and subsequently those works from the community that depend on it, ARTist provides a novel approach to re-instantiate these on Android versions that build on the Android Runtime. Furthermore, introducing a compiler-based instrumentation framework, for the first time on Android, provides the opportunity to learn from and integrate advances from the compiler-based security community that independently evolved in related fields outside of Android. Using this important building block, we can efficiently implement surgical as well as large scale changes to Java-based components on Android (i.e., apps and system services) to support dynamic analyses.

Troop. Based on a thorough requirements analysis of related work in the area of Android middleware dynamic analysis, we designed Troop to provide the community with a unified platform to instantiate dynamic analyses without having to re-invent the wheel over and over again. We implemented and integrated often used building blocks from prior work and provide the platform to compare and evaluate them in order to find the best strategies for certain dynamic analysis tasks. For example, we integrated own and existing fuzzers, both from academia and industry, to evaluate the impact of their particular design decisions, such as coverage feedback-driven input generation. We envision our system to provide a modular starting ground for dynamic analyses that allows researchers to focus on the analysis task at hand and easily share the results and artifacts in an effort to advance open and reproducible research.

Future Research Directions. We learned that, while bug and vulnerability discovery are popular topics among related work that targets Android's application and middleware layer, the field still lacks behind the state-of-the-art of related research areas, such as binary analysis. We see it as an interesting research question to identify whether this gap is originating from the particularities of the underlying platform (i.e., Android versus commodity Linux) or might stem from, e.g., disjoint research communities following similar goals independently, and subsequently how to bridge this gap to fully exploit the results of neighboring areas and bring their research to the Android platform.

Studying static analysis, we also see a body of research that focuses on analyses that do not necessarily involve the detection of typical bugs but rather focus on, e.g., auditing and making policies visible, such as permission mapping, or uncovering possible access control re-delegation, such as confused deputy analysis. Exploring whether analyses beyond bug hunting can benefit from pure dynamic analysis or in combination with static analysis is an interesting line of research that is yet uncharted.

Another problem we often see in the literature is the lack of a cross-layer approach when it comes to mixed Java and native code targets. Besides a few exceptions, most analyses either stop at the border and declare analysis of the other parts as out of scope, thereby creating specialized tools for either Java *or* native code analysis. Our platform can, to a certain extent, already support the dynamic analysis of mixed environments, such as Systemserver-based Java services, JNI bridges, and fully native services, by providing different building blocks for, e.g., instrumentation (ARTist for Java, Frida for native), or input generation (Chizpurfle for Java, RandFuzz for native). However, in the future we would like to further resolve this border by investigating joint analyses that can handle both worlds in a shared environment.

Furthermore, our evaluation results challenged the common belief that feedback-driven input generators are superior to black box alternatives in the context of Android middleware analysis. Building on this insight and utilizing Troop as an evaluation platform, we see a whole line of research that investigates and establishes *optimal* strategies in this particular scenario and investigates which of those insights from other fields carry over to the Android domain and why. This research will be particularly useful to steer this field towards a more principled approach where analysis strategies are based on empirical research instead of anecdotal evidence.

A
Tools & Software

For ARTist (see Chapter 3) and Troop (see Chapter 4) we created a whole ecosystem that eases dynamic analysis on Android. Figure A.1 gives an impression on how they fit into the overall picture. This Appendix additionally serves as a full list for the tools, software, and other repositories created for and with these two research projects.

A.1 ARTist

Name	Description	Type	Languages	Forked
		Framework		
ARTist	Instrumentation	Framework	C++	new
art	ARTist Dependency	Runtime	C++	AOSP/art[77]
ArtistGui	App-level Deployment	App	Java	new
module-sdk-gen	Module SDK Generator	Script	Python	new
monkey-troop	ARTist Evaluation	Framework	Python	new
codelib-gen	Codelib Helper	Script	Python	new
		Modules		
template-module	Template Module	Module	C++	new
template-codelib	Template Codelib	Library	Java	new
trace-module	Method Tracing Module	Module	C++	new
trace-codelib	Method Tracing Codelib	Library	Java	new
stetho-module	Stetho Injection Module	Module	C++	new
stetho-codelib	Method Tracing Codelib	Library	Java	new
logtimization-module	Simple Test Module	Module	C++	new

A.2 Troop

Name	Description	Type	Languages	Forked
Host-level				
troop	Orchestrator	Framework	Python	monkey-troop[104]
db-backend	Task Backend	Server	Python	new
avd-tool	Emulator Management	Script	Python	new
transaction-id-mapper	API to Binder IDs	Script	Python	new
On-Device				
Chizpurfle	Customized Fuzzer	Java Executable	Java	chizpurfle [129]
android-afl	AFL for Android	Binary	C	andorid-afl[3]
rand-fuzz	random fuzzer	Binary	C++	new
libuplink	Ashmem FD Sharing	Service & Lib	C++	new
Servitor	Test Harness	Binary	C++	AOSP/Service[18]
FuzzWatch	Ashmem Visualization	Binary	C++	new
Fuzzing Codelib	Injected Code Host	Library	Java	CodeLib[36]
Replay App	Verification	App	Java	new
AOSP Build Support				
fuzzing-scripts	Automation	Scripts	Bash	new
build	Extension	Makefile	Text	AOSP/build[16]
build_fuzzing	Extension	Makefile	Text	new

A.3 Ecosystem

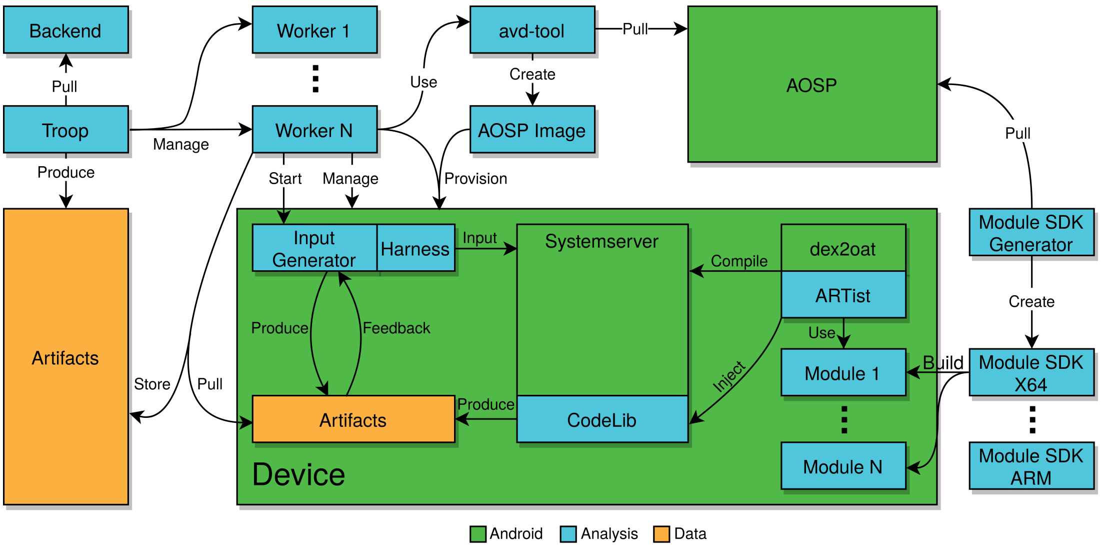


Figure A.1: Full ecosystem of tools for dynamically analyzing the Systemserver.

B Responsible Disclosure

In order to responsibly disclose the issues we found during the Troop evaluations (see Section 4.6.1), we created reports for each flaw that we submitted via Google's official issue tracker [74]. We exemplarily display one of those reports in the following section.

B.1 Title: Repeatedly calling `WindowManagerService.setOverscan` soft-bricks the device (bootloop, broken SystemUI state)

TL;DR:
The API `com.android.server.wm.WindowManagerService.setOverscan (int,int,int,int,int)` can be abused by a privileged application/system component to send the device into an unstable state where the UI is unusable and there is nothing left but restarting the device. However, a restart triggers an infinite bootloop where the SystemUI cannot properly be instantiated (unusable for the user) and shortly afterwards `Watchdog` kills the systemserver and everything repeats. We could only reclaim the device after deleting the userdata image via `fastboot` (~ reset). The described vulnerability was automatically detected by our systemserver dynamic analysis system and manually confirmed afterwards on a Google Pixel running Android 7.1 (rooted, otherwise stock ROM).

Fingerprint:
google/sailfish/sailfish:7.1.1/NOF27D/3757586:user/release-keys

Description:
The `setOverscan` API can be trivially abused to deny the user access to their device by setting an overscan value that effectively moves all content outside the visible area.

Additionally, if we keep on sending these requests in a loop, the window manager service lock is taken for too long so `Watchdog` cannot execute its checker on the foreground thread, hence killing the systemserver. Now the system reboots but the SystemUI is broken, i.e. only the navigation bar is displayed but unusable. In particular, the `sensorservice` that died earlier does not restart but the service manager needs to wait for it:

```
I ServiceManager: Waiting for service sensorservice...
E slim_daemon: [NDK] bindNDKSensors: Sensor server is unavailable.
I ServiceManager: Waiting for service sensorservice...
E slim_daemon: [NDK] bindNDKSensors: Sensor server is unavailable.
I ServiceManager: Waiting for service sensorservice...
E slim_daemon: [NDK] bindNDKSensors: Sensor server is unavailable.
E slim_daemon: [NDK] bindNDKSensors: Sensor server is unavailable.
I ServiceManager: Waiting for service sensorservice...
E slim_daemon: [NDK] bindNDKSensors: Sensor server is unavailable.
I ServiceManager: Waiting for service sensorservice...
E slim_daemon: [NDK] bindNDKSensors: Sensor server is unavailable.
I ServiceManager: Waiting for service sensorservice...
E slim_daemon: [NDK] bindNDKSensors: Sensor server is unavailable.
I ServiceManager: Waiting for service sensorservice...
E slim_daemon: [NDK] bindNDKSensors: Sensor server is unavailable.
I ServiceManager: Waiting for service sensorservice...
E slim_daemon: [NDK] bindNDKSensors: Sensor server is unavailable.
E slim_daemon: [NDK] bindNDKSensors: Sensor server is unavailable.
```

From now on, the only thing the user can do is press the power button until the device restarts. However, now the device is kept in a boot loop because `Watchdog` detects unresponsiveness:

```
W Watchdog: *** WATCHDOG KILLING SYSTEM PROCESS: Blocked in monitor com.android.server.wm.WindowManagerService on
foreground thread  (android.fg), Blocked in handler on ui thread (android.ui), Blocked in handler on display
thread (android.display)
W Watchdog: foreground thread stack trace:
W Watchdog:   at com.android.server.wm.WindowManagerService.monitor(WindowManagerService.java:11041)
W Watchdog:   at com.android.server.Watchdog$HandlerChecker.run(Watchdog.java:179)
W Watchdog:   at android.os.Handler.handleCallback(Handler.java:751)
W Watchdog:   at android.os.Handler.dispatchMessage(Handler.java:95)
W Watchdog:   at android.os.Looper.loop(Looper.java:154)
W Watchdog:   at android.os.HandlerThread.run(HandlerThread.java:61)
Watchdog:     at com.android.server.ServiceThread.run(ServiceThread.java:46)
Watchdog: ui thread stack trace:
W Watchdog:   at com.android.server.wm.WindowManagerService$LocalService.waitForAllWindowsDrawn(WindowManagerService
.java:11663)
W Watchdog:   at com.android.server.policy.PhoneWindowManager.finishKeyguardDrawn(PhoneWindowManager.java:6385)
W Watchdog:   at com.android.server.policy.PhoneWindowManager.-wrap8(PhoneWindowManager.java)
W Watchdog:   at com.android.server.policy.PhoneWindowManager$PolicyHandler.handleMessage(PhoneWindowManager.java:773)
W Watchdog:   at android.os.Handler.dispatchMessage(Handler.java:102)
W Watchdog:   at android.os.Looper.loop(Looper.java:154)
W Watchdog:   at android.os.HandlerThread.run(HandlerThread.java:61)
W Watchdog:   at com.android.server.ServiceThread.run(ServiceThread.java:46)
W Watchdog: display thread stack trace:
W Watchdog:   at com.android.server.wm.WindowManagerService$H.handleMessage(WindowManagerService.java:8850)
W Watchdog:   at android.os.Handler.dispatchMessage(Handler.java:102)
W Watchdog:   at android.os.Looper.loop(Looper.java:154)
W Watchdog:   at android.os.HandlerThread.run(HandlerThread.java:61)
W Watchdog:   at com.android.server.ServiceThread.run(ServiceThread.java:46)
W Watchdog: *** GOODBYE!
```

Full logcat dumps are attached.

Proof-of-Concept:
Our framework uses a version of the `service` tool that forwards requests to the Systemserver via Binder IPC. In order to confirm a potential issue, we simply replay the

generated inputs using a script that repeatedly invokes `service` with these inputs. The script and the inputs that trigger this behavior are attached. For reproduction we currently use the adb shell b/c an app cannot obtain the required permission for changing the overscan, hence our attacker model is a misbehaving or buggy system app/component (potentially from a third-party vendor). While this attacker model is strictly weaker than considering regular apps, e.g., from the play store, research has shown [1,2,3] that misbehaving vendor components including confused deputies existed in the past and that in some circumstances it is possible for regular apps to obtain system privileges. Note: We did not (yet) investigate whether there are eligible apps in AOSP or other ROMs that could be misused for the attack vector presented here.

Contact:
Please feel free to contact us for any questions that might come up for this issue, we are happy to provide further information, receive feedback and engage in discussions.

Attachments:

- script to reproduce findings via `adb shell`
- input list that triggers the flaw
- logcat dumps: During attack & after first reboot

[1] http://www.sh4ka.fr/Android_OEM_applications_insecurity_and_backdoors_without_permission.pdf
[2] Paper: Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis (https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_aafer.pdf)
[3] Paper: The Impact of Vendor Customizations on Android Security (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.687.360&rep=rep1&type=pdf)

Bibliography

Author's Papers for this Thesis

[P1] Backes, M., Bugiel, S., Schranz, O., Styp-Rekowsky, P. von, and Weisgerber, S. ARTist: The Android Runtime Instrumentation and Security Toolkit. In: *IEEE EuroS&P'17*.

[P2] Schranz, O., Weisgerber, S., Derr, E., Backes, M., and Bugiel, S. Towards a Principled Approach for Dynamic Analysis of Android's Middleware. Under Submission.

Other Papers of the Author

[S1] Backes, M., Bugiel, S., Hammer, C., Schranz, O., and Styp-Rekowsky, P. von. Boxify: full-fledged app sandboxing for stock android. In: *USENIX SEC'15*.

[S2] Backes, M., Bugiel, S., Schranz, O., and Styp-Rekowsky, P. von. Boxify: bringing full-fledged app sandboxing to stock android. *USENIX ; login* 41, 2 (2016).

[S3] Huang, J., Schranz, O., Bugiel, S., and Backes, M. The art of app compartmentalization: compiler-based library privilege separation on stock android. In: *ACM CCS'17*.

Author's Tech Reports

[T1] Schranz, Oliver. *ARTist - A Novel Instrumentation Framework for Reversing and Analyzing Android Apps and the Middleware*. URL: https://i.blackhat.com/us-18/Thu-August-9/us-18-Schranz-ARTist-A-Novel-Instrumentation-Framework-for-Reversing-and-Analyzing-Android-Apps-and-the-Middleware-wp.pdf (Accessed: July 3, 2020).

Other references

[3] (GitHub user), ele7enxxh. *Fuzzing Android program with american fuzzy lop (AFL).* URL: `https://github.com/ele7enxxh/android-afl` (Accessed: July 3, 2020).

[4] Aafer, Y., Huang, J., Sun, Y., Zhang, X., Li, N., and Tian, C. Acedroid: normalizing diverse android access control checks for inconsistency detection. In: *NDSS'18*.

[5] Aafer, Y., Tao, G., Huang, J., Zhang, X., and Li, N. Precise Android API Protection Mapping Derivation and Reasoning. In: *ACM CCS'18*.

[6] Aafer, Y., Zhang, X., and Du, W. Harvesting inconsistent security configurations in custom android roms via differential analysis. In: *USENIX SEC'16*.

[7] Aafer, Y., Zhang, X., and Du, W. Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis. In: *USENIX SEC'16*.

[8] Acar, Y., Backes, M., Bugiel, S., Fahl, S., McDaniel, P., and Smith, M. Sok: lessons learned from android security research for appified software platforms. In: *IEEE S&P'16*.

[9] alfink. *GitHub - ASIS CTF 2018 Finals - Gunshop Module.* URL: `https://github.com/alfink/asisfinals2018-gunshop-module` (Accessed: July 3, 2020).

[10] Amazon. *Fire OS Overview.* URL: `https://developer.amazon.com/docs/fire-tv/fire-os-overview.html` (Accessed: July 3, 2020).

[11] Android Developer Documentation. *Bound services overview.* URL: `https://developer.android.com/guide/components/bound-services` (Accessed: July 3, 2020).

[12] Android Developer Documentation. *Manifest: sharedUserId.* URL: `https://developer.android.com/guide/topics/manifest/manifest-element#uid` (Accessed: July 3, 2020).

[13] Android Developer Documentation. *Platform Architecture.* URL: `https://developer.android.com/guide/platform` (Accessed: July 3, 2020).

[14] Android Developer Documentation. *UI/Application Exerciser Monkey.* URL: `https://developer.android.com/studio/test/monkey` (Accessed: July 3, 2020).

[15] Android Developer Documentation. *Understand the Activity Lifecycle.* URL: `https://developer.android.com/guide/components/activities/activity-lifecycle` (Accessed: July 3, 2020).

[16] *android/platform/build/nougat-release.* URL: `https://android.googlesource.com/platform/build/+/nougat-release` (Accessed: July 3, 2020).

[17] AOSP. *AddressSanitizer.* URL: `https://source.android.com/devices/tech/debug/asan` (Accessed: July 3, 2020).

[18] AOSP. *android/platform/frameworks/native/nougat-release/./cmds/service.* URL: `https://android.googlesource.com/platform/frameworks/native/+/refs/heads/nougat-release/cmds/service/` (Accessed: July 3, 2020).

[19] *ARTist - The Android Runtime instrumentation and security toolkit.* URL: `https://github.com/Project-ARTist/ARTist` (Accessed: July 3, 2020).

[20] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: *ACM PLDI '14*.

[21] Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., and Holz, T. Redqueen: fuzzing with input-to-state correspondence. In: *NDSS'19*.

[22] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. Pscout: analyzing the android permission specification. In: *ACM CCS'12*.

[27] Backes, M., Bugiel, S., and Derr, E. Reliable third-party library detection in android and its security applications. In: *ACM CCS'16*.

[28] 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: *USENIX SEC'16*.

[29] Backes, M., Gerling, S., Hammer, C., Maffei, M., and Styp-Rekowsky, P. von. Appguard–enforcing user requirements on android apps. In: *TACAS'13*.

[30] Bianchi, A., Fratantonio, Y., Kruegel, C., and Vigna, G. Njas: sandboxing unmodified applications in non-rooted devices running stock android. In: *ACM CCS SPSM'15*.

[31] Bits, T. of. *DeepState.* URL: `https://github.com/trailofbits/deepstate` (Accessed: July 3, 2020).

[32] Book, T., Pridgen, A., and Wallach, D. S. Longitudinal analysis of android ad library permissions. *arXiv preprint arXiv:1303.0857* (2013).

[33] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. Towards Taming Privilege-Escalation Attacks on Android. In: *NDSS'12*.

[34] Cao, C., Gao, N., Liu, P., and Xiang, J. Towards Analyzing the Input Validation Vulnerabilities Associated with Android System Services. In: *ACM ACSAC'15*.

[35] Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., and Chen, Y. EdgeMiner: automatically detecting implicit control flow transitions through the android framework. In: *NDSS'15*.

[36] *Codelib.* URL: `https://github.com/Project-ARTist/template-codelib` (Accessed: July 3, 2020).

[37] Copperhead. *CopperheadOS - A security and privacy focused mobile operating system compatible with Android apps.* URL: `https://copperhead.co/android/` (Accessed: July 3, 2020).

[38] Cotroneo, D., Iannillo, A. K., and Natella, R. Evolutionary fuzzing of Android OS vendor system services. *Empirical Software Engineering* (2019).

[39] Cox, L. P., Gilbert, P., Lawler, G., Pistol, V., Razeen, A., Wu, B., and Cheemalapati, S. Spandex: secure password tracking for android. In: *USENIX SEC'14*.

[40] Daniel Micay. *GrapheneOS*. URL: `https://grapheneos.org/` (Accessed: July 3, 2020).

[41] Davi, L., Dmitrienko, A., Sadeghi, A.-R., and Winandy, M. Privilege escalation attacks on android. In: *ISC'10*.

[42] Davis, B. and Chen, H. Retroskeleton: retrofitting android apps. In: *ACM MobiSys'13*.

[43] Davis, B., Sanders, B., Khodaverdian, A., and Chen, H. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In: *IEEE MoST'12*.

[44] Dawoud, A. and Bugiel, S. Bringing Balance to the Force: Dynamic Analysis of the Android Application Framework. Under Submission.

[45] Documentation, A. D. *Android Debug Bridge (adb)*. URL: `https://developer.android.com/studio/command-line/adb` (Accessed: July 3, 2020).

[46] Documentation, A. D. *Android Interface Definition Language (AIDL)*. URL: `https://developer.android.com/guide/components/aidl` (Accessed: July 3, 2020).

[47] Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., and Whelan, R. Lava: large-scale automated vulnerability addition. In: *IEEE S&P'16*.

[48] Elenkov, N. *Android security internals: An in-depth guide to Android's security architecture*. No Starch Press, 2014.

[49] Enck, W., Gilbert, P., Han, S., Tendulkar, V., 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. *ACM TOCS* 32, 2 (2014).

[50] Enck, W., Octeau, D., McDaniel, P. D., and Chaudhuri, S. A study of android application security. In: *USENIX SEC'11*.

[51] Erik Derr (reddr. *GitHub - axplorer/permissions/*. URL: `https://github.com/reddr/axplorer/tree/master/permissions` (Accessed: July 3, 2020).

[52] Facebook. *Stetho - A debug bridge for Android applications*. URL: `https://facebook.github.io/stetho/` (Accessed: July 3, 2020).

[53] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. Permission re-delegation: attacks and defenses. In: *USENIX SEC'11*.

[54] Feng, H. and Shin, K. G. BinderCracker: Assessing the Robustness of Android System Services. *arXiv preprint arXiv:1604.06964* (2016).

[55] *Flask*. URL: `http://flask.pocoo.org/` (Accessed: July 3, 2020).

[56] Gallopsled. *pwntools.* URL: http://docs.pwntools.com/en/stable/ (Accessed: July 3, 2020).

[57] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[58] Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., and Chen, Z. Collafl: path sensitive fuzzing. In: *IEEE S&P'18.*

[59] *GitHub - preeny.* URL: https://github.com/zardus/preeny (Accessed: July 3, 2020).

[60] Gong, G. *Slides: Fuzzing Android System Services by Binder Call to Escalate Privilege.* URL: https://www.blackhat.com/docs/us-15/materials/us-15-Gong-Fuzzing-Android-System-Services-By-Binder-Call-To-Escalate-Privilege.pdf (Accessed: July 3, 2020).

[61] Google. *Android Runtime (ART) and Dalvik.* URL: https://source.android.com/devices/tech/dalvik (Accessed: July 3, 2020).

[62] Google. *Changes to binder driver.* URL: https://source.android.com/devices/architecture/hidl/binder-ipc?hl=en (Accessed: July 3, 2020).

[63] Google. *Chrome DevTools.* URL: https://developers.google.com/web/tools/chrome-devtools (Accessed: July 3, 2020).

[64] Google. *Compatibility Test Suite.* URL: https://source.android.com/compatibility/cts (Accessed: July 3, 2020).

[65] Google. *Cuttlefish Virtual Android Devices.* URL: https://source.android.com/setup/create/cuttlefish (Accessed: July 3, 2020).

[66] Google. *Debugging Android JNI with CheckJNI.* URL: https://android-developers.googleblog.com/2011/07/debugging-android-jni-with-checkjni.html (Accessed: July 3, 2020).

[67] Google. *Flutter.* URL: https://flutter.dev/ (Accessed: July 3, 2020).

[68] Google. *Fuzzing with libFuzzer.* URL: https://source.android.com/devices/tech/debug/libfuzzer (Accessed: July 3, 2020).

[69] Google. *GitHub: OSS-Fuzz - Trophies.* URL: https://github.com/google/oss-fuzz#trophies (Accessed: July 3, 2020).

[70] Google. *Google Play Console.* URL: https://play.google.com/apps/publish/signup/ (Accessed: July 3, 2020).

[71] Google. *Google Play Store - Google Play services.* URL: https://play.google.com/store/apps/details?id=com.google.android.gms&hl=de (Accessed: July 3, 2020).

[72] Google. *Implementing ART Just-In-Time (JIT) Compiler.* URL: https://source.android.com/devices/tech/dalvik/jit-compiler (Accessed: July 3, 2020).

[73] Google. *Improving app performance with ART optimizing profiles in the cloud.* URL: https://android-developers.googleblog.com/2019/04/improving-app-performance-with-art.html (Accessed: July 3, 2020).

[74] Google. *IssueTracker.* URL: https://issuetracker.google.com (Accessed: July 3, 2020).

[75] Google. *Queue the Hardening Enhancements.* URL: https://android-developers.googleblog.com/2019/05/queue-hardening-enhancements.html (Accessed: July 3, 2020).

[76] Google. *Security-Enhanced Linux in Android.* URL: https://source.android.com/security/selinux (Accessed: July 3, 2020).

[77] Google. *The Android Open Source Project.* URL: https://source.android.com/ (Accessed: July 3, 2020).

[78] Google. *Verifying app behavior on the Android runtime (ART).* URL: https://developer.android.com/guide/practices/verifying-apps-art (Accessed: July 3, 2020).

[79] Google Bughunter University. *Wayback Machine - Bugs with no security impact.* URL: https://web.archive.org/web/20170110154209/https://sites.google.com/site/bughunteruniversity/android/invalid-bugs (Accessed: July 3, 2020).

[80] *Google Code: Droidbox.* URL: https://code.google.com/p/droidbox/ (Accessed: July 3, 2020).

[81] Google Project Zero. *Return to libstagefright: exploiting libutils on Android.* URL: https://googleprojectzero.blogspot.com/2016/09/return-to-libstagefright-exploiting.html (Accessed: July 3, 2020).

[82] Google Project Zero. *Stagefrightened?* URL: https://googleprojectzero.blogspot.com/2015/09/stagefrightened.html (Accessed: July 3, 2020).

[84] Gorski, S. A., Andow, B., Nadkarni, A., Manandhar, S., Enck, W., Bodden, E., and Bartel, A. Acminer: extraction and analysis of authorization checks in android's middleware. In: *CODASPY'19*.

[83] Gorski, S. A. and Enck, W. Arf: identifying re-delegation vulnerabilities in android system services. In: *ACM WiSec'19*.

[85] Grace, M. C., Zhou, W., Jiang, X., and Sadeghi, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In: *ACM WiSec'12*.

[86] Hao, H., Singh, V., and Du, W. On the Effectiveness of API-level Access Control Using Bytecode Rewriting in Android. In: *ACM ASIACCS'13*.

[87] He, Q. *Slides: Hey your parcel looks bad - fuzzing and exploiting parcelization vulnerabilities in Android.* URL: https://www.blackhat.com/docs/asia-16/materials/asia-16-He-Hey-Your-Parcel-Looks-Bad-Fuzzing-And-Exploiting-Parcelization-Vulnerabilities-In-Android-wp.pdf (Accessed: July 3, 2020).

[88] Heise Medien GmbH & Co. KG. *Google Play Store - heise online - News.* URL: https://play.google.com/store/apps/details?id=de.heise.android.heiseonlineapp (Accessed: July 3, 2020).

[89] 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: *ACM CCS'11.*

[90] Huang, H., Zhu, S., Chen, K., and Liu, P. From system services freezing to system server shutdown in android: all you need is a loop in an app. In: *ACM CCS'15.*

[91] Iannillo, A. K., Natella, R., Cotroneo, D., and Nita-Rotaru, C. Chizpurfle: A gray-box android fuzzer for vendor service customizations. In: *IEEE ISSRE'17.*

[92] IBM. *Wala - T.J. Watson Libraries for Analysis.* URL: http://wala.sourceforge.net/wiki/index.php/Main_Page (Accessed: July 3, 2020).

[93] ibotpeaches. *A tool for reverse engineering Android apk files.* (Accessed: July 3, 2020).

[94] Intel. *Pin - A Dynamic Binary Instrumentation Tool.* URL: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool (Accessed: July 3, 2020).

[95] 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: *ACM SPSM'12.*

[96] Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. Evaluating fuzz testing. In: *ACM CCS'18.*

[97] Liu, B., Zhang, C., Gong, G., Zeng, Y., Ruan, H., and Zhuge, J. FANS: fuzzing android native system services via automated interface analysis. In: *USENIX SEC'20.*

[98] Liu, B., Liu, B., Jin, H., and Govindan, R. Efficient privilege de-escalation for ad libraries in mobile apps. In: *MobiSys'19.*

[99] Livshits, B. *Dynamic taint tracking in managed runtimes.* Tech. rep. Microsoft Research.

[100] Livshits, B. and Chong, S. Towards fully automatic placement of security sanitizers and declassifiers. In: *ACM SIGPLAN'13.*

[101] Lockwood, A. *Binder & Death Recipients.* URL: https://www.androiddesignpatterns.com/2013/08/binders-death-recipients.html (Accessed: July 3, 2020).

[102] Microsoft. *Xamarin.* URL: https://dotnet.microsoft.com/apps/xamarin (Accessed: July 3, 2020).

[103] *Mobile Operating System Market Share Worldwide - July 2020.* URL: http://gs.statcounter.com/os-market-share/mobile/worldwide (Accessed: July 3, 2020).

[104] *Monkey Troop - ARTist's evaluation tool.* URL: https://github.com/Project-ARTist/monkey-troop (Accessed: July 3, 2020).

[105] Moulu, A. *Slides: Android OEM's applications (in)security and backdoors without permission.* URL: http://www.sh4ka.fr/Android_OEM_applications_insecurity_and_backdoors_without_permission.pdf (Accessed: July 3, 2020).

[106] Now Secure. *Frida - Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers.* URL: https://frida.re (Accessed: July 3, 2020).

[107] Now Secure. *Stalker.* URL: https://www.frida.re/docs/javascript-api/#stalker (Accessed: July 3, 2020).

[108] Pearce, P., Felt, A. P., Nunez, G., and Wagner, D. Addroid: privilege separation for applications and advertisers in android. In: *ACM CCS'12.*

[109] Peng, H., Shoshitaishvili, Y., and Payer, M. T-fuzz: fuzzing by program transformation. In: *IEEE S&P'18.*

[110] Porter Felt, A., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In: *ACM CCS'11.*

[111] Project-ARTist. *GitHub - Stetho Module.* URL: https://github.com/Project-ARTist/stetho-module (Accessed: July 3, 2020).

[112] Projekt ARTist Team. *Welcome to the ARTist Project.* URL: https://artist.cispa.saarland (Accessed: July 3, 2020).

[113] *QEMU.* URL: https://www.qemu.org/ (Accessed: July 3, 2020).

[114] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., and Bos, H. Vuzzer: application-aware evolutionary fuzzing. In: *NDSS'17.*

[115] reddit inc. *Google Play Store - Reddit.* URL: https://play.google.com/store/apps/details?id=com.reddit.frontpage&hl=de (Accessed: July 3, 2020).

[116] Russello, G., Conti, M., Crispo, B., and Fernandes, E. Moses: supporting operation modes on smartphones. In: *ACM SACMAT'12.*

[117] Sabanal, P. *Hiding behind ART.* 2015. URL: https://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp.pdf (Accessed: July 3, 2020).

[118] Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., and Wang, X. Soundcomber: a stealthy and context-aware sound trojan for smartphones. In: *NDSS'11.*

[119] Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., and Holz, T. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In: *USENIX SEC'17.*

[120] Schütte, J., Titze, D., and De Fuentes, J. AppCaulk: data leak prevention by injecting targeted taint tracking into android apps. In: *TrustCom14.*

[121] Secure Software Engineering Group - Paderborn University. *GitHub - DroidBench 2.0.* URL: https://github.com/secure-software-engineering/DroidBench (Accessed: July 3, 2020).

[122] Seo, J., Kim, D., Cho, D., Shin, I., and Kim, T. FLEXDROID: Enforcing In-App Privilege Separation in Android. In: *NDSS'16.*

[123] Shao, Y., Ott, J., Chen, Q. A., Qian, Z., and Mao, Z. M. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In: *NDSS'16.*

[124] Shekhar, S., Dietz, M., and Wallach, D. S. Adsplit: separating smartphone advertising from applications. In: *USENIX SEC'12.*

[125] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: *IEEE S&P'16.*

[126] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Driller: augmenting fuzzing through selective symbolic execution. In: *NDSS'16.*

[127] Stevens, R., Gibler, C., Crussell, J., Erickson, J., and Chen, H. Investigating user privacy in android ad libraries. In: *IEEE MoST'12.*

[128] Sun, M. and Tan, G. Nativeguard: protecting android applications from third-party native libraries. In: *ACM WiSec'14.*

[129] *The Fantastic Beasts Framework for the Android OS.* URL: https://github.com/dessertlab/fantastic_beasts (Accessed: July 3, 2020).

[130] The LineageOS Project. *LineageOS Android Distribution.* URL: https://lineageos.org/ (Accessed: July 3, 2020).

[131] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot: a java bytecode optimization framework. In: *CASCON First Decade High Impact Papers'10.*

[23] Various Authors. *Clang: a C language family frontend for LLVM.* URL: https://clang.llvm.org/ (Accessed: July 3, 2020).

[24] Various Authors. *libFuzzer - A library for coverage-guided fuzz testing.* URL: https://llvm.org/docs/LibFuzzer.html (Accessed: July 3, 2020).

[25] Various Authors. *Randoop - Automatic unit test generation for Java.* URL: https://randoop.github.io/randoop/ (Accessed: July 3, 2020).

[26] Various Authors. *The LLVM compiler infrastructure.* URL: https://llvm.org/ (Accessed: July 3, 2020).

[132] Wu, D., Gao, D., Cheng, E. K. T., Cao, Y., Jiang, J., and Deng, R. H. Towards understanding android system vulnerabilities: techniques and insights. In: *ACM ASIA CCS'19.*

[133] Wu, J., Liu, S., Ji, S., Yang, M., Luo, T., Wu, Y., and Wang, Y. Exception Beyond Exception: Crashing Android System by Trapping in "uncaughtException". In: *ACM/IEEE ICSE'17.*

[134] Wu, L., Grace, M., Zhou, Y., Wu, C., and Jiang, X. The Impact of Vendor Customizations on Android Security. In: *ACM CCS'13*.

[135] XDA-Developers. *Xposed Framework Hub*. URL: https://www.xda-developers.com/xposed-framework-hub/ (Accessed: July 3, 2020).

[136] Xing, L., Pan, X., Wang, R., Yuan, K., and Wang, X. Upgrading Your Android, Elevating My Malware: Privilege Escalation Through Mobile OS Updating. In: *IEEE S&P'14*.

[137] Xu, R., Saïdi, H., and Anderson, R. Aurasium – Practical Policy Enforcement for Android Applications. In: *USENIX SEC'12*.

[138] 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: *ICSE'16*.

[139] Zalewski, M. *american fuzzy lop*. URL: http://lcamtuf.coredump.cx/afl/ (Accessed: July 3, 2020).

[140] Zalewski, M. *AFL - Understanding the status screen*. URL: http://lcamtuf.coredump.cx/afl/status_screen.txt (Accessed: July 3, 2020).

[141] Zeller, A. Isolating Cause-effect Chains from Computer Programs. In: *ACM SIGSOFT SFE'02*.

[142] Zhang, X., Ahlawat, A., and Du, W. Aframe: isolating advertisements from mobile applications in android. In: *ACSAC'13*.

[143] Zhou, X., Lee, Y., Zhang, N., Naveed, M., and Wang, X. The peril of fragmentation: security hazards in android device driver customizations. In: *IEEE S&P'14*.