Florian Draschbacher

Exploring Misconfigured Security Across the Mobile Platform Stack

PhD Thesis

Assessors: Stefan Mangard, Sven Bugiel

June 2026

Institute of Information Security
Graz University of Technology

Abstract

Mobile devices running Android and iOS are indispensable parts of everyday life. In the context of many applications, they store, process, or transmit sensitive user data, motivating the need to integrate security mechanisms across all involved components. However, the effectiveness of these security mechanisms highly depends on their proper configuration. Misconfiguration can instill a false sense of security while still leaking sensitive user data to attackers.

In this thesis, we advance the state of the art in mobile security by exploring misconfigured security in three important areas across the mobile platform stack. First, we analyze mobile-specific communication interfaces. Specifically, we look into USB file exchange implementations on Android and iOS and data migration tools on Android. Both analyses uncover severe misconfigurations in mechanisms for connection authorization, device authentication, and data confidentiality. For USB file exchange, we demonstrate a novel attack family, ChoiceJacking, that allows malicious phone chargers to bypass existing mitigations and extract user files. For data migration tools, we show that apps from seven major smartphone vendors, including Samsung, Google, and Xiaomi, are susceptible to wireless attacks that allow data extraction or manipulation, even gaining code execution in some cases.

Second, we examine supply chain attacks on Android, specifically focusing on security mechanisms for app integrity protection. As part of recent changes to app distribution, Google mandates developers to hand over APK signing keys to app store operators. To mitigate the possibility of malicious manipulation, they introduced an additional signature scheme called Code Transparency. In our analyses, we demonstrate gaps in this scheme that allow manipulating app execution flows without invalidating the Code Transparency.

Third, we address API misuse in Android applications through a dynamic instrumentation-based approach. We use this approach to detect and mitigate crypto API misuse in compiled Android apps by intercepting method invocations. From interception monitors, we employ mitigation procedures for common crypto misuse cases that are transparent to the target app's program logic. We further extend our dynamic approach into

a generalized, application-agnostic patching pipeline that can be adapted to detect or mitigate further API misuse scenarios.

The contributions in this thesis improve the overall security of the mobile platform stack. Through responsible disclosure and vendor updates, our research has already had a profound positive impact on billions of end users. Additionally, we make tooling available that facilitates research, which we anticipate will ultimately benefit end users as well.

This thesis consists of two parts. The first part provides context for the individual contributions, including background information and an overview of the state of the art. The second part presents the five first-author papers that are the core contribution of this thesis, with only minor changes in layout. All contributions in this thesis are peer-reviewed and have been accepted at renowned conferences.

Acknowledgements

A project of this scale is never the product of just a single person. I therefore extend my heartfelt gratitude to the co-authors and colleagues who supported my journey through inspiring discussions, ideas, implementations, and innumerable more kinds of contributions (did I mention the cake?). Thank you, Lukas Maar, Mathias Oberhuber, Lorenz Schumm, Stefan More, Jakob Heher, Edona Fasllija, and Sudheendra Raghav Neela for allowing me to share not only the ups and downs of academic work, but also various topics of life besides it (apparently it does exist!). I am also grateful to Lukas Treffner, Rene Denifl, Andreas Karner, Michael Schaffer, and all the other motivated students who accompanied me in my explorations of mobile security. A special thanks is due to Johannes Feichtner, for motivating me to strive for the top and for helping me gain momentum during the crucial starting phase of my PhD. My sincere appreciation goes to my advisor, Stefan Mangard, for generously carving out time from his busy schedule whenever needed, and to my team leader, Arne Tauber, for giving me the freedom to pursue the research ideas of my choice and for trusting my instincts. I would also like to sincerely thank Sven Bugiel for taking the time to review this thesis and for his valuable feedback.

Finally, I would like to thank the people dearest to me, who probably suffered the most when I was under pressure pre-deadline, and still cheered the loudest about acceptance notifications. Thank you to my friends and family, my brother Thomas, and my parents Heimo and Regine for their relentless support throughout my career and far beyond. To my girlfriend Doris for changing the trajectory of my life. And for being mad at me whenever I try to downplay my accomplishments. I really did my best here!

Contents

Part I.

Exploring Misconfigured Security Across the Mobile Platform Stack

1

Introduction

Modern mobile devices, such as tablets and smartphones, have little in common with cell phones, their original ancestors. Although they have not lost the ability to accept and make phone calls on the go, that functionality has long faded from the spotlight. What has come instead, can be considered one of the most important computing milestones in recent history. Advancements in various involved technologies have enabled novel use cases that render a mobile device the centerpiece of its owner's digital life. Fast ubiquitous network connectivity enables mobile devices to bring the Internet into all aspects of everyday life, and vice versa. High-quality cameras and microphones have turned smartphones into consumers' primary choice for taking pictures, videos, or voice recordings. The vast array of additional sensors almost eliminates the gap between the physical and digital worlds. Users carry their smartphone wherever they go, making it a convenient authentication factor for digital services. All these use cases turn mobile devices into treasure troves of sensitive user data, increasing their appeal as targets for attackers.

At the same time, mobile phones offer vast potential for attacks. The high degree of connectivity inherently exposes large attack surfaces to remote or proximal attackers. The portability of these devices mean that physical attacks after device theft or loss are realistic possibilities. The tight OS integration of third-party applications opens the system to local attacks through malware or malicious libraries.

In this challenging environment, modern smartphones are complex technical systems. On an architectural level, they consist of a large number of hardware components, most with their own software stack, ranging from low-level firmware to high-level applications. Additionally, modern smartphones are part of a larger ecosystem that also includes proprietary quality-of-life services, such as app stores. All these components can be viewed as a three-tiered mobile platform stack. At the base of this stack

are the various hardware components and communication interfaces that make up a mobile device, their low-level firmware, and the OS kernel. Above it, a middleware layer provides the foundational services for the user-facing application ecosystem. Finally, the top layer consists of the user-facing applications that implement concrete use cases for the device.

Security needs to be considered in each individual component at every layer of the mobile platform stack. A vulnerability in one component can already allow considerable access and often offers opportunities for further escalation. For example, hardware-level vulnerabilities in chip firmwares [278, 286], device drivers [5, 39, 192, 249], or the bootchain [35, 41, 146] have in the past allowed device compromise on both iOS and Android. Common exploitation targets in the middleware layer include vulnerabilities in inter-process communication (IPC) [38, 138, 250] or other system libraries [37, 308], permission systems [88, 129, 238], or trust anchors such as platform signing keys [237]. Finally, application-layer attacks commonly take advantage of malicious or vulnerable dependencies [202, 203, 218, 239] or general memory and logic bugs [136, 243, 254] to leak user data from or gain code execution in a process.

To address the security concerns on each layer of the mobile platform stack, system designers and app developers need to integrate security mechanisms. For example, common security mechanisms include cryptography to ensure the confidentiality, integrity, and authenticity of data, or user consent prompts for authorization of sensitive actions. However, when integrating such security mechanisms, their design and implementation details must be carefully tailored to the specific assets and threat scenarios. Any misconfiguration or flawed implementation can lead to an ineffective security mechanism that leaves exploitable gaps or can be bypassed entirely. Given the false sense of security instilled in end users and developers, ineffective security mechanisms can be considered worse than no such mechanisms at all. This motivates a need for rigorous examinations of both design and implementation aspects of such security mechanisms. In this thesis, we explore misconfigured security mechanisms in three important areas across the mobile platform stack.

(1) Communication Interfaces: Mobile devices are equipped with a large number of communication modules, including cellular modems, Wi-Fi and Bluetooth chips, and USB or NFC interfaces. On top of these low-level hardware interfaces, vendors often implement proprietary protocols and services that are highly security-critical. For all communication, the confidentiality, integrity, and authenticity of the

exchanged data are crucial. Under no circumstances should it be possible for an unauthorized party to extract, manipulate, or inject messages into the communication. Additionally, care must be taken to ensure that only authorized parties can initiate data exchanges. All of these requirements need to be addressed through properly configured security mechanisms. Any misconfiguration of the security mechanisms can have serious consequences for the user's privacy. For example, wireless communication can be eavesdropped on or manipulated by proximal attackers unless encryption is used and configured with a key strong enough to resist brute-forcing. A particularly exposed interface in smartphones is USB, which is used for both charging and data exchange, so that only properly implemented user authentication can prevent malicious chargers from stealing data. Across mobile communication interfaces, researchers have uncovered severe vulnerabilities in the recent past [3, 52, 58, 74, 137, 142, 148, 244, 245], motivating further analyses in this thesis.

(2) Supply Chains: On all stack layers, mobile platforms are complex assemblies of parts from different origins. Malicious parties somewhere along the supply chain from each respective developer to the end user could introduce vulnerabilities or back doors. Over the last few years, such supply chain attacks have led to compromises of various other platforms [1, 85, 89, 151]. It is therefore crucial to also identify gaps in mobile supply chain security mechanisms and adjust overarching trust models for reducing the impact attacks can have. Asymmetric cryptography offers a way to incorporate authenticity and integrity proofs into supply chains of software components. For example, embedding a digital signature into application packages proofably ensures that no one has tampered with the package after it was signed by the developer. However, despite the cryptographic soundness of the signature itself, it only offers effective protection if it is properly configured and integrated into the overall signature scheme. Any misconfiguration may allow manipulation of app behavior without invalidating the signature, ultimately enabling an attacker to gain code execution in the context of an arbitrary application. For example, APK signature verification in early Android versions contained an inconsistency when dealing with duplicate files in a package, allowing code injection while keeping the signature intact [86]. Such past misconfigurations motivate the need to scrutinize subsequently released app signature schemes, such as Code Transparency for the new mandatory Android App Bundle distribution format [78, 97].

(3) API Misuse: The application programming interface (API) of mobile platforms allows applications to take advantage of the extensive functionality provided by lower layers. Commonly, the offered functionality is related to the security mechanisms that apps integrate. Among others, there are platform APIs for cryptographic primitives, app runtime integrity attestation, or user authentication. In the past, researchers have uncovered widespread mistakes in the integration and configuration of these APIs that void their security guarantees [126, 169, 199, 208, 299, 303]. Particularly common examples are flawed TLS certificate validation logic that enables Man-In-The-Middle attacks or hardcoded encryption keys that allow trivial recovery of plaintexts. For hardening applications against threats, it is necessary to understand which APIs are prone to misuse and to detect resulting vulnerabilities in real-world apps. Additionally, solutions have to be found for mitigating instances of known API misuse cases in apps whose developers are unable or unwilling to implement fixes.

1.1. Main Contributions

This PhD thesis encompasses five first-author papers. One of them was published at a top-tier security conference, three at tier-2 conferences, and one at a tier-3 conference. These publications are:

- **Clone2Pwn**: A Systematic Security Analysis of Data Migration Tools in the Android Ecosystem [72]
- **ChoiceJacking**: Compromising Mobile Devices through Malicious Chargers like a Decade Ago [71]
- **Manifest Problems**: Analyzing Code Transparency for Android Application Bundles [70]
- **A2P2**: An Android Application Patching Pipeline Based On Generic Changesets [68]
- **CryptoShield**: Automatic On-Device Mitigation for Crypto API Misuse in Android Applications [69]

These five publications advance the state of the art in mobile security by exploring security misconfigurations across a wide spectrum of the platform stack. They can be assigned to the three important areas described above: Secure Communication Interfaces, Supply Chain Security, and API Misuse.

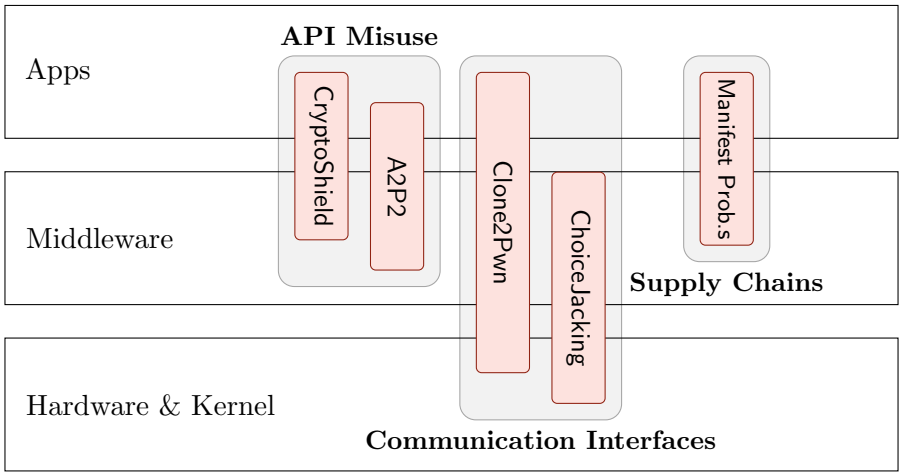


Figure 1.1: The main contributions in this thesis, Clone2Pwn [72], ChoiceJacking [71], Manifest Problems [70], A2P2 [68], and CryptoShield [69], organized into three areas and arranged by their covered platform stack parts.

Figure 1.1 presents an overview of the five first-author papers, as well as the covered platform stack parts and areas within mobile security. The following sections describe the contributions for each area.

1.1.1. Secure Communication Interfaces

While communication interfaces have been understood as attack vectors in mobile platforms for a long time [185, 277, 291], many previous works have focused on their foundational aspects. They considered communication and its security mostly as a concern of low-level hardware [60, 137, 142], drivers/daemons [102, 115, 130, 143], and specifications [6, 93, 226, 257]. In our contributions in this field, we focus on security misconfigurations in higher-level vendor-specific implementations of standardized protocols and entirely proprietary protocols.

Novel Attacks Through Malicious Chargers. The USB interface of mobile devices offers a unique and complex attack surface for devices that an attacker has physical access to. In contrast to all other interfaces, connections can be enforced onto the device even while it is locked, by establishing a physical link. This is one of the reasons why USB-based attacks are commonly used in commercial smartphone forensics tools [5].

However, these attacks usually exploit low-level kernel memory bugs, which means that they only work for specific smartphone manufacturers and OS versions. We therefore aim to investigate USB attacks based on vulnerabilities in higher-level trust models that can be exploited across vendors. In 2012, such an attack called JuiceJacking gained public attention [149, 153]. It exploited the fact that the single USB port in mobile devices is used for both charging and file exchange with a computer. A manipulated charger could therefore stealthily establish a file exchange connection during charging to steal user data. As a mitigation to this threat, iOS and Android added the requirement for explicitly activating the file exchange protocol for each USB connection through user interaction. This countermeasure was widely believed to be effective, and subsequent higher-level USB-based attacks concentrated on other, less powerful attack vectors [183, 184, 196] or side channels [60, 133, 274, 292]. We introduce ChoiceJacking, a novel class of USB-based attacks against mobile devices and the first to circumvent existing JuiceJacking defenses. We observe that current mitigations rely on the assumption that malicious chargers cannot inject input events while establishing a file exchange connection. While this assumption holds when considering the low-level USB specification in isolation, we identify upper-layer components on mobile devices that offer opportunities for workarounds. We describe a platform-agnostic attack concept along with three concrete techniques targeting Android and iOS, enabling a malicious charger to autonomously spoof user input and activate its own data connection. Our evaluation, based on a low-cost custom malicious charger, reveals a concerning state of USB security across Android and iOS. Despite vendor-specific USB stack customizations, ChoiceJacking attacks successfully access sensitive user data on all tested devices from eight vendors, including the six largest by market share. For two vendors, the attacks even allow data extraction from locked devices. To stealthily carry out attacks that require an unlocked device, we leverage a power-line side channel to identify moments when users are unlikely to notice visual telltales. This work was published at the 2025 USENIX Security Symposium [71] in collaboration with Lukas Maar, Mathias Oberhuber, and Stefan Mangard.

Security Analysis of Android Data Migration Tools. All major Android device vendors provide dedicated data migration tools that operate over wireless ad-hoc networks. Given that these tools handle highly sensitive user data, a secure design and implementation is paramount. Despite this, the only independent security analysis to date was conducted in 2019 and lacks coverage of current market leaders [173]. In our

work titled Clone2Pwn, we expand this state-of-the-art in two significant ways. First, we introduce a methodology that enables systematic, extensible analysis. We base this effort on a set of possible attack scenarios against data migration tools, for which we identify necessary design and implementation requirements. Second, our analyses represent the current landscape of state-of-the-art data migration tools, encompassing seven apps from major smartphone manufacturers. We show that all evaluated apps are vulnerable to wireless attacks due to security misconfigurations of lower-level Wi-Fi functionality. Specifically, six of the seven apps rely on WPA2 pre-shared keys that can be realistically brute-forced, either in near-realtime or within minutes. The remaining app uses Wi-Fi Direct, but is vulnerable to attacks due to flaws in peer identification. As a result, adversaries can eavesdrop on transmitted user data and, in some cases, modify it in transit. Such attacks can be conducted using inexpensive equipment positioned within Wi-Fi range of locations such as mobile carrier stores, where data migrations commonly occur. In addition, for three tools, a malicious app can extract or manipulate user data due to missing authentication on exposed TCP sockets. This paper was a collaboration with Lukas Maar, Lorenz Schumm, Rene Denifl, Lukas Treffner, and Stefan Mangard, and has been accepted for publication at the European Symposium on Research in Computer Security (ESORICS) in 2026 [72].

1.1.2. Supply Chain Security

Previous work on supply chain security for mobile platforms focused mostly on threats caused by malicious libraries in otherwise benign applications [168, 213, 269, 273]. While both iOS and Android have app stores designed as primary channels for app distribution, their potential for supply chain attacks remains underexplored. Our contributions in this domain concentrate on the interplay between recent changes in Google Play app distribution [78] and Android's platform security model [180].

Flaws in Code Transparency for Android App Bundle. In 2018, Google proposed the new Android App Bundle (AAB) distribution format and published tooling to foster adoption by independent Android app stores [79, 96]. In the new distribution paradigm, the final APK files for installation on end-user devices are generated and signed by the app store operator. Three years later, Google mandated that all new apps uploaded to Google Play use the new format and paradigm [78]. For developers, this change meant handing over ultimate control over APK contents to the

app store operator. To prevent misconduct, Google introduced the Code Transparency (CT) integrity scheme as an optional extension to AAB [97]. At its core, Code Transparency is meant to allow detecting downstream changes to the app by having the developer embed a signature in the AAB. The signature is later copied into every generated APK file and covers all executable parts of the app. Despite this radical change in the Android app supply chain, no independent security analyses of CT have been conducted. In our work titled Manifest Problems, we address this gap. We contrast the AAB-induced change in the trust model with the role APK signatures play in the platform security model. Additionally, we analyze 3.5 million apps from Google Play and Huawei AppGallery regarding their use of AAB and CT. Our research uncovers multiple ways for app store operators to inject code while keeping the Code Transparency intact. This work was a collaboration with Lukas Maar, published at the Annual Computer Security Applications Conference (ACSAC) 2024 [70].

1.1.3. Vulnerabilities Resulting from API Misuse

Most existing works on API misuse on mobile platforms focused on detecting it (in compiled apps) and relied on approaches based on static analysis [188]. However, static analysis is generally inefficient and/or suffers from severe imprecisions [4]. We therefore focus our contributions in this field on dynamic methods that can also be used for mitigating API misuse.

Mitigating Crypto API Misuse in Compiled Android Apps. A series of publications over the last few years has shown that Android applications commonly suffer from vulnerabilities caused by misuse of cryptographic APIs [75, 82, 207]. Many of these APIs are offered by the platform, and are therefore easy to integrate into apps. However, the complexity of these APIs frequently results in mistakes that, e.g., leak sensitive user data. From the large volume of publications investigating the prevalence of these vulnerabilities (cf. Section 3.4), we know the problem affects the overwhelming majority of mobile apps. Still, despite the problem and its prevalence being known for years and various initiatives by Google, no effective systemic mitigation has been found. Not only do developers keep re-introducing crypto API misuse into their apps [91], users are also left unprotected if they rely on vulnerable apps that are no longer maintained. We therefore propose CryptoShield to mitigate crypto API misuse in compiled apps. CryptoShield is a complete system that can be installed on any Android device. It injects a monitoring module into

any application package installed on the device. This module intercepts calls to cryptographic APIs, checking passed parameters to detect misuse. For detected misuses, CryptoShield includes generic mitigation procedures that are transparent to the app. We demonstrate the efficiency and efficacy of CryptoShield through large-scale automated and manual tests on popular apps, and synthetic benchmarks. This paper was published at the ACM ASIA Conference on Computer and Communications Security (AsiaCCS) in 2023 [69] in collaboration with Johannes Feichtner.

Android App Patching via Generic Changesets. To identify API misuse candidates and to detect or mitigate concrete misuse cases, researchers require tools to inspect the runtime behavior of compiled applications. However, existing tooling lacks a comprehensive, application-agnostic approach. It either requires manual adaptation for each target application or focuses exclusively on executable code, neglecting the app manifest and resources, which play a key role in an app's system integration. Consequently, these tools are ill-suited for research scenarios that require systematic transformations across diverse parts of large app sets. We therefore introduce A2P2, a flexible patching pipeline for compiled Android applications. A2P2 features a custom declarative patch specification format that supports complex transformations across all components of an application package. Patches can be applied to an arbitrary number of Android application Package (APK) files using a configurable pipeline whose stages can be reordered, extended, or augmented with user-defined stages. This design enables the construction of sophisticated transformations from a small set of fundamental primitives. We evaluate A2P2 by measuring deployment time and assessing its impact on application compatibility, package size, and runtime performance across representative use cases. Finally, we demonstrate its practical utility by implementing patches that reproduce prior research results and support common tasks in Android security analysis. This work was published at the 2023 International Conference on Availability, Reliability and Security (ARES) [68] and received the Best Research Paper Award out of 177 accepted and about 850 submitted regular papers.

1.2. Other Contributions

In addition to the first-author contributions listed above, I also co-authored three further papers in the domain of mobile security. Two of these were

published at top-tier conferences. While my first-author papers overall lean towards the higher-level components of the mobile platform stack, most of my co-authored work focuses on the security of lower-level components such as the kernel, drivers, and hardware. Overall, my contributions therefore cover the entire mobile platform stack.

Userspace-Accessible Kernel Attack Surface on Android. Traditionally, Android end-to-end attacks escalated across the platform stack from unprivileged apps to privileged system services, and exploited flaws in the kernel attack surface from there. However, in recent years, attacks have shifted to directly attacking the kernel attack surface reachable from unprivileged apps. While Google acknowledged these attacks, they mostly focused their systemic hardening efforts on GPU drivers. In our work Doom of Device Drivers, we conduct a large-scale analysis to identify additional kernel drivers exposed to unprivileged applications. We then enumerate known security vulnerabilities in these drivers and check device firmwares for inclusion of corresponding patches. Our research uncovers widespread n-day vulnerabilities in these particularly sensitive components of the Android platform stack. Corroborating these findings, concurrent research demonstrates active exploitation of vulnerabilities in kernel drivers accessible from unprivileged apps. This paper was published at USENIX Security Symposium in 2025 [176] together with Lukas Maar, Lorenz Schumm, Martínez García, and Stefan Mangard.

Android Kernel Defense-In-Depth Mechanisms. Upstream Linux offers defense-in-depth mechanisms intended to thwart exploitation of various memory vulnerability classes. However, it is unclear to what degree these mechanisms are present in downstream vendor-specific Android kernels prior to the adoption of the Generic Kernel Image (GKI). In our work Defects-In-Depth, we address this gap. We start by collecting recent n-day vulnerabilities in Android kernels and classify their required exploitation techniques. For these techniques, we then identify defense-in-depth mechanisms from mainline Linux that would be effective against them. Lastly, we conduct a large-scale firmware analysis to determine whether these defense-in-depth mechanisms are employed on Android devices. Our research uncovers that Android devices frequently fail to integrate these existing and effective mechanisms. This paper was published at USENIX Security Symposium in 2024 [175] together with Lukas Maar, Lukas Lamster, and Stefan Mangard.

Ensuring Network Availability through TEE. Critical networking applications such as Wireless Emergency Alerts (WEA) or Multi-Factor

Authentication Notifications (MFA) commonly rely on kernel-level networking drivers and stacks. However, given their complex functionality and the resulting attack surface, mobile OS kernels can be compromised. Through exploiting vulnerabilities, an adversary could gain kernel-level system access and prevent the delivery of critical messages such as WEAs. In our paper Don't Stop Receiving, we propose moving a minimal network driver into the Trusted Execution Environment (TEE). This allows utilizing trusted I/O to ensure network availability for critical applications even under a compromised kernel. We implement a PoC of our solution and demonstrate that it has negligible runtime overhead when compared to hosting the network driver in the normal world. This paper was published in 2025 at the International Symposium on Network Computing and Applications (NCA) [255] as a joint work with Tom Van Eyck, Stefan More, Sam Michiels, and Danny Hughes.

1.3. Outline

This thesis is structured as follows. Chapter 2 provides background on the mobile platform stacks of Android and iOS, as well as key characteristics of their security architectures. Chapter 3 organizes the state of the art in mobile security in general, and in the subfields this thesis contributes to. Chapter 4 concludes the contextualization of the contributions in this thesis. The first-author papers are presented in Chapters 5 to 9, along with a list of all first- and co-author papers.

2

Background

The device scope referred to by the term *mobile* in mobile platforms, mobile security, or mobile platform security is not consistently defined. In the literal sense, it extends to any computing device that the user can move from one place to another. Although this would even include most of today's desktop computers, the common consensus is that the devices in scope need to be intended for non-stationary use. However, this still leaves a relatively wide range of devices, from very resource-constrained microcontrollers to laptops with desktop-grade computing power. For this thesis, we define the term mobile platforms as equivalent to smartphone platforms. This definition is consistent with major academic systems security conferences [128]. It is worth noting that it does not necessarily restrict the devices in scope to exclusively the smartphone form factor. Instead, we consider all devices in scope that run an operating system primarily targeted at smartphones, such as tablets. In this chapter, we provide background information on iOS[1] and Android, the dominant mobile platforms by the above definition. Beyond descriptions of the respective platform stacks, we highlight the security challenges and solutions unique to iOS and Android.

2.1. The Mobile Platform Stack

Today's mobile platform market is split between two players: Android (72 %) and iOS (28 %) [242]. After the introduction of the respective pioneering devices, the Apple iPhone (2007) and HTC Dream (2008), the two platforms quickly eliminated all existing competitors. While Android and iOS share many similarities in their high-level platform

[1]We use the term iOS to include both iOS and iPadOS, which, despite different marketing names, are almost identical from a security perspective.

stacks, there are key differences at the technical and organizational levels. In the following, we briefly introduce both platforms before describing the commonalities and differences in their platform stacks.

2.1.1. Android

Android is an open-source mobile operating system developed by Google under the Android Open Source Project (AOSP). While based on the Linux kernel, the system's core user-space components are purpose-built for mobile devices. Although development itself does not occur in public, the latest source code of the AOSP project is published twice a year [216]. The availability of source code results in a significantly different starting position for security researchers and malicious parties than on iOS. However, it is worth noting that the published sources do not necessarily reflect the code running on real devices. While the kernel is GNU-licensed, which requires source publication of derivative works, AOSP uses the more lenient Apache 2.0 license. As a result, device manufacturers are free to develop closed-source derivatives and usually make extensive use of this right. In fact, many vendors heavily customize the AOSP codebase and even introduce their own naming for the resulting OSs. Examples are Xiaomi's HyperOS/MIUI [284], Vivo's Funtouch OS [260], Oppo's ColorOS [201], Honor's MagicOS [118], or Huawei's EMUI [123]. To ensure a consistent ecosystem for app developers and end users, devices have to undergo compatibility testing [180]. Vendors can also have their devices certified by Google to ship Google Mobile Services (GMS). GMS is a software suite that includes standalone apps such as Play Store and Gmail, as well as components that can be integrated into third-party apps, e.g., for advertisements or maps.

2.1.2. iOS

iOS is a mobile OS developed by Apple and exclusively shipped with the company's devices. It shares its technical foundations with other Apple platforms, such as macOS, tvOS, or watchOS. Parts of these foundations, including the XNU kernel and some user-space components, are open-sourced as the Darwin project [29]. However, iOS includes countless additional closed-source subsystems, services, and frameworks. In practice, this either means that security researchers have to treat these components as black boxes or tediously reverse-engineer implementation details.

2.1.3. Platform Stack

In the following, we describe the platform stacks of both Android and iOS, highlighting similarities and differences in all layers. It is worth noting that individual devices might slightly deviate from the descriptions, in particular on Android, where many implementation details are up to device vendors. Still, there are not only strong similarities between devices of the same platform, but between the two platforms. Figure 2.1 presents an overview of the involved key components on Android and iOS.

2.1.3.1. Hardware & Kernel

At the lowest level, both Android and iOS include a large number of tightly integrated hardware components. The arguably most notable of these components is the System-on-Chip (SoC), which includes not only the CPU but also a number of integrated coprocessors and peripherals on both platforms. For most smartphones, tablets, and related devices, the CPU is built on the ARM instruction set architecture. Typical coprocessors and peripherals include the GPU, signal processors, or communication interfaces [214]. Further connectivity, such as cellular, Wi-Fi, Bluetooth, USB, GPS, NFC, and more, can also be implemented as separate Integrated Circuits (ICs). Many of these coprocessors and peripherals, in turn, include programmable computing cores, which run their own firmware. Both Android and iOS base the integrity of the system and confidentiality of particularly sensitive data on a Hardware Security Module (HSM). On iOS, this HSM is implemented as a separate computing core within the SoC, called Secure Enclave Processor (SEP) [177]. Most Android vendors use ARM TrustZone, a Trusted Execution Environment (TEE) within the ARM core [105]. Still, Android also supports secure elements with dedicated processing cores.

On both Android and iOS, a privileged kernel provides a POSIX-compliant interface to userspace. While Linux kernels shipped with Android devices used to be individually maintained and built by device vendors, this has changed in recent years. Modern devices usually ship with a Generic Kernel Image (GKI) compiled by Google and load vendor-specific modules only as needed [103]. This change partly eliminates security issues caused by fragmentation. Since Apple has full control over the iOS device landscape, their patch integration is typically faster. iOS's XNU kernel consists of a Mach microkernel augmented with higher-level functionality from

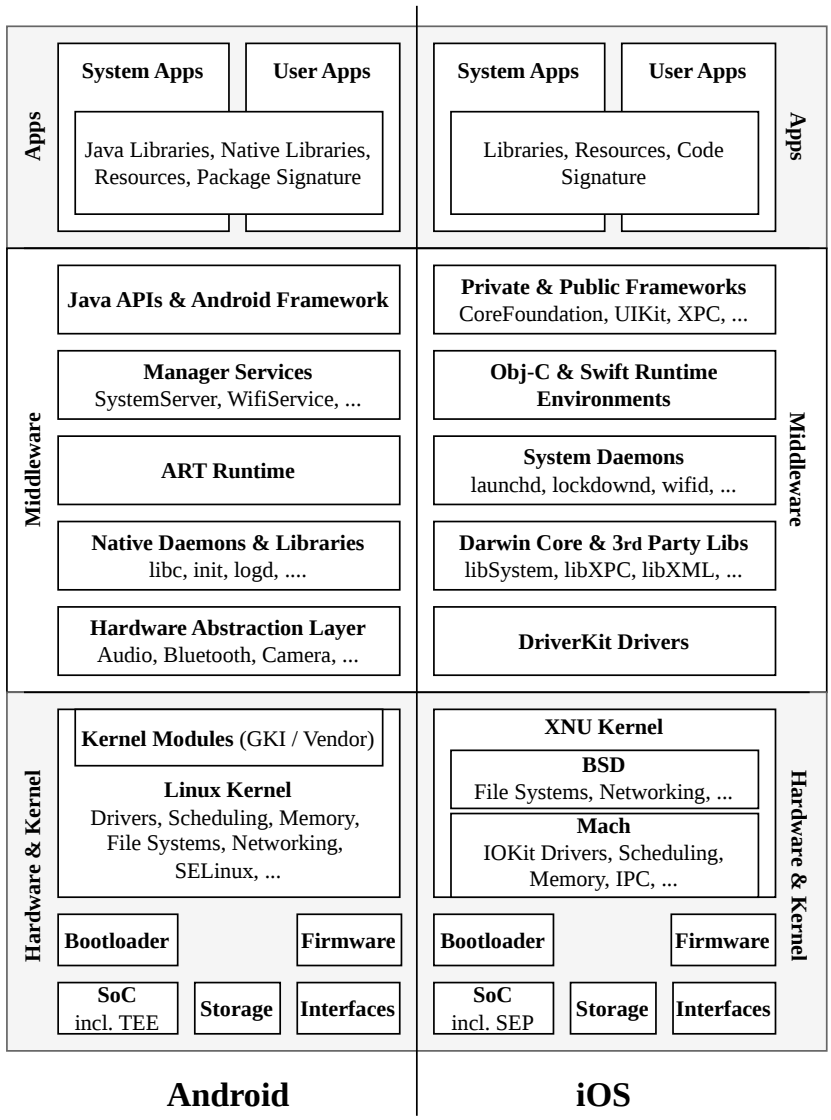


Figure 2.1: Platform stacks of Android and iOS. Based on official Android documentation [95, 98] and Jonathan Levin [159].

FreeBSD [159]. Notably, the OS's architecture allows some device drivers to run in userspace, through a framework called DriverKit.

2.1.3.2. Middleware

A key middleware-level architectural element on Android is the Hardware Abstraction Layer (HAL) [104]. It consists of vendor-supplied libraries and services that abstract low-level hardware functionality into clearly defined interfaces. Ultimately, it enables a unified app ecosystem across diverse Android devices by allowing applications and higher-level platform components to be agnostic of hardware details. Further native libraries implement low-level shared functionality such as access to kernel-provided interfaces and logging [98]. Among others, this functionality is consumed by low-level daemons for general device management and vendor-agnostic hardware subsystems. While a complex HAL is not necessary on iOS due to a lack of device diversity, much of its middleware layer is similar to Android's. The low-level libraries on the platform encompass open-source components from Darwin and other projects, as well as closed-source components [159]. They again form the basis for system daemons for general device management and hardware subsystems.

Higher-level components in the middleware layer prepare the ground for the application layer. On both platforms, they include a dedicated runtime environment. On iOS, applications are written in Objective-C or Swift. While these languages offer advanced features such as reflection, they are directly compiled to machine code and executed natively on the CPU [159]. The runtime environment, therefore, consists of a relatively lightweight collection of support libraries and frameworks that implement interaction with lower-level components. On Android, applications are usually written in Java or Kotlin and are compiled to Dalvik bytecode. For execution, Dalvik bytecode is either interpreted or further compiled into machine code. A more complex runtime environment is therefore needed, called Android Runtime (ART) [100]. It not only implements garbage collection, but also keeps track of frequently executed app parts to prepare them for Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilation. Applications can interact with the system and other apps through APIs in the Android frameworks. Many of these APIs use Android's Binder IPC subsystem to communicate with central subsystem-specific manager services. For example, such services exist for communication interfaces, sensors, power management, package management, and many more.

2.1.3.3. Applications

Both iOS and Android ship with preinstalled apps and allow installation of third-party software. Preinstalled apps are commonly developed by the device vendor or its partners, and run with higher privileges than user-installed apps. In all cases and across platforms, modern apps are complex software packages that include compiled code and assets from an array of different sources. In addition to first-party code, apps typically integrate both open-source projects and proprietary closed-source SDKs for purposes such as advertisements, telemetry, or ecosystem integration [265]. On both Android and iOS, apps can integrate libraries in the platform's preferred Java/Kotlin or Objective-C/Swift languages, and also libraries originating from C or C++ code. On both platforms, developers take this route to reuse existing C/C++ code bases or implement performance-critical functionality. Additionally, applications on both platforms include a metadata file or application manifest. This file encodes various runtime properties of the app, including the functionality it offers the system and the system capabilities it consumes.

Security researchers often need to understand the implementation details of compiled apps, which typically involves reverse engineering. For Android apps, most logic is implemented in Dalvik bytecode, which retains much high-level information that facilitates decompilation. Compiled iOS binaries retain less information, but often still contain helpful Objective-C or Swift metadata. Native libraries are typically hardest to reverse-engineer, since they might be stripped of helpful symbols and heavily obfuscated.

2.2. Security Paradigms of Mobile Platforms

While mobile platforms share many similarities with traditional computing platforms, there are key differences in usage patterns and operating environments. These differences also create unique security challenges that mobile platforms must address.

2.2.1. Operational Constraints and Challenges

Portability. In contrast to a laptop or desktop computer, a smartphone has a physical footprint that is small enough for the owner to carry it wherever they go. This means that the device can easily be lost or stolen,

rendering physical attacks highly likely. It also means that data produced by the various on-device sensors could be used for physical activity or location tracking.

Connectivity. Mobile devices typically include more communication modules than most traditional computing devices. Not only are they constantly connected to the cellular network, they also routinely communicate with other devices in their surroundings. Mobile platforms also commonly integrate unique dual-role support for interfaces such as USB or Wi-Fi, or implement custom solutions on top of established protocols. All of these communication modes are potential entry points for attackers.

Data Sensitivity. As a result of mobile devices' portability and connectivity, most users accumulate large amounts of sensitive data on them. Among others, this includes photos or videos, communication records such as personal messages, and credentials used to log in to websites while on the go. All of this data represents valuable assets to an attacker.

Usability. Smartphones are typically used on the move, often during brief interruptions while users remain engaged in other activities. To accommodate this usage pattern, mobile platforms place particular emphasis on usability, ensuring that common tasks can be performed single-handedly with minimal interaction effort. In addition, smartphones provide low-cost access to computing for a broad population, requiring them to be approachable even for individuals with little or no prior computing experience. Consequently, achieving an appropriate balance between usability and security represents a particularly significant challenge for mobile platforms, more so than for traditional computing systems.

Efficiency. Many users depend on their smartphone as a communicator, payment method, or authentication factor. It is therefore essential that the device remains operational for as long as possible despite limited battery capacity. Balancing functionality, security, and energy efficiency, therefore, represents a fundamental design challenge for smartphones, one that is considerably less pronounced in traditional computing platforms with continuous power availability.

2.2.2. Security Architecture and Defense Mechanisms

In response to these and more operational constraints and challenges, mobile platforms have pioneered novel security architectures and defensive

mechanisms. While many of them remain unique to mobile platforms, others have found their way into traditional computing platforms over time.

2.2.2.1. Sandboxing

Executing third-party code on a device full of sensitive data inevitably poses a risk. To reduce this risk, Android and iOS run applications in their own isolated containers [11, 31]. The isolation goes beyond traditional process isolation, and also includes discretionary access control (DAC) and mandatory access control (MAC). Android additionally creates distinct Linux users for each app. Both platforms are also very restrictive regarding the lifecycle of applications. By default, apps can only be executed while they are in the foreground, i.e., displaying a user interface on the screen. While some possibilities exist for an app to remain active in the background, these are intended for performing a number of pre-defined tasks such as playing music, navigating, exchanging data with accessories, and others [12, 23]. Beyond the lifecycle, the sandbox also restricts an app's interactions with the file system, with the kernel, and with other apps.

On both Android and iOS, freshly installed applications by default only get file system access to a private data folder inaccessible to other apps. Across platforms, reading other apps' data is only possible for certain public locations and requires explicit user consent. Android uses Filesystem in Userspace (FUSE) to further limit file system access beyond DAC and MAC [14]. This allows, e.g., restricting an app's view of a public folder to just those files it created.

On iOS, the possibilities for apps to communicate with each other are very limited. Although low-level primitives such as sockets exist, they offer limited use for inter-app communication due to the app lifecycle described above. Communication would only work while both involved apps are active, which is unlikely. The only way for iOS apps to directly and reliably communicate with other apps is through custom URI schemes [25]. While this mechanism moves invoked apps into an active state if needed, it can only be used for text-based data. Other IPC mechanisms based on Apple's XPC framework exist, but they require explicit user consent and only allow communication with isolated app parts running separately from the main app [19]. On Android, there are multiple communication paradigms that allow an app to directly invoke components in other apps and execute

a certain action. Most of these paradigms are based on high-level wrappers of Android's Binder IPC mechanism, such as Intents and Intent Filters [15]. They allow passing complex data or even file descriptors, and move the invoked app into an active state if needed.

The app sandboxes on Android and iOS also tightly restrict the system calls available to user-installed applications. For many interactions with the kernel or hardware, apps must call higher-level APIs that act as proxies for more privileged platform daemons. Both platforms offer opportunities for apps to integrate into the system by fulfilling predefined roles. For example, both platforms allow apps to act as system-wide network gateways, on-screen keyboards, password managers, accessibility tools, data sources, and more. For all these cases, apps need to use clearly defined APIs, and all communication is routed via the system. Additionally, explicit user consent is strictly necessary.

2.2.2.2. Permissions Systems

As the core security philosophy, the Android Platform Security Model states that most key actions need authorization from all three main stakeholders [180]. These are the platform, the end user, and the app (or its developer). While Apple does not publish a detailed security model for iOS, many similarities can be observed. In general, the platform encodes its authorization by limiting accessible APIs to only actions intended for apps. The app developer likewise encodes their authorization through implementing corresponding functionality and (if needed) declaring it to the rest of the system through the app's manifest. Finally, both Android and iOS implement permission systems that ultimately put the user in control over the most critical actions. For example, permissions decide whether apps may access the camera, the contacts store, or files on the device.

The permission system on Android requires applications to statically declare the required permissions, each identified through a unique name, in their app manifest [17]. On iOS, a static usage declaration is only necessary for some permissions, and the exact format varies across permissions. For most permissions, it is a description string displayed to the user at runtime when the app accesses protected APIs in the corresponding category [30]. Statically declaring permissions in the app manifest permits auditability. It allows end users to estimate the privacy impact of an app,

and security researchers to identify potential attack vectors. While, in general, a permission protects a certain set of APIs, no complete official mapping exists on either platform.

On iOS, the platform generally displays a user prompt for any permission-protected API. This user prompt is shown the first time the app calls a relevant API, and allows the user to either grant or deny access to the corresponding system resource. For subsequent API invocations, the system saves the user's decision and allows later adjustments through the system settings. Android's permission system is more nuanced. Each system-defined permission is assigned a protection level. For normal-level permissions, the user implicitly grants access through app installation, which can only be revoked through uninstalling the app. Dangerous-level permissions work much like permissions on iOS, except that the app must explicitly trigger the user prompt. Additionally, the authorization is more coarse-grained, with permissions bundled into categories and prompts shown only once per category.

As a key difference to iOS, Android allows third-party apps to extend the system's permission system. They can define custom permission names, which they can use to restrict access to their exposed components. For example, an app can ensure that its components are accessible only to other apps signed by the same developer. Finally, apps can also protect their components using permissions defined by other apps or by the system. The platform's API provides functionality (backed by the Binder IPC system) to check at runtime whether another process has a specific permission.

2.2.2.3. Tight Ecosystem Control

Apple's tight control over the iOS ecosystem is evident in its role as the platform's exclusive device manufacturer. While Android started as a much more open ecosystem, Google has increasingly tightened its grip on the platform over the last few years.

On both platforms, developers have always needed to sign apps before they could be installed on end-user devices. However, the role this signature plays in the security architecture of the two platforms differs significantly. On iOS, every code instruction executed needs to be either directly or indirectly (through Apple-issued provisioning profiles) signed by Apple [20]. This gives Apple a chance to vet any code executed on their devices, or at least the corresponding developer. While a small number of exceptions

are possible, these are limited to development devices or software written by Apple. This approach therefore impedes certain use cases such as Just-In-Time (JIT) compilers for third-party JavaScript engines. Android uses package signatures typically created from a self-signed certificate generated by the developer [10]. The signature is checked only during installation, and APIs exist for dynamically loading and executing additional code from arbitrary sources. The signature's main purpose is protecting the integrity of app updates, i.e., ensuring that updates originate from the same developer as the original installation.

The undisputed primary distribution channels for Android and iOS apps are Google Play and Apple's App Store, respectively. Some Android vendors maintain their own separate app stores, such as Samsung's Galaxy Store [223], Xiaomi's GetApps [285], or Huawei's AppGallery [122]. While these vendor-specific stores typically augment Google's offerings, they remain the only option on devices from vendors without a GMS license, such as Huawei. On Google Play and Apple's App Store, apps have to undergo a rigorous review process that includes manual and automated checks of the app, its resources, and marketing materials. The goal is to rule out malicious behavior and eliminate violations against platform guidelines, legal regulations, or security recommendations. Android has supported installing apps from arbitrary sources (sideloading) and third-party app stores since its inception, and iOS was forced to follow suit by the EU's Digital Markets Act in 2022 [27]. However, these options have considerable limitations on recent versions of both platforms. On iOS, apps distributed through third-party app stores or websites are still subject to a notarization process and commission paid to Apple [28]. Android is increasingly making it more difficult for users to install apps from independent app stores or websites, citing security concerns [248]. In 2025, Google announced that all developers, regardless of distribution channel, would have to register their apps with Google in the near future.

Because of their central positions in the respective platforms, Apple and Google have ultimate control over apps in their ecosystems. On Android, Google Play is very selective about apps allowed to use certain permissions. For example, only those apps are allowed to request the permission for full file system access that Google deems requiring it for their legitimate operation [107]. While developers can distribute apps that use these permissions through other channels, installing apps from unknown sources is growing increasingly difficult, as described above. Effectively, this means that platform authorization for key actions as described in the Android

Platform Security Model is split between Google and the device vendor. On iOS, applications require so-called entitlements to access certain system resources [30]. While many of these can be used in any app and align with user-facing permissions, some entitlements require Apple's authorization on a per-app basis. Entitlements are encoded in an app's code signature (or provisioning profile), which cements Apple's exclusive control over them already at the technical level.

2.2.2.4. Hardware-based Security and Trust

Both Android and iOS use hardware-based trust as a key foundation for their security architectures. It ensures the integrity of the operating system and the confidentiality of the user's data.

Both platforms implement secure boot procedures intended to prevent tampering and corruption [18, 22]. The first code executed when the device is powered up is the boot ROM, an immutable, low-level firmware embedded in the SoC's read-only memory. It includes the public part of a cryptographic key pair and proceeds with booting only if the subsequent bootloader stage has been signed with the corresponding private key. The key pair, or more specifically, its public part, therefore serves as a root of trust. All the following components in the boot chain in turn verify the integrity and authenticity of the next stage before executing it. Finally, this procedure reaches the kernel, which provides foundational security infrastructure for userspace, such as DAC and MAC. Critical coprocessors, such as the Secure Enclave and the baseband chip, follow their own secure boot procedures, triggered by the main CPU. Some Android devices allow turning off secure boot, and even setting a custom root of trust, which facilitates security research but may lower device security [16].

Modern versions of Android and iOS by default encrypt files on disk [13, 24]. The most sensitive files are encrypted with individual keys protected by master keys derived from or wrapped with a unique hardware-bound key and the user's passcode. Encryption is typically implemented via the Trusted Execution Environment (TEE) on Android devices and the Secure Enclave Processor (SEP) on iOS. Overall, the used Data Protection (iOS) or File-Based Encryption (Android) schemes effectively prevent physical attackers from extracting data from a shut-down device. Dependence on the hardware-bound key means any brute-forcing attempt on the user's passcode must occur in-situ, where attempts can be strictly limited.

On iOS and on some Android implementations, a crypto engine in the storage controller is employed, so that even individual file keys cannot be leaked through a compromised kernel. One key difference between iOS and Android needs to be noted. While iOS discards master keys on every device lock, Android only does so on reboots. As a result, Android's file encryption is only effective in the before-first-unlock (BFU) state.

Android and iOS also offer hardware-based key storage [9, 32]. Apps can request the generation of key pairs whose private component never leaves the HSM. Cryptographic operations using the key in the HSM can be requested, allowing, e.g., solving authentication challenges with backend servers. Keys can be tied to specific requirements, such as authorizing each individual use via biometric authentication or restricting use to specific cryptographic operations. The iOS implementation relies on the platform's SEP, while Android variants include proprietary or Google-supplied solutions based on the TEE or a dedicated StrongBox secure element. Android allows requesting key attestations that prove a key's unexportable residency in the HSM, as well as the integrity of the OS and the signature of the running application. Backend servers can use these attestations to verify they are communicating with an untampered client environment. Similar attestations can be obtained on iOS as well, but focus on the OS and app integrity.

2.2.2.5. No root-level access

Neither Android nor iOS provides end users with root-level access on intact installs. This prevents users from inadvertently subverting the platforms' elaborate security models. Still, both platforms allow gaining some degree of elevated access for development or research purposes.

On Android, the system can be development-enabled to unlock shell access via USB or TCP/IP [8]. This process entails a setup procedure in the system settings that also requires user authentication. Finally, the Android Debug Bridge (ADB) daemon can be enabled on the device, which allows access to a shell running under its own Linux user. It offers commands to inspect many parts of the file system and to manage device aspects such as installed apps. A development mode is also available on iOS, but does not grant shell access [26]. Instead, it permits installing and debugging development builds of apps.

While root-level access is not generally allowed on Android and iOS, there are ways to lift this restriction. Some Android devices allow unlocking the bootloader, and installing arbitrary firmware builds [16]. Researchers can take advantage of this possibility to add root access to stock firmware images. On devices that do not allow bootloader unlocking, root access can only be obtained through exploit chains used by attackers. Even userspace processes running as root are subject to strict SELinux rules, so that full system access requires compromising the kernel. On iOS, bootloader unlocking is not possible, which means that exploit chains remain the only choice for gaining root-level access on production devices. However, modern iOS devices and OS versions integrate countless security mechanisms that render complete exploit chains highly valuable and therefore inaccessible to most white hat researchers. Apple therefore runs a Security Research Device (SRD) program that offers modified iOS devices with root-level access to the research community [21].

3

State of the Art

In this chapter, we first present a high-level overview of the state of the art in mobile security research. Then, we delve into more details in the three areas of Communication Interfaces, Supply Chains, and API Misuse, which this thesis contributes to.

3.1. Overview of Mobile Security Research

In the following, we organize the current state of the art in mobile security across the three stack layers applications, middleware, and hardware/kernel. Research on mobile applications can broadly be categorized into two areas: malicious apps and vulnerable apps.

Malicious Apps. Within the field of malicious apps, a major research direction in recent years has been Android malware, with a strong focus on its detection. The currently most capable approaches for detecting malware in large app sets use machine learning based on features statically extracted from code and resources in the application package [90]. However, such approaches are vulnerable to evasion attacks, such as adversarial examples [110, 160, 161], and suffer from concept drift due to the fast-paced evolution of both benign and malicious apps. Recent works, therefore, suggested various methods for bypassing these limitations [55, 253, 304]. While not as scalable, alternative dynamic approaches help in analyzing highly obfuscated malware or apps whose malicious intent is not apparent just from their packages [162, 290]. Finally, some state-of-the-art literature focused on the post-compromise phase, e.g., victim discovery [288] and takedown [301] for specific malware types. While not as decidedly malicious as malware itself, some apps are overly intrusive on user privacy or device resources, sometimes even stealthily so. Recent research has investigated this phenomenon from multiple angles, including detection at app-global

scale through static and dynamic analysis [235, 296, 297], and analysis of specific app functionality patterns [64, 163, 279]. Some intrusive behavior was shown to stem from integrated third-party libraries, which constitutes an aspect of supply chain security discussed in detail in Section 3.3.

Vulnerable Apps. Vulnerabilities in apps put user data at risk, so an active research field has emerged around this topic. Several recent works have been devoted to threats in mini-app ecosystems, which are formed by apps such as WeChat that offer app-in-app experiences. This design bypasses the operating system's isolation between applications. Several resulting vulnerabilities have been scientifically explored, including leakage of user or app secrets between mini-apps [45, 236, 306], or manipulation of cross-mini-app communication [295, 310]. Another recent thrust of research has explored more general weakness patterns in mobile apps. Several works uncovered shortcomings in custom encryption schemes that apps use in network communication or data storage [145, 211, 271]. Similarly lacking protection schemes were also recently found for machine learning models embedded in mobile apps [63, 247]. One technique explored for uncovering app vulnerabilities is fuzzing, both on native app parts [65], as well as on apps' IPC endpoints [34, 131]. We discuss API misuse, another common source for vulnerabilities in apps, in detail in Section 3.4.

Permission Systems. A major research area on the middleware level of the mobile platform stack focuses on permission systems. One of the concrete goals here is uncovering permission inconsistencies, i.e., resources accessible through multiple APIs with incoherent permission requirements. Recent works presented novel approaches for more precise inconsistency detection across different framework endpoints [61, 219] or focused on inconsistencies across specific API layers, such as JNI [313] or system broadcasts [312]. Permission inconsistencies are often caused by vendor customizations to the OS, which motivated research that approaches the problem entirely from this angle [261, 262]. Several research groups have analyzed inconsistencies in specific permission domains on both Android and iOS, such as the file system [155, 157], networking [227], or user tracking [182, 187]. The Android permission system has also undergone recent scrutiny from a usability perspective, exploring misconceptions and decision rationales [48, 77, 233].

Middleware Vulnerabilities and Exploits. A number of contributions have been made over the past five years towards the discovery of middleware vulnerabilities, their exploitation through malicious apps, and possible mitigations. Several recent publications focused on UI-based flaws

and attacks on Android, such as accessibility services that extract user secrets from app UIs [120, 156, 289] or UI overlays that, e.g., trick users into granting permissions [40, 272, 314]. The latter can be complemented with state inference attacks for determining the presence and state of victim apps [166, 210]. Further works have shown exploitable vulnerabilities in Android's IPC infrastructure that allow for denial-of-service [300] attacks and privilege escalations [47, 178]. Another subsystem with recently published vulnerabilities is the file system, in particular Android's external storage, which can be exploited to bypass restrictions [67] and attack other processes [73, 108]. Several recent studies have investigated the effects of vendor customizations on the platform's security posture. Despite approaching from different angles, ranging from hardcoded access-control exceptions to update mechanisms and contents, they found that vendor customizations violate Google's guidelines [43, 209] and negatively affect security [132, 264], not least because updates are delivered less frequently [2].

App Stores. Application stores are essential parts of the middleware layer of the mobile platform stack and have been the subject of several recent studies. In particular, many of these studies investigated the accuracy of Google's Data Safety Section and Apple's App Privacy Labels. While intended to offer users a faithful representation of an app's privacy and security impact, these indicators rely on developer-supplied data. As a result, developers on both platforms were found to frequently misrepresent their apps, in particular with regard to data collection practices [33, 282]. Orthogonal studies suggested that inconsistencies might be caused by intransparent third-party libraries [139, 283]. Although two recent publications described app stores, particularly Google Play, as a potential distribution vector for malware and other unwanted apps [147, 172], their supply chain aspects remain largely underexplored. We provide a detailed discussion of our contributions to analyzing app stores from a supply chain security perspective in Section 3.3.

Hardware Side Channels and Injection Attacks. A major research direction at the hardware/kernel layer of the mobile platform stack are attacks that exploit physical or architectural hardware characteristics. For data extraction through side channels, attacks typically train machine learning models on correlations between emanated signals and specific device activity, so that later measurements on victim devices allow deduction in the opposite direction. Several recent works have focused on side channels based on electromagnetic phenomena emanating from the electronics

in Android and iOS devices or connected peripherals. They demonstrated that attackers can infer screen [171] or audio output [51], touch input [134], app activity [193], and even user fingerprints [195] through probes in proximity to the victim. Additional studies exploited power fluctuations measured from wired or wireless chargers to determine screen content changes [60], browsed websites [152], or user interactions [194]. Some hardware-based side channels have also been shown to be measurable by local adversaries, including leakage caused by chip implementation properties [141, 263], and general electrical or physical side-effects quantifiable through on-device sensors [66, 197, 246, 251]. Researchers have also used physical phenomena to manipulate mobile devices, such as injecting touch events [133, 231, 270] or voice commands [274, 298] via electromagnetic interference or power-line noise. We extend some of the state-of-the-art side-channel results in our paper ChoiceJacking [71].

Trusted Execution Environments. Research on the security of and potential uses for trusted execution environments such as ARM TrustZone is an active research area in its own right, largely independent of mobile platforms. Still, recent overlapping works investigating aspects specific to Android and iOS can be found. These include security assessments of the proprietary TEE-based fingerprint authentication subsystems on Android and iOS [56] and vendor-specific implementations for hardware-bound keystores [230] or trusted app roll-back prevention on Android [44]. We use ARM TrustZone in a mobile-specific setting in the paper Don't Stop Receiving [255], which the thesis author co-authored.

Kernel Vulnerabilities and Hardening. The Linux and XNU kernels serve as the software foundation for Android and iOS, making them highly interesting to security researchers. In fact, the Linux kernel, due to its use in countless other platforms, is already subject to an entire independent research field examining vulnerabilities and mitigations. Still, numerous works on both the Linux and XNU kernels specifically approach the topic from a mobile platform security angle. Recent publications in this direction can be found for both platforms, uncovering mobile-specific kernel-level vulnerabilities [135, 311] and side-channels that allow bypassing sandbox restrictions [166, 276]. Some scientists also work towards mobile-specific kernel hardening, such as extending SELinux to isolate functionality modules of individual Android apps [221]. A particular focus in recent years has been large-scale surveys of vendor-specific Android kernel security posture and vulnerability patch behavior [2, 144, 307]. As a collaborator

of Lukas Maar, the author of this thesis contributed to two works in this domain, Defects-In-Depth [175] and Doom of Device Drivers [176].

3.2. Secure Communication Interfaces on Mobile Platforms

Modern mobile devices are equipped not only with a cellular modem but also with a multitude of communication interfaces for protocols such as Wi-Fi, Bluetooth, and USB. These communication standards are important pieces of today's interconnected world and are also used in products unrelated to Android or iOS. Therefore, they are the subjects of active research fields independent of mobile platform security. In the following, we focus our description of the state of the art on studies with a clear relation to Android or iOS. These are either works dedicated to vendor-specific implementations of these standardized protocols on Android or iOS devices, or works dedicated to closed-source, proprietary communication solutions from a specific smartphone vendor.

Vendor-specific implementations of standardized protocols. Although most Android and iOS devices are very similar in the standardized communication protocols they support, their implementations often differ considerably. This is because even two Android devices might utilize interface controllers from different vendors, resulting in divergence in hardware functionality and low-level firmware or driver implementations. Given that our contributions in this area concern the USB stack of mobile devices, we keep our descriptions for other interfaces short.

Of all the communication interfaces embedded in mobile devices, the cellular modem is commonly considered the most complex, owing not least to the sheer number of involved subprotocols. Much of the functionality for dealing with these protocols is implemented in the baseband chip and its firmware. Any vulnerability in the network-facing attack surface of these components could allow an attacker to compromise the device's communication, possibly even the entire device. Numerous recent works have therefore analyzed the security of proprietary closed-source baseband implementations on both Android and iOS [116, 117, 124, 140, 164, 217], uncovering exploitable logic and memory flaws. While most publications on security aspects of Wi-Fi examined design vulnerabilities of the involved protection schemes [256, 257, 258, 259], exploits are commonly

exacerbated by implementation flaws in Android or iOS. For identifying implementation flaws, multiple teams have employed fuzzing on wireless device drivers and firmware for both Wi-Fi and Bluetooth [46, 125, 222]. Further publications have uncovered Man-in-the-Middle [287, 305] and device impersonation [240] attacks, connection leakage across apps [240], and tracking possibilities [280] in Android and iOS implementations of Bluetooth and Bluetooth Low Energy (BLE).

Various works have examined the security of USB on Android and iOS. In 2011, Markus et al. [149] presented an attack termed Juice Jacking, which embedded a computer in a public phone charger to stealthily extract files or install malware while charging. The attack was made possible as mobile phones use a single physical port for both charging and file exchange, and, at the time, blindly trusted any USB connection the user established. In 2013, Lau et al. [153] described in detail how a Juice Jacking attack can install malware. As a mitigation to these threats, both Android and iOS introduced the requirement for explicit user consent to USB file exchange with a computer. This mitigation was widely believed to be effective, and subsequent USB-based attacks focused on less powerful attack vectors.

One of these attack vectors is embedding a USB peripheral, such as a USB keyboard or audio/video interface, in the malicious charger. Nohl and Lell [196] used this method to inject input events, Meng et al. [183, 184] extracted screen contents, and Wang et al. [274] injected voice commands. The data extraction capabilities of these attacks are limited. In concurrent research, Pereira et al. [206], Tian et al. [252], and Imtiaz et al. [137] investigated the AT interfaces that some Android devices expose for communicating with the baseband processor via USB. While the AT protocol was standardized by ETSI for cellular networks, the specific commands Android phones expose over USB depend on the baseband chipset manufacturer. Tian et al. [252] reported that 6 out of 14 devices tested in 2019 exposed such AT interfaces. In accordance with Pereira et al. [206], they found commands for firmware flashing or extraction of user files via USB. Imtiaz et al. [137] identified vulnerabilities in AT command parsing that allowed denial-of-service attacks or extraction of sensitive device identifiers. Despite the impact of the uncovered attacks on AT interfaces, they affected only a limited number of Android devices [252]. Kim et al. presented fuzzers for uncovering issues in USB gadget [143] and USB Power Delivery [142] (PD) stacks on mobile devices and other platforms. They uncovered previously unknown memory-safety issues in Android gadget drivers and specification violations in the USB PD power

negotiation procedures of Android and iOS devices. It is worth noting that these vulnerabilities were again device-specific and therefore hard to exploit at scale. In summary, there has been a lot of research in recent years, but no attacks had an impact comparable to JuiceJacking. This indicates a wide-spread belief that the implemented mitigations were effective. In the first contribution of this thesis, we show that this is not the case.

Contribution I

Draschbacher et al. [71] demonstrate that the user consent requirements introduced in Android and iOS are ineffective in preventing JuiceJacking-style attacks. They present ChoiceJacking, a novel family of hybrid USB attacks that combine characteristics of host-based and device-based attacks. Through three different techniques, the attacks allow a malicious charger to spoof user consent and extract user files autonomously. As the attacks are rooted in trust model flaws, they affect all evaluated state-of-the-art Android or iOS devices.

Closed-source proprietary communication solutions. As part of their efforts to differentiate through customization, all major smartphone vendors have implemented proprietary extensions on top of standardized communication interfaces. Academic research on the security of these implementations remains relatively sparse. Curiously, most of the limited works that do exist focused on iOS.

Several research teams analyzed the protocols involved in Apple's Continuity, which allows seamless transfer of settings and activities between macOS and iOS devices. In 2019, Stute et al. [245] reverse-engineered Apple's proprietary Apple Wireless Direct Link (AWDL) protocol for Wi-Fi-based peer-to-peer communication. Among other findings, they presented a denial-of-service vulnerability that an attacker can exploit to eventually gain a Man-in-the-Middle position in AirDrop file exchanges. In concurrent work, Martin et al. [179] reverse-engineered Continuity subprotocols based on Bluetooth Low Energy (BLE). They showed that devices constantly broadcast BLE messages for various UI interactions, which allows user profiling and tracking from proximity. A follow-up paper by Celosia and Cunche [49] in 2020 highlighted further design and implementation flaws in the Continuity protocol and its various applications. These included additional possibilities for user tracking and device fingerprinting, as well as leakage of user email addresses, phone numbers, and executed voice commands. A very similar leakage of email addresses

and phone numbers was later also reported by Heinrich et al. [113] in AirDrop authentication messages. In their 2021 paper, Stute et al. [244] further reverse-engineered Continuity features and demonstrated that an attacker can gain a network Man-in-the-Middle position by exploiting Continuity-based Wi-Fi Password Sharing.

Several publications have investigated security and privacy aspects of Apple's "Find My" network and the underlying BLE tracking technology based on asymmetric cryptography. Every tracked offline device broadcasts its public key to surrounding observers, who report the location to Apple servers, encrypted using the received public key. Users can then query Apple's database for reports of their devices and use the corresponding private key to decrypt the location report. Heinrich et al. [114] in 2021 reverse-engineered Apple's closed-source implementation and highlighted potential for denial-of-service and location-correlation attacks. In a follow-up publication, Heinrich et al. [112] also presented a stalking detector that checks whether any device in the "Find My" network closely follows an Android device. Apple already integrates similar functionality into iOS. In 2025, Chen et al. [53] showed that it is possible to misuse the "Find My" network to track any BLE device through a combination of malicious software running as non-root user and clever key-pair computation. Their approach eliminated the need for a device to overwrite its advertising address to match the public key, which requires root access.

Finally, two recent publications have reverse-engineered Apple-proprietary protocols that so far have remained entirely unexplored. Classen et al. [57] scrutinized Apple's satellite communication protocol, revealing ways for a root-level attacker to bypass Apple's regional restrictions or send arbitrary messages. Rollhausen et al. [220] documented the wireless protocols of the Apple Watch. While they identified protocol flaws that allow redirecting encrypted traffic or forging health data, the required attacker models assume that parts of the layered protocol stack have already been compromised.

Only a small number of contributions have been made towards security analyses of closed-source, proprietary communication implementations on Android. In 2019, Antonioli et al. [7] reverse-engineered Google's Nearby Connections framework, which third-party applications can build upon for proximity-based communication. Even if the framework offers a public API, its closed-source implementation is embedded in the heavily obfuscated Google Play Services. Antionioli et al. [7] documented that Nearby Connections uses a combination of Wi-Fi and Bluetooth for device

discovery and payload exchange. Additionally, they identified multiple attacks, including the possibility of interception of decrypted traffic. One of the applications that use the Nearby Connections framework is Google's *QuickShare* file sharing tool, formerly called *Nearby Share* before a merger with Samsung's similar technology. In 2024, researchers from security firm SafeBreach reverse-engineered its full protocol and identified multiple vulnerabilities, which they termed *QuickShell* [58]. While the eponymous attack allowed code execution only on the tool's Windows version, the researchers also found a bypass for file accept user confirmations, effectively allowing unauthorized file write on the device. Google in 2025 hired security assessors from NetSPI to conduct penetration testing on a later version of QuickShare, which, according to the public report, did not yield any noteworthy findings [36]. Also in 2025, Wang et al. [275] examined the first-party file sharing tools from Android device manufacturers Huawei and Xiaomi and reported that the latter is susceptible to stealthy Man-in-the-Middle attacks.

Already in 2019, Ma et al. [173] analyzed data migration tools offered by 7 smartphone vendors. These tools facilitate setting up a newly purchased smartphone by copying all data from the old device, typically through a wireless peer-to-peer connection. In their study, the authors demonstrated wireless data eavesdropping on four tools due to insecure link-level security and a lack of higher-level encryption. Since then, no other researchers have carried out security evaluations of the data migration tool landscape in the Android ecosystem. This is despite the landscape's significant evolution over the past years. For example, Google used to recommend data migrations via wired connections [101], but removed any hints at this option when rebranding their tool as *Android Switch* in 2024 [106]. These developments motivated a renewed evaluation of the security posture of Android data migration tools offered by device manufacturers.

Contribution II

Draschbacher et al. [72] present the first systematic analysis of current-generation data migration tools on Android. They go beyond Ma's 2019 study et al. [173] methodologically, by grounding their evaluations on a set of generic attack scenarios. Additionally, they are the first to scrutinize Samsung's Smart Switch and Google's Android Switch tools, the current market leaders. Draschbacher et al. demonstrate proximal attacks against all 7 evaluated tools (Google, Samsung, Xiaomi, Vivo, Oppo, Huawei, Honor), ranging from data extraction to code execution.

3.3. Supply Chain Security on Mobile Platforms

A product's supply chain consists of the various stages it undergoes from the design of its primitive components to its delivery to end users. When a product or an individual part is compromised anywhere along this chain, the term *supply chain attack* can be used. It is worth noting that such compromise can occur due to an inside attacker, i.e., a malicious party working at some station of the supply chain, or due to the malicious party gaining unauthorized access, i.e., breaching the supply chain itself. At least three distinct supply chains exist on mobile platforms. One is the hardware supply chain, where the sheer amount of individual components in a smartphone and the complexity of their design and fabrication offer ample potential for compromise. For example, authentication keys that allowed interception and impersonation on the cellular network have in the past been stolen from SIM card manufacturers [225]. The firmware supply chain revolves around the device's operating system, where attacks or compromises can happen anywhere from suppliers of board support package components or preinstalled apps to the delivery infrastructure of over-the-air updates. Multiple vulnerabilities in this supply chain have been documented in the past that were caused by leaked platform [237], system module [111] or Android Verified Boot signing keys [158]. Finally, the application supply chain spans the process from assembling code components into a compiled application and shipping it to end users. Since our contributions in this field concern the latter of the three, it is also the focus of our state-of-the-art elaboration in this section. At a high level, the application supply chain consists of three possible entry points for attackers. These are third-party libraries, build tools, and app distribution.

Third-party libraries. Modern mobile applications are no longer monolithic software projects. Instead, they commonly incorporate closed-source third-party libraries for tasks such as advertisements, analytics, or social integration. These libraries run in the same process space as the main application logic, which means that they can interact with app-internal data and piggyback on the permissions the user granted to the app for its legitimate functionality. Researchers have long identified this as a security and privacy threat. More than 15 years ago, Egele et al. [76] and Grace et al. [109] found that over half of the analyzed apps on both Android and iOS included advertisement or tracking libraries. On iOS, they leaked identifiers allowing cross-app user tracking, while on Android, they even accessed the user's SMS messages. Pearce et al. [205] reported that almost

half of the analyzed apps that integrated ad libraries requested privacy-sensitive permissions just for them. The longitudinal analysis conducted by Wang et al. [265] indicated that the number of apps monetized through advertisements has continued to increase since then. More recent studies found multifaceted privacy-intrusive or outright malicious behaviors of third-party SDKs embedded in Android applications. Zhang et al. [309] studied third-party libraries that exhibit potentially malicious behaviors such as accessing location, contacts, calendar, storage, or body sensors. Wang et al. [273] uncovered threats emerging from malicious libraries that override resources from other app components. Mi et al. [186] identified SDKs that monetize apps by renting the device's Internet connection to a proxy network. Wang et al. [268] documented predatory SDKs that harvest sensitive data from other SDKs in the same app process, a phenomenon later also confirmed on iOS by Liu et al. [168].

For Android in particular, a long line of research has proposed solutions to alleviate these problems, most of which introduce a separation between the main application logic and third-party libraries. Shekhar et al. [232], Pearce et al. [205], Zhang et al. [302], and others suggested full-on compartmentalization for advertisement libraries, i.e., executing them in isolated processes and only granting them a minimal subset of the app's permissions. All of these solutions required some modifications to the Android framework and applications. Shekhar et al. [232] offered an automated tool for retrofitting necessary changes into compiled apps. Huang et al. [121] proposed a solution that works entirely compiler-based, through a modified version of the on-device *dex2oat* ahead-of-time compiler that moves third-party advertisement libraries into separate processes while translating Dalvik bytecode to native instructions. While strict process isolation can work for ad libraries due to the limited interaction between the core app and third-party code, it is more difficult to achieve for a more general model of third-party libraries. Seo et al. [229], Dawoud and Bugiel [62], and Rossi et al. [221] suggested letting app developers specify a policy of what app parts should be allowed access to what resources, and enforcing it at runtime, either at subprocess-granularity or based on call stacks.

It is worth noting that any such solution can only realistically be deployed on end-user devices if it is integrated into the upstream AOSP project. Otherwise, it would only exacerbate the Android platform's already existing fragmentation problems. Potentially out of this consideration, Google in 2022 announced plans to introduce process-based isolation for ad libraries in the official app framework [59]. Their *SDK Runtime* [99], introduced as

part of the larger Android Privacy Sandbox initiative, would have tightly limited the permissions available to SDK processes and entirely disabled runtime features such as native code for them. Advertisement libraries would have been updated independently of applications, additionally solving the problem of outdated dependencies that cause vulnerabilities in apps. Although Google published a working prototype with Android 14 in 2023 [99], they eventually abandoned the project in 2025, citing feedback from marketers and publishers, as well as low adoption [50].

Build tools and dependency managers. Although cases of compromised build tools have been documented, academic research on the topic remains relatively sparse. In 2015, researchers found that threat actors had published a manipulated version of Apple's Xcode development environment [281]. Any app compiled using this tainted version included malware that sent sensitive user data to the attackers. According to Apple's internal estimates, more than 2500 apps had been infected, impacting 128 million users [87]. In their 2020 study, Zhang et al. [309] found hints at Android apps containing modified versions of third-party libraries. While they did not investigate the causes, compromises in the build chain are one possibility. More recently, EVA Information Security uncovered flaws in the CocoaPods dependency manager commonly used in iOS app development [218]. These flaws allowed attackers to manipulate packages in a CocoaPods repository, gaining code execution capabilities in any dependent app. Schmidt et al. [228] also examined iOS dependency management solutions, and found further potential for compromise in CocoaPods.

App distribution. While some research exists on the prevalence of malware in app stores, the potential for legitimate apps to be compromised during distribution remains underexplored. Fahl et al. [80] presented an important first step in this direction, analyzing app signing practices on Google Play and proposing an Application Transparency framework derived from Certificate Transparency for TLS certificates. Kotzias et al. [147] and Shen et al. [234] found that malicious parties routinely used Google Play to disseminate potentially harmful apps, where they survived for more than two months on average before being detected and removed. The results by Wang et al. [266] suggested that about 3 % of all apps removed from Google Play were malicious. In a follow-up study, Wang et al. [265] reported a clear downward trend in malware prevalence on Google Play over a period of 3 years. Wang et al. [267] also compared various aspects of Android app stores, and found that third-party app stores performed about twice as poorly as Google Play in the best case in terms of malware

prevalence and removal. Yang et al. [293] even uncovered a tendency for identical apps to be more likely to contain malicious code when distributed through third-party app stores than when distributed through Google Play. Hu et al. [119] showed that malware authors commonly masqueraded their apps as popular apps, a technique termed app squatting. Given the lack of truly independent third-party iOS app stores, most of these publications focused on Android. Liu et al. [170] recently investigated unofficial iOS app stores that misuse procedures intended for app testing. They found that the majority of the catalogs consisted of manipulated versions of popular apps that might also introduce malicious code.

In light of recent developments and ongoing discourse on ecosystem monopolies and digital sovereignty, the position of established app stores as central supply chain gatekeepers warrants critical scrutiny. When Google introduced the Android Application Bundle (AAB) format and, in 2020, announced plans to require it for app distribution through Google Play, concern was raised only by developer advocates [190]. The criticism pointed at the potential for app stores to stealthily manipulate apps under the new distribution scheme. This potential arises because the AAB format requires developers to hand over signing keys to app store operators. An app's intact signature created from the authentic (when compared to a previous app version) certificate is therefore no longer indicative of its integrity. Handing over signing keys had been necessary for Google Play Signing before, which offered server-side app signing as a convenience feature for developers. While, as pointed out by Lins et al. [167], Play Signing had therefore allowed for stealthy compromise as well, it was never mandatory. In 2021, Google presented *Code Transparency* to limited appraisal from developer advocates [189] as a solution to their concerns regarding AAB. However, despite its key role in the application supply chain under the AAB format, the exact security properties of Code Transparency remained untested, motivating its analysis in this thesis.

Contribution III

Draschbacher and Maar [70] present the first systematic analysis of the Code Transparency scheme for the AAB distribution format. They uncover serious shortcomings that allow an attacker to influence the app's code without invalidating its Code Transparency. These findings show that the problems introduced through the AAB distribution scheme remain unsolved, with potentially serious consequences for app developers and end users in the Android ecosystem.

In concurrent work, Lins et al. [167] proposed the Mobile App Distribution Transparency (MADT) scheme based on verifiable logs for mitigating problems introduced by signing key sharing.

3.4. API Misuse on Mobile Platforms

From a security perspective, API misuse refers to an application using APIs offered by a (platform) library in an improper way that invalidates its security guarantees, e.g., through misparametrization. In the past, most publications investigating API misuse in mobile applications focused on crypto APIs, i.e., TLS or cryptographic primitives provided by the platform's app framework. However, there is also a body of work on the misuse of other system APIs that affects an application's security posture. In the following, we discuss the state of the art in both areas.

TLS API Misuse. The first major publication to uncover crypto API misuse in mobile applications was presented by Gergiev et al. [92] in 2012, who analyzed libraries and non-browser apps on various platforms for TLS validation flaws. For Android and iOS, their dynamic testbed identified flaws in messaging and banking apps, as well as popular analytics and ad SDKs, which they attributed to complex low-level TLS APIs. In a parallel work, Fahl et al. [81] conducted the first large-scale mobile-only study, also focused on misuse of TLS/SSL APIs. Through automated static analysis, the authors uncovered TLS validation flaws in 8 % of analyzed apps and manually confirmed exploitability on a smaller subset. In 2014, Sounthiraraj et al. [241] presented SMV-Hunter, the first automated TLS misuse detector for mobile applications that combined static and dynamic analysis. Their static analysis identified entry points and user input required to reach TLS functionality in the dynamic analysis, during which the app was subjected to a MITM attack. The dynamic stage eliminated 50 % of the misuses reported by the initial static analysis as false positives. Google introduced the Network Security Configuration (NSC) in 2016 [94], which facilitated implementing advanced TLS validation and added more secure default logic. Still, Oltrogge et al. [200] in 2021 reported that apps bypassed secure TLS validation defaults, finding vulnerabilities in more than a third of the statically analyzed apps. Pourali et al. [212] conducted a dynamic analysis study in 2024, reporting TLS validation failures in 4 % of apps from Google Play and 56 % from a Chinese app

store. Third-party libraries that hijacked validation in the entire app were responsible for more than a fifth of the cases on Google Play.

Crypto API Misuse. A parallel thread of research examined the misuse of APIs for general-purpose cryptographic primitives such as ciphers. Egele et al. [75] presented CryptoLint, a static detection tool for violations of six crypto rules designed to ensure that ciphers and password-based encryption are indistinguishable under a chosen-plaintext attack. They reported that 88 % of the analyzed applications from Google Play violated at least one rule. They included case studies showcasing how the flaws rendered the affected crypto primitives ineffective. Li et al. [165] proposed iCryptoTracer, a hybrid between static and dynamic analysis for identifying misuse of crypto primitives in iOS apps. They statically determined crypto API calls that they later intercepted at runtime on a jailbroken device. By analyzing the passed parameters, they identified flaws in 65 % of the evaluated apps. In 2016, Ma et al. [174] presented the first solution for mitigating crypto API misuse in compiled Android applications, called CDRep. They identified misuse locations through static analysis and applied patches from manually assembled fix patterns. While their solution represented an important first step for users of otherwise vulnerable apps, its purely static approach had limitations, such as only being able to mitigate the first misuse per app method. Feichtner et al. [84] put forth the first purely static crypto API misuse detector on iOS, based on the crypto rules by Egele et al. [75]. They reported a crypto misuse prevalence of 82 % among evaluated iOS apps.

While early crypto API misuse detection was characterized by a race to ever higher prevalence numbers, the general trend in later works points at more nuanced analyses. Muslukhov et al. [191] extended CryptoLint by Egele et al. [75] and attributed findings to originating code packages. They noted that up to 90 % of the vulnerable Android apps only contained crypto API misuse due to the libraries they integrated. Krüger et al. [150] aimed to comprehensively cover misuse of any API in the Java Cryptography Architecture (JCA) through rules encoded in a custom definition language called CrySL. Their static analysis tool checked Android apps for violations against these rules, reporting that 95 % of evaluated apps contained at least one misuse, with manual assessment indicating recall of 100 % and precision of 85 %. Rahaman et al. [215] proposed domain-specific heuristics that significantly reduced the false positives that generally haunt static analysis. Their tool, CryptoGuard, still reported a crypto API misuse prevalence of 91 % among Android apps, but beat existing

tools in terms of precision and recall on a synthetic benchmark. In 2021, Piccolboni et al. [207] presented CryLogger, the first purely dynamic crypto API misuse detector for Android. Through a modified Android framework, their solution collected API invocations and detected misuse of TLS and crypto primitives from logged parameters. The authors compared CryLogger to CryptoGuard and found that both tools suffered from false negatives, but only CryptoGuard suffered from false positives. Recent meta-studies by Ami et al. [4] and Chen et al. [54] have analyzed limitations of state-of-the-art static crypto API misuse detectors, focusing on false negatives and false positives, respectively. The limitations of static tools and promising results of Piccolboni et al. [207] motivate the further research into dynamic approaches for detection and mitigation of API misuse in this thesis.

Contribution IV

Draschbacher et al. present CryptoShield [69], the first approach for dynamically mitigating crypto API misuse in compiled Android apps. While dynamic approaches already showed promising results in pure detectors, the authors demonstrate that they are particularly well-suited for mitigation. Although aimed at mitigation, CryptoShield's misuse detection capabilities outperform those of CryptoGuard [215] on synthetic benchmarks.

In the 2025 systematic review by Ochoa et al. [198] of approaches for automatically mitigating harmful API use, CryptoShield was still the only approach that used dynamic instrumentation.

Misuse of Other APIs. Further studies have examined security vulnerabilities arising from the misuse of a broad range of system-provided APIs. In 2014, Poeplau et al. [208] statically analyzed misuse of dynamic code loading APIs in Android applications. Apps can use these APIs to load add-ons or shared components from disk, or to implement self-update mechanisms. Poeplau et al. found that more than 9 % of evaluated apps dynamically loaded code in an insecure manner via HTTP or public storage, enabling code-injection attacks. In a follow-up paper, Falsina et al. [83] proposed an improved API for secure dynamic code loading and provided a tool for retroactively patching apps to integrate it. Bianchi et al. [42] identified widespread misuse of the fingerprint API on Android. Apps use this API to ensure that the device owner has authorized sensitive operations. It works by presenting a biometric prompt for user authentication and providing proof of successful authentication via unlocked key

material stored in the TEE. In their static analysis, Bianchi et al. [42] found that more than half of the apps using the API did not use this proof, enabling bypass attacks on a compromised OS. Zhang et al. [303] in 2025 presented a misuse analysis of the same API, but considered additional lifecycle events, such as the enrollment of new fingerprints. They reported at least one type of misuse in as many as 97 % of the analyzed apps that use fingerprint APIs. In 2021, Ibrahim et al. [127] investigated apps' misuse of app attestation APIs, in particular Google's SafetyNet API. Apps can use these APIs to obtain an attestation of their APK signature and the integrity of the operating system. This attestation is signed by a root certificate embedded in the TEE. The dynamic analysis conducted by Ibrahim et al. [127] indicated that over 50 % of apps did not check the attestation on the backend server, again enabling bypasses by a compromised OS. Yang et al. [294] analyzed improper configuration of app manifest files in Android apps, which represent the contract between the app and OS. Through misplacing or misspelling entries in this XML file, developers can introduce security vulnerabilities such as unprotected components. In fact, Yang et al. [294] identified security-relevant misuse in about 5 % of analyzed applications. Lee et al. [154] analyzed how Android apps using in-app browser APIs validated advanced TLS certificate configurations. Given Android's lenient default TLS validation logic, apps need to implement custom logic to, e.g., detect misconfigurations in TLS extensions. However, Lee et al. [154] found no such implementation in 20 popular Android apps they analyzed manually.

From the works referenced above, it is evident that most misuse-detection approaches employed static analysis. Additionally, they did not attempt to mitigate the flaws they uncovered. While it is unclear whether retrospective mitigations are possible for all API misuse cases, they offer a valuable option for end users. In cases where developers are unable to address API misuse, users, e.g., of niche apps with no alternative, are otherwise left vulnerable. Those studies that included API misuse mitigation for compiled applications used purpose-specific and computation-intensive static implementations that could hardly be repurposed for other misuse cases [83, 174]. However, API misuse mitigation is where dynamic approaches are particularly well-suited, even more so than for detection. While dynamic detection tools are bound by the quality of the exploration, these limitations do not apply to mitigation. This potential motivated the research into re-usable dynamic app patching solutions with a particular focus on API misuse mitigation.

Contribution V

Draschbacher et al. present A2P2 [68], an application-agnostic patching pipeline for Android applications. It allows specifying generic changesets even for advanced use cases dealing with, e.g., app manifests [294], and applying them to an arbitrary number of applications. Code changes are applied through dynamic instrumentation, enabling extensive call tracing and parameter manipulation at runtime.

By using A2P2 to mitigate apps that abuse unprivileged device property APIs to fingerprint users, Palfinger et al. [204] showcased the tool's versatility and adaptability to a wide range of use cases.

4

Conclusion

This thesis presented novel contributions on misconfigured security across the mobile platform stack. It uncovered severe vulnerabilities in wired and wireless mobile communication, explored attack vectors in the app supply chain, and proposed dynamic methods to detect and mitigate API misuse. Through responsible disclosure and vendor updates, our security analyses and the attacks we uncovered have had an immediate positive impact on billions of end users, protecting them against data theft during device migration and charging. For researchers, we provided novel insights into the potential of supply chain attacks and novel tooling for API misuse detection and mitigation.

In the domain of mobile communication, we investigated misconfigured security in vendor-specific implementations of standardized protocols and closed-source proprietary communication solutions. Our examinations of mitigations against malicious charging stations uncovered severe vulnerabilities across vendors caused by flawed trust models and implementations. They allowed for ChoiceJacking attacks that extract data from state-of-the-art Android and iOS devices. Through a systematic analysis, we also revealed significant design and implementation issues in Android data migration tools. Although the general operation of these tools shared commonalities across vendors, each vendor designed their own protocol. Due to the discovered security misconfigurations, all these protocols were vulnerable to wireless attacks in which a malicious party could either eavesdrop on the exchanged data or manipulate it.

Our investigations of the Android supply chain concentrated on misconfigurations in app signature schemes. Google's introduction of the Android Application Bundle format and its adoption by all major Android app stores brought a radical shift in the trust assumptions of app signing. The new format requires developers to hand over their APK signing keys to the app store operator. Our detailed analysis of the Code Transparency

scheme, designed to prevent misuse of this powerful role, revealed serious flaws. We demonstrated that there are multiple ways to manipulate an app's control flow without compromising its Code Transparency.

Our contributions to API misuse detection and mitigation in Android applications centered around a dynamic instrumentation-based approach. Through this approach, we proposed mitigating crypto API misuse in compiled Android applications by intercepting method invocations. We designed mitigation procedures for a set of common crypto API misuse cases that work transparently to application logic. We also generalized our dynamic approach into an application-agnostic patching pipeline that can be adapted for the detection and mitigation of other API misuse cases.

Future Work. The areas covered in this thesis offer ample opportunities to explore additional security misconfigurations. Both iOS and vendor-specific Android variants are constantly augmented with new communication modes. As a noteworthy example, Google and Apple recently integrated a proprietary protocol into their OSs that allows cross-platform data migration [181]. Even existing communication protocols keep evolving, e.g., to keep up with competitors and trends. For example, Google and Samsung recently added compatibility with Apple's AirDrop protocol to their QuickShare peer-to-peer file sharing service [224]. Many of these proprietary protocols are used to transmit sensitive data, so secure implementations are crucial. These developments underscore the need for rigorous, continuous, independent security evaluations of the integrated security mechanisms.

As supply chain complexity grows, so does the potential for supply chain attacks. App stores and platform providers remain particularly interesting pieces of mobile software supply chains, both from the perspective of market monopolies and digital sovereignty. For these reasons, research needs to further explore gaps in authenticity and integrity mechanisms and develop technical means to ensure the traceability and auditability of software components. The result of these efforts should be a supply chain in which the compromise or misbehavior of individual links is either entirely impossible or immediately noticeable.

API misuse continues to ail mobile applications, creating vulnerabilities that attackers can exploit. Researchers can contribute in multiple ways to solving this problem. Through their work, they can identify additional APIs prone to misuse, evaluate the prevalence of misuse in real-world applications and the impact of resulting attacks, and find methods to

mitigate misuse in the absence of developer support. They can also identify the causes of these misuse patterns and suggest improved APIs or clearer documentation.

Ultimately, these anticipated efforts have the same overarching goal as this thesis: Furthering the security posture across the mobile platform stack, i.e., protecting Android and iOS users against attacks.

References

[1] Daniel Abeles and Michael Henriksen. GitLab discovers widespread npm supply chain attack. 2025. URL: https://about.gitlab.com/blog/gitlab-discovers-widespread-npm-supply-chain-attack/ (p. 5).

[2] Abbas Acar, Güliz Seray Tuncay, Esteban Luques, Harun Oz, Ahmet Aris, and A Selcuk Uluagac. 50 Shades of Support: A Device-Centric Analysis of Android Security Updates. In: NDSS. 2024 (pp. 31, 32).

[3] Gunnar Alendal, Stefan Axelsson, and Geir Olav Dyrkolbotn. Leveraging USB Power Delivery Implementations for Digital Forensic Acquisition. In: Advances in Digital Forensics XVII. 2021 (p. 5).

[4] Amit Seal Ami, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. Why Crypto-detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques. In: IEEE Symposium on Security and Privacy (SP). 2022 (pp. 10, 44).

[5] Amnesty International. Cellebrite zero-day exploit used to target phone of Serbian student activist. 2025. URL: https://www.amnesty.org/fr/wp-content/uploads/2025/03/EUR7091182025ENGLISH.pdf (pp. 4, 7).

[6] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. BIAS: Bluetooth Impersonation AttackS. In: S&P. 2020 (p. 7).

[7] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Nearby Threats: Reversing, Analyzing, and Attacking Google's Nearby Connections on Android. In: (2019) (p. 36).

[8] AOSP. Android Debug Bridge (adb). 2025. URL: https://developer.android.com/tools/adb (p. 27).

[9] AOSP. Android Keystore system. 2026. URL: https://developer.android.com/privacy-and-security/keystore (p. 27).

[10] AOSP. App Signing. 2025. URL: https://source.android.com/docs/security/features/apksigning (p. 25).

[11] AOSP. Application Sandbox. 2025. URL: https://source.android.com/docs/security/app-sandbox (p. 22).

[12] AOSP. Foreground service types. 2026. URL: https://developer.android.com/develop/background-work/services/fgs/service-typesg (p. 22).

[13] AOSP. Full-disk encryption. 2025. URL: https://source.android.com/docs/security/features/encryption/full-disk (p. 26).

[14] AOSP. FUSE passthrough. 2025. URL: https://source.android.com/docs/core/storage/fuse-passthrough (p. 22).

[15] AOSP. Intents and intent filters. 2026. URL: https://developer.android.com/guide/components/intents-filters (p. 23).

[16] AOSP. Lock and unlock the bootloader. 2025. URL: https://source.android.com/docs/core/architecture/bootloader/locking_unlocking (pp. 26, 28).

[17] AOSP. Permissions on Android. 2026. URL: https://developer.android.com/guide/topics/permissions/overview (p. 23).

[18] AOSP. Verified Boot. 2024. URL: https://source.android.com/docs/security/features/verifiedboot (p. 26).

[19] Apple, Inc. App and system extensions. URL: https://developer.apple.com/documentation/technologyoverviews/app-extensions (p. 22).

[20] Apple, Inc. App code signing process in iOS, iPadOS, tvOS, visionOS. URL: https://support.apple.com/guide/security/app-code-signing-process-sec7c917bf14/web (p. 24).

[21] Apple, Inc. Apple Security Research Device. 2022. URL: https://support.apple.com/en-us/guide/security/seca7ff718d2/web (p. 28).

[22] Apple, Inc. Boot process for iPhone and iPad devices. URL: https://support.apple.com/en-gb/guide/security/secb3000f149/web (p. 26).

[23] Apple, Inc. Configuring background execution modes. URL: https://developer.apple.com/documentation/xcode/configuring-background-execution-modes (p. 22).

[24] Apple, Inc. Data Protection. 2024. URL: https://support.apple.com/en-us/guide/security/secf6276da8a/web (p. 26).

[25] Apple, Inc. Defining a custom URL scheme for your app. URL: https://developer.apple.com/documentation/xcode/defining-a-custom-url-scheme-for-your-app (p. 22).

[26] Apple, Inc. Enabling Developer Mode on a device. URL: https://developer.apple.com/documentation/xcode/enabling-developer-mode-on-a-device (p. 27).

[27] Apple, Inc. European Digital Markets Act (DMA). 2022. URL: https://www.apple.com/legal/dma/ (p. 25).

[28] Apple, Inc. Installing apps through alternative app distribution. 2025. URL: https://support.apple.com/en-mk/117767 (p. 25).

[29] Apple, Inc. Kernel Architecture Overview. 2013. URL: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Architecture/Architecture.html (p. 16).

[30] Apple, Inc. Requesting access to protected resources. URL: https://developer.apple.com/documentation/uikit/requesting-access-to-protected-resources (pp. 23, 26).

[31] Apple, Inc. Security of runtime process in iOS and iPadOS. 2024. URL: https://support.apple.com/guide/security/security-of-runtime-process-sec15bfe098e/web (p. 22).

[32] Apple, Inc. Storing Keys in the Keychain. URL: https://developer.apple.com/documentation/security/storing-keys-in-the-keychain (p. 27).

[33] Ioannis Arkalakis, Michalis Diamantaris, Serafeim Moustakas, Sotiris Ioannidis, Jason Polakis, and Panagiotis Ilia. Abandon all hope ye who enter here: A dynamic, longitudinal investigation of android's data safety section. In: USENIX Security. 2024 (p. 31).

[34] Ammar Askar, Fabian Fleischer, Christopher Kruegel, Giovanni Vigna, and Taesoo Kim. Malintent: Coverage guided intent fuzzing framework for android. In: NDSS. 2025 (p. 30).

[35] axi0mX. EPIC JAILBREAK: Introducing checkm8. 2019. URL: https://x.com/axi0mX/status/1177542201670168576 (p. 4).

[36] Sam Beaumont, Larry Trowell, Ruchit Patel, Paroksh Sharma, and Will Strei. Public Report: Android Quick Share Application Penetration Test. 2025. URL: https://www.netspi.com/wp-content/uploads/2025/11/google-feature-review-report.pdf (p. 37).

[37] Ian Beer. A deep dive into an NSO zero-click iMessage exploit: Remote Code Execution. 2021. URL: https://projectzero.google/2021/12/a-deep-dive-into-nso-zero-click.html (p. 4).

[38] Ian Beer. In-the-wild iOS Exploit Chain 3. 2019. URL: https://projectzero.google/2019/08/in-wild-ios-exploit-chain-3.html (p. 4).

[39] Ian Beer. Mind the Gap. 2022. URL: https://projectzero.google/2022/11/mind-the-gap.html (p. 4).

[40] Philipp Beer, Marco Squarcina, Sebastian Roth, and Martina Lindorfer. TapTrap: animation-driven tapjacking on android. In: USENIX Security. 2025 (p. 31).

[41] Maxime Rossi Bellom and Raphael Neveu. Attacking the Samsung Galaxy A* Boot Chain. 2024. URL: https://blog.quarkslab.com/attacking-the-samsung-galaxy-a-boot-chain.html (p. 4).

[42] Antonio Bianchi, Yanick Fratantonio, Aravind Machiry, Christopher Kruegel, Giovanni Vigna, Simon Pak Ho Chung, and Wenke Lee. Broken Fingers: On the Usage of the Fingerprint API in Android. In: NDSS. 2018 (pp. 44, 45).

[43] Eduardo Blázquez, Sergio Pastrana, Álvaro Feal, Julien Gamba, Platon Kotzias, Narseo Vallina-Rodriguez, and Juan Tapiador. Trouble Over-The-Air: An Analysis of FOTA Apps in the Android Ecosystem. In: S&P. 2021 (p. 31).

[44] Marcel Busch, Philipp Mao, and Mathias Payer. Spill the TeA: an empirical study of trusted application rollback prevention on android smartphones. In: USENIX Security. 2024 (p. 32).

[45] Yifeng Cai, Ziqi Zhang, Mengyu Yao, Junlin Liu, Xiaoke Zhao, Xinyi Fu, Ruoyu Li, Zhe Liu, Xiangqun Chen, Yao Guo, et al. I can tell your secrets: Inferring privacy attributes from mini-app interaction history in super-apps. In: USENIX Security. 2025 (p. 30).

[46] Hongjian Cao, Lin Huang, Shuwei Hu, Shangcheng Shi, and Yujia Liu. Owfuzz: Discovering wi-fi flaws in modern devices through over-the-air fuzzing. In: WiSec. 2023 (p. 34).

[47] Sheng Cao, Hao Zhou, Songzhou Shi, Yanjie Zhao, and Haoyu Wang. Parcel Mismatch Demystified: Addressing a Decade-Old Security Challenge in Android. In: CCS. 2025 (p. 31).

[48] Weicheng Cao, Chunqiu Xia, Sai Teja Peddinti, David Lie, Nina Taft, and Lisa M Austin. A large scale study of user behavior, expectations and engagement with android permissions. In: USENIX Security. 2021 (p. 30).

[49] Guillaume Celosia and Mathieu Cunche. Discontinued privacy: Personal data leaks in apple bluetooth-low-energy continuity protocols. In: PETS (2020) (p. 35).

[50] Anthony Chavez. Update on Plans for Privacy Sandbox Technologies. 2025. URL: https://privacysandbox.google.com/blog/update-on-plans-for-privacy-sandbox-technologies (p. 40).

[51] Huiling Chen, Wenqiang Jin, Yupeng Hu, Zhenyu Ning, Kenli Li, Zheng Qin, Mingxing Duan, Yong Xie, Daibo Liu, and Ming Li. Eavesdropping on Black-box Mobile Devices via Audio Amplifier's EMR. In: NDSS. 2024 (p. 32).

[52] Junming Chen, Xiaoyue Ma, Lannan Luo, and Qiang Zeng. Tracking You from a Thousand Miles Away! Turning a Bluetooth Device into an Apple {AirTag} Without Root Privileges. In: USENIX Security. 2025 (p. 5).

[53] Junming Chen, Xiaoyue Ma, Lannan Luo, and Qiang Zeng. Tracking you from a thousand miles away! turning a bluetooth device into an apple {AirTag} without root privileges. In: USENIX Security. 2025 (p. 36).

[54] Yikang Chen, Yibo Liu, Ka Lok Wu, Duc Viet Le, and Sze Yiu Chau. Towards Precise Reporting of Cryptographic Misuses. In: NDSS. 2024 (p. 44).

[55] Yizheng Chen, Zhoujie Ding, and David A. Wagner. Continuous Learning for Android Malware Detection. In: USENIX Security. 2023 (p. 29).

[56] Yu Chen, Yang Yu, and Lidong Zhai. {InfinityGauntlet}: Expose smartphone fingerprint authentication to brute-force attack. In: USENIX Security. 2023 (p. 32).

[57] Jiska Classen, Alexander Heinrich, Fabian Portner, Felix Rohrbach, and Matthias Hollick. Starshields for iOS: Navigating the Security Cosmos in Satellite Communication. In: NDSS. 2025 (p. 36).

[58] Shmuel Cohen and Or Yair. QuickShell: Sharing Is Caring about an RCE Attack Chain on Quick Share. 2024. URL: https://www.safebreach.com/blog/rce-attack-chain-on-quick-share/ (pp. 5, 37).

[59] Adam Conway. Google details SDK Runtime design proposal for the Android Privacy Sandbox. 2022. URL: https://www.xda-developers.com/google-details-sdk-runtime-design-proposal-android-privacy-sandbox/ (p. 39).

[60] Patrick Cronin, Xing Gao, Chengmo Yang, and Haining Wang. Charger-Surfing: Exploiting a Power Line Side-Channel for Smartphone Information Leakage. In: USENIX Security. 2021 (pp. 7, 8, 32).

[61] Abdallah Dawoud and Sven Bugiel. Bringing Balance to the Force: Dynamic Analysis of the Android Application Framework. In: NDSS. 2021 (p. 30).

[62] Abdallah Dawoud and Sven Bugiel. DroidCap: OS Support for Capability-based Permissions in Android. In: NDSS. 2019 (p. 39).

[63] Zizhuang Deng, Kai Chen, Guozhu Meng, Xiaodong Zhang, Ke Xu, and Yao Cheng. Understanding Real-world Threats to Deep Learning Models in Android Apps. In: CCS. 2022 (p. 30).

[64] Karel Dhondt, Victor Le Pochat, Yana Dimova, Wouter Joosen, and Stijn Volckaert. Swipe left for identity theft: an analysis of user data privacy risks on location-based dating apps. In: USENIX Security. 2024 (p. 30).

[65] Luca Di Bartolomeo, Philipp Mao, Yu-Jye Tung, Jessy Ayala, Samuele Doria, Paolo Celada, Marcel Busch, Joshua Garcia, Eleonora Losiouk, and Mathias Payer. Hercules Droidot and the murder on the {JNI} Express. In: USENIX Security. 2025 (p. 30).

[66] Michalis Diamantaris, Serafeim Moustakas, Lichao Sun, Sotiris Ioannidis, and Jason Polakis. This Sneaky Piggy Went to the Android Ad Market: Misusing Mobile Sensors for Stealthy Data Exfiltration. In: CCS. 2021 (p. 32).

[67] Zikan Dong, Tianming Liu, Jiapeng Deng, Li Li, Minghui Yang, Meng Wang, Guosheng Xu, and Guoai Xu. Exploring covert third-party identifiers through external storage in the android new era. In: USENIX Security. 2024 (p. 31).

[68] Florian Draschbacher. A2P2 - An Android Application Patching Pipeline Based On Generic Changesets. In: ARES. 2023 (pp. 6, 7, 11, 46).

[69] Florian Draschbacher and Johannes Feichtner. CryptoShield: Automatic On-Device Mitigation for Crypto API Misuse in Android Applications. In: AsiaCCS. 2023 (pp. 6, 7, 11, 44).

[70] Florian Draschbacher and Lukas Maar. Manifest Problems: Analyzing Code Transparency for Android Application Bundles. In: ACSAC. 2024 (pp. 6, 7, 10, 41).

[71] Florian Draschbacher, Lukas Maar, Mathias Oberhuber, and Stefan Mangard. ChoiceJacking: Compromising Mobile Devices through Malicious Chargers like a Decade Ago. In: USENIX Security. 2025 (pp. 6–8, 32, 35).

[72] Florian Draschbacher, Lukas Maar, Lorenz Schumm, Rene Denifl, Treffner Lukas, and Stefan Mangard. Clone2Pwn: A Systematic Security Analysis of Data Migration Tools in the Android Ecosystem. In: ESORICS. 2026 (pp. 6, 7, 9, 37).

[73] Shaoyong Du, Xin Liu, Guoqing Lai, and Xiangyang Luo. Watch Out for Race Condition Attacks When Using Android External Storage. In: USENIX Security. 2022 (p. 31).

[74] Sayon Duttagupta, Seppe Wyns, Nikola Antonijevic, Bart Preneel, and Dave Singelee. WhisperPair: Hijacking Bluetooth Accessories Using Google Fast Pair. 2026. URL: https://whisperpair.eu (p. 5).

[75] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In: ACM SIGSAC Conference on Computer and Communications Security (CCS). 2013 (pp. 10, 43).

[76] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting privacy leaks in ios applications. In: NDSS. 2011 (p. 38).

[77] Yusra Elbitar, Michael Schilling, Trung Tin Nguyen, Michael Backes, and Sven Bugiel. Explanation beats context: The effect of timing & rationales on users' runtime permission decisions. In: USENIX Security. 2021 (p. 30).

[78] Dom Elliott. The future of Android App Bundles is here. online. June 2021. URL: https://android-developers.googleblog.com/2021/06/the-future-of-android-app-bundles-is.html (pp. 5, 9).

[79] Dom Elliott. What a new publishing format means for the future of Android. Accessed on Feb. 17, 2020. Oct. 2018. URL: https://medium.com/googleplaydev/what-a-new-publishing-format-means-for-the-future-of-android-2e34981793a (p. 9).

[80] Sascha Fahl, Sergej Dechand, Henning Perl, Felix Fischer, Jaromir Smrcek, and Matthew Smith. Hey, NSA: Stay Away from my Market! Future Proofing App Markets against Powerful Attackers. In: CCS. 2014 (p. 40).

[81] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: an analysis of android SSL (in)security. In: CCS. 2012 (p. 42).

[82] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. Why eve and mallory love android: an analysis of android SSL (in)security. In: ACM Conference on Computer and Communications Security (CCS). 2012 (p. 10).

[83] Luca Falsina, Yanick Fratantonio, Stefano Zanero, Christopher Kruegel, Giovanni Vigna, and Federico Maggi. Grab 'n Run: Secure and Practical Dynamic Code Loading for Android Applications. In: Proceedings of the 31st Annual Computer Security Applications Conference. 2015 (pp. 44, 45).

[84] Johannes Feichtner, David Missmann, and Raphael Spreitzer. Automated Binary Analysis on iOS - A Case Study on Cryptographic Misuse in iOS Applications. In: WiSec. 2018 (p. 43).

[85] FireEye. Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor. 2020. URL: https://cloud.google.com/blog/topics/threat-intelligence/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor (p. 5).

[86] Jeff Forristal. Android: One Root to Own Them All. 2013. URL: https://media.blackhat.com/us-13/US-13-Forristal-Android-One-Root-to-Own-Them-All-Slides.pdf (p. 5).

[87] Lorenzo Franceschi-Bicchierai. The Fortnite Trial Is Exposing Details About the Biggest iPhone Hack on Record. 2021. URL: https://www.vice.com/en/article/the-fortnite-trial-is-exposing-details-about-the-biggest-iphone-hack-of-all-time/ (p. 40).

[88] Ivan Frantric. DER Entitlements: The (Brief) Return of the Psychic Paper. 2023. URL: https://projectzero.google/2023/01/der-entitlements-brief-return-of.html (p. 4).

[89] Anders Freund. Backdoor in upstream xz/liblzma leading to ssh server compromise. 2024. URL: https://www.openwall.com/lists/oss-security/2024/03/29/4 (p. 5).

[90] Cuiying Gao, Gaozhun Huang, Heng Li, Bang Wu, Yueming Wu, and Wei Yuan. A Comprehensive Study of Learning-based Android Malware Detectors under Challenging Environments. In: ICSE. 2024 (p. 29).

[91] Jun Gao, Pingfan Kong, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Negative results on mining crypto-API usage rules in Android apps. In: 16th International Conference on Mining Software Repositories (MSR). 2019 (p. 10).

[92] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In: the ACM Conference on Computer and Communications Security (CCS). 2012 (p. 42).

[93] Heloise Gollier and Mathy Vanhoef. SSID Confusion: Making Wi-Fi Clients Connect to the Wrong Network. In: WiSec. 2024 (p. 7).

[94] Google. Andrid Developer Documentation: Network security configuration. 2021. URL: https://developer.android.com/training/articles/security-config (p. 42).

[95] Google. Android Developers: Architecture overview. 2025. URL: https://source.android.com/docs/core/architecture (p. 18).

[96] Google. Android Developers: bundletool. online. Accessed 2023-12-05. July 2023. URL: https://developer.android.com/tools/bundletool (p. 9).

[97] Google. Android Developers: Code transparency for app bundles. online. Accessed 2023-12-05. July 2021. URL: https://developer.android.com/guide/app-bundle/code-transparency (pp. 5, 10).

[98] Google. Android Developers: Platform architecture. 2024. URL: https://developer.android.com/guide/platform (pp. 18, 19).

[99] Google. Android Developers: SDK Runtime overview. 2025. URL: https://web.archive.org/web/20260214034232/https://privacysandbox.google.com/private-advertising/sdk-runtime/architecture (pp. 39, 40).

[100] Google. Android runtime and Dalvik. 2024. URL: https://source.android.com/docs/core/runtime (p. 19).

[101] Google. Copy apps & data from an Android to a new Android device. Accessed: October 08, 2025. 2023. URL: https://support.google.com/android/answer/13761358?hl=en (p. 37).

[102] Google. External USB fuzzing for Linux kernel. 2019. URL: https://github.com/google/syzkaller/blob/master/docs/linux/external_fuzzing_usb.md (p. 7).

[103] Google. Generic Kernel Image (GKI) project. 2025. URL: https://source.android.com/docs/core/architecture/kernel/generic-kernel-image (p. 17).

[104] Google. Hardware abstraction layer (HAL) overview. 2025. URL: https://source.android.com/docs/core/architecture/hal (p. 19).

[105] Google. Hardware security best practices. 2025. URL: https://source.android.com/docs/security/best-practices/hardware (p. 17).

[106] Google. How to transfer data between Android phones. Accessed: October 08, 2025. 2025. URL: https://web.archive.org/web/20251008132752/https://www.android.com/transfer-data-android-to-android/ (p. 37).

[107] Google. Use of All files access (MANAGE_EXTERNAL_STORAGE) permission. URL: https://support.google.com/googleplay/android-developer/answer/10467955?hl=en (p. 25).

[108] Sigmund Albert Gorski III, Seaver Thorn, William Enck, and Haining Chen. {FReD}: Identifying file {Re-Delegation} in android system services. In: USENIX Security. 2022 (p. 31).

[109] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In: WISEC. 2012 (p. 38).

[110] Ping He, Yifan Xia, Xuhong Zhang, and Shouling Ji. Efficient Query-Based Attack against ML-Based Android Malware Detection under Zero Knowledge Setting. In: CCS. 2023 (p. 29).

[111] Tom Hebb. Missing signs: how several brands forgot to secure a key piece of Android. 2024. URL: https://rtx.meta.security/exploitation/2024/01/30/Android-vendors-APEX-test-keys.html (p. 38).

[112] Alexander Heinrich, Niklas Bittner, and Matthias Hollick. Airguard-protecting android users from stalking attacks by apple find my devices. In: WiSec. 2022 (p. 36).

[113] Alexander Heinrich, Matthias Hollick, Thomas Schneider, Milan Stute, and Christian Weinert. {PrivateDrop}: Practical {privacy-preserving} authentication for Apple {AirDrop}. In: USENIX Security. 2021 (p. 36).

[114] Alexander Heinrich, Milan Stute, Tim Kornhuber, and Matthias Hollick. Who Can Find My Devices? Security and Privacy of Apple's Crowd-Sourced Bluetooth Location Tracking System. In: PETS (2021) (p. 36).

[115] Dennis Heinze, Jiska Classen, and Matthias Hollick. ToothPicker: Apple picking in the iOS bluetooth stack. In: WOOT. 2020 (p. 7).

[116] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin Butler. FIRMWIRE: Transparent dynamic analysis for cellular baseband firmware. In: NDSS. 2022 (p. 33).

[117] Tuan Dinh Hoang, Taekkyung Oh, CheolJun Park, Insu Yun, and Yongdae Kim. LLFUZZ: An Over-the-Air Dynamic Testing Framework for Cellular Baseband Lower Layers. In: USENIX Security. 2025 (p. 33).

[118] Honor. MagicOS 10. 2025. URL: https://www.honor.com/global/magic-os/ (p. 16).

[119] Yangyu Hu, Haoyu Wang, Ren He, Li Li, Gareth Tyson, Ignacio Castro, Yao Guo, Lei Wu, and Guoai Xu. Mobile App Squatting. In: WWW. 2020 (p. 41).

[120] Jie Huang, Michael Backes, and Sven Bugiel. A11y and Privacy don't have to be mutually exclusive: Constraining Accessibility Service Misuse on Android. In: USENIX Security. 2021 (p. 31).

[121] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android. In: CCS. 2017 (p. 39).

[122] HUAWEI. HUAWEI AppGallery. URL: https://consumer.huawei.com/en/mobileservices/appgallery/ (p. 25).

[123] Huawei. EMUI 13. 2023. URL: https://consumer.huawei.com/en/emui-13/ (p. 16).

[124] Syed Rafiul Hussain, Imtiaz Karim, Abdullah Al Ishtiaq, Omar Chowdhury, and Elisa Bertino. Noncompliance as deviant behavior: An automated black-box noncompliance checker for 4g lte cellular devices. In: CCS. 2021 (p. 33).

[125] Sönke Huster, Matthias Hollick, and Jiska Classen. To boldly go where no fuzzer has gone before: Finding bugs in Linux' wireless stacks through virtio devices. In: S&P. 2024 (p. 34).

[126] Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. SafetyNOT: on the usage of the SafetyNet attestation API in Android. In: MobiSys '21: The 19th Annual International Conference on Mobile Systems, Applications, and Services. 2021 (p. 6).

[127] Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. SafetyNOT: on the usage of the SafetyNet attestation API in Android. In: MobiSys. 2021 (p. 45).

[128] Internet Society. NDSS Symposium 2026 Call for Papers. 2026. URL: https://web.archive.org/web/20251124200736/https://www.ndss-symposium.org/ndss2026/submissions/call-for-papers/ (p. 15).

[129] Jamf Threat Labs. Unauthorized access to iCloud: analyzing an iOS vulnerability that could expose sensitive data to attackers. 2024. URL: https://www.jamf.com/blog/tcc-bypass-steals-data-from-icloud/ (p. 4).

[130] Jisoo Jang, Minsuk Kang, and Dokyung Song. ReUSB: replay-guided USB driver fuzzing. In: USENIX Security. 2023 (p. 7).

[131] Seongyun Jeong, Minseong Choi, Haehyun Cho, Seokwoo Choi, Hyungsub Kim, and Yuseok Jeon. Intent-aware Fuzzing for Android Hardened Application. In: CCS. 2025 (p. 30).

[132] Yuede Ji, Mohamed Elsabagh, Ryan Johnson, and Angelos Stavrou. {DEFInit}: An Analysis of Exposed Android Init Routines. In: USENIX Security. 2021 (p. 31).

[133] Yan Jiang, Xiaoyu Ji, Kai Wang, Chen Yan, Richard Mitev, Ahmad-Reza Sadeghi, and Wenyuan Xu. WIGHT: Wired ghost touch attack on capacitive touchscreens. In: S&P. 2022 (pp. 8, 32).

[134] Wenqiang Jin, Srinivasan Murali, Huadi Zhu, and Ming Li. Periscope: A Keystroke Inference Attack Using Human Coupled Electromagnetic Emanations. In: CCS. 2021 (p. 32).

[135] Joonkyo Jung, Jisoo Jang, Jo Yongwan, Jonas Vinck, Alexios Voulimeneas, Stijn Volckaert, and Dokyung Song. Moneta: Ex-Vivo GPU Driver Fuzzing by Recalling In-Vivo Execution States. In: NDSS (2025) (p. 32).

[136] Chariton Karamitas. Remote exploitation of a man-in-the-disk vulnerability in WhatsApp (CVE-2021-24027). 2021. URL: https://census-labs.com/news/2021/04/14/whatsapp-mitd-remote-exploitation-CVE-2021-24027/ (p. 4).

[137] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. Opening Pandora's box through ATFuzzer: dynamic analysis of AT interface for Android smartphones. In: ACSAC. 2019 (pp. 5, 7, 34).

[138] Hao Ke, Bernardo Rufino, Yang Yang, and Maria Uretsky. Android parcels: the bad, the good and the better. 2022. URL: https://i.blackhat.com/EU-22/Wednesday-Briefings/EU-22-Ke-Android-Parcels-Introducing-Android-Safer-Parcel.pdf (p. 4).

[139] Rishabh Khandelwal, Asmit Nayak, Paul Chung, and Kassem Fawaz. Unpacking privacy labels: a measurement and developer perspective on google's data safety section. In: USENIX Security. 2024 (p. 31).

[140] Eunsoo Kim, Min Woo Baek, CheolJun Park, Dongkwan Kim, Yongdae Kim, and Insu Yun. {BASECOMP}: A comparative analysis for integrity protection in cellular baseband software. In: USENIX Security. 2023 (p. 33).

[141] Jason Kim, Stephan Van Schaik, Daniel Genkin, and Yuval Yarom. iLeakage: Browser-based timerless speculative execution attacks on Apple devices. In: CCS. 2023 (p. 32).

[142] Kyungtae Kim, Sungwoo Kim, Kevin R. B. Butler, Antonio Bianchi, Rick Kennell, and Dave (Jing) Tian. Fuzz The Power: Dual-role State Guided Black-box Fuzzing for USB Power Delivery. In: USENIX Security. 2023 (pp. 5, 7, 34).

[143] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin R. B. Butler, Antonio Bianchi, and Dave Jing Tian. FuzzUSB: Hybrid Stateful Fuzzing of USB Gadget Stacks. In: S&P. 2022 (pp. 7, 34).

[144] Daniel Klischies, Philipp Mackensen, and Veelasha Moonsamy. Vulnerability, Where Art Thou? An Investigation of Vulnerability Management in Android Smartphone Chipsets. In: NDSS. 2024 (p. 32).

[145] Jeffrey Knockel, Mona Wang, and Zoë Reichert. The Not-So-Silent Type: Vulnerabilities in Chinese IME Keyboards' Network Security Protocols. In: CCS. 2024 (p. 30).

[146] Daniel Komaromy. CVE-2021-22429: Huawei Buffer Overflow in BootROM USB Stack. 2021. URL: https://labs.taszk.io/blog/post/bootrom_usb/ (p. 4).

[147] Platon Kotzias, Juan Caballero, and Leyla Bilge. How did that get in my phone? unwanted app distribution on android devices. In: S&P. 2021 (pp. 31, 40).

[148] Uri Kratz, Avi Lumelsky, and Gal Elbaz. Airborne: Wormable Zero-Click Remote Code Execution (RCE) in AirPlay Protocol Puts Apple & IoT Devices at Risk. 2025. URL: https://www.oligo.security/blog/airborne (p. 5).

[149] Brian Krebs. Beware of Juice-Jacking. 2011. URL: https://krebsonsecurity.com/2011/08/beware-of-juice-jacking/ (pp. 8, 34).

[150] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In: IEEE Trans. Software Eng. 47.11 (2019) (p. 43).

[151] Georgy Kucherin. The Notepad++ supply chain attack — unnoticed execution chains and new IoCs. 2026. URL: https://securelist.com/notepad-supply-chain-attack/118708/ (p. 5).

[152] Alexander S. La Cour, Khurram K. Afridi, and G. Edward Suh. Wireless Charging Power Side-Channel Attacks. In: CCS. 2021 (p. 32).

[153] Billy Lau, Yeongjin Jang, Chengyu Song, Tielei Wang, Pak Ho Chung, and Paul Royal. Mactans: Injecting Malware into iOS Devices via Malicious Chargers. In: Black Hat USA. 2013 (pp. 8, 34).

[154] Woonghee Lee, Junbeom Hur, and Hyunsoo Kwon. Deep Dive into In-app Browsers: Uncovering Hidden Pitfalls in Certificate Validation. In: CCS. 2025 (p. 45).

[155] Yu Tsung Lee, Haining Chen, William Enck, Hayawardh Vijayakumar, Ninghui Li, Zhiyun Qian, Giuseppe Petracca, and Trent Jaeger. PolyScope: Multi-Policy Access Control Analysis to Triage Android Scoped Storage. In: USENIX Security. 2021 (p. 30).

[156] Chongqing Lei, Zhen Ling, Yue Zhang, Kai Dong, Kaizheng Liu, Junzhou Luo, and Xinwen Fu. Do not give a dog bread every time he wags his tail: Stealing passwords through content queries (CONQUER) attacks. In: NDSS. 2023 (p. 31).

[157] Zeyu Lei, Güliz Seray Tuncay, Beatrice Carissa Williem, Z Berkay Celik, and Antonio Bianchi. ScopeVerif: Analyzing the Security of Android's Scoped Storage via Differential Analysis. In: NDSS. 2025 (p. 30).

[158] Ernst Leierzopf, Stefan Kempinger, René Mayrhofer, and Michael Roland. AVBTestKeyInTheWild: Bypassing Android Verified Boot Using A Firmware Supply Chain Vulnerability on Locked Devices. In: (2025) (p. 38).

[159] Jonathan Levin. *OS Internals: Volume 1 User Mode. In: Jonathan Levin, 2020 (pp. 18, 19).

[160] Heng Li, Zhang Cheng, Bang Wu, Liheng Yuan, Cuiying Gao, Wei Yuan, and Xiapu Luo. Black-box adversarial example attack towards FCG based android malware detection under incomplete feature information. In: USENIX Security. 2023 (p. 29).

[161] Heng Li, Zhiyuan Yao, Bang Wu, Cuiying Gao, Teng Xu, Wei YUAN, and Xiapu Luo. Automated Mass Malware Factory: The Convergence of Piggybacking and Adversarial Example in Android Malicious Software Generation. In: NDSS. 2025 (p. 29).

[162] Jiawei Li, Jian Mao, Jun Zeng, Qixiao Lin, Shaowen Feng, and Zhenkai Liang. UIHASH: detecting similar android uis through grid-based visual appearance representation. In: USENIX Security. 2024 (p. 29).

[163] Shuai Li, Zhemin Yang, Nan Hua, Peng Liu, Xiaohan Zhang, Guangliang Yang, and Min Yang. Collect Responsibly But Deliver Arbitrarily? A Study on Cross-User Privacy Leakage in Mobile Apps. In: CCS. 2022 (p. 30).

[164] Wenqiang Li, Haohuang Wen, and Zhiqiang Lin. BaseMirror: Automatic Reverse Engineering of Baseband Commands from Android's Radio Interface Layer. In: CCS. 2024 (p. 33).

[165] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications. In: NSS. 2015 (p. 43).

[166] Yan Lin, Joshua Wong, Xiang Li, Haoyu Ma, and Debin Gao. Peep with a mirror: breaking the integrity of android app sandboxing via unprivileged cache side channel. In: USENIX Security. 2024 (pp. 31, 32).

[167] Mario Lins, René Mayrhofer, Michael Roland, and Alastair R Beresford. Mobile App Distribution Transparency (MADT): Design and evaluation of a system to mitigate necessary trust in mobile app distribution systems. In: NordSec. 2023 (pp. 41, 42).

[168] Dexin Liu, Yue Xiao, Chaoqi Zhang, Kaitao Xie, Xiaolong Bai, Shikun Zhang, and Luyi Xing. iHunter: Hunting Privacy Violations at Scale in the Software Supply Chain on iOS. In: USENIX Security. 2024 (pp. 9, 39).

[169] Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang. Measuring the Insecurity of Mobile Deep Links of Android. In: USENIX Security. 2017 (p. 6).

[170] Yijing Liu, Yiming Zhang, Baojun Liu, and Haixun Duan. Cracks in the Walled Garden: Dissecting the Gray-Market of Unauthorized iOS App Distribution via Ad Hoc Sideloading. In: (2026) (p. 41).

[171] Zhuoran Liu, Niels Samwel, Leo Weissbart, Zhengyu Zhao, Dirk Lauret, Lejla Batina, and Martha A. Larson. Screen Gleaning: A Screen Reading TEMPEST Attack on Mobile Devices Exploiting an Electromagnetic Side Channel. In: NDSS. 2021 (p. 32).

[172] Shang Ma, Chaoran Chen, Shao Yang, Shifu Hou, Toby Jia-Jun Li, Xusheng Xiao, Tao Xie, and Yanfang Ye. Careful about what app promotion ads recommend! detecting and explaining malware promotion via app promotion graph. In: NDSS. 2025 (p. 31).

[173] Siqi Ma, Hehao Li, Wenbo Yang, Juanru Li, Surya Nepal, and Elisa Bertino. Certified Copy? Understanding Security Risks of Wi-Fi Hotspot based Android Data Clone Services. In: ACSAC. 2020 (pp. 8, 37).

[174] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In: 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS). 2016 (pp. 43, 45).

[175] Lukas Maar, Florian Draschbacher, Lukas Lamster, and Stefan Mangard. Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels. In: USENIX Security. 2024 (pp. 12, 33).

[176] Lukas Maar, Florian Draschbacher, Lorenz Schumm, Ernesto Martínez García, and Stefan Mangard. The Doom of Device Drivers: Your Android Device (most likely) has N-day Kernel Vulnerabilities. In: USENIX Security. 2025 (pp. 12, 33).

[177] Tarjei Mandt, Mathew Solnik, and David Wang. Demystifying the Secure Enclave Processor. 2016. URL: https://blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf (p. 17).

[178] Philipp Mao, Marcel Busch, and Mathias Payer. NASS: fuzzing all native android system services with interface awareness and coverage. In: USENIX Security. 2025 (p. 31).

[179] Jeremy Martin, Douglas Alpuche, Kristina Bodeman, Lamont Brown, Ellis Fenske, Lucas Foppe, Travis Mayberry, Erik C. Rye, Brandon Sipes, and Sam Teplov. Handoff All Your Privacy - A Review of Apple's Bluetooth Low Energy Continuity Protocol. In: PETS (2019) (p. 35).

[180] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The Android Platform Security Model. In: CoRR (2023) (pp. 9, 16, 23).

[181] Marcus Mendes. Apple readies new framework to let iPhone users migrate app data to and from Android. 2025. URL: https://9to5mac.com/2025/10/22/appmigrationkit-framework-migration-between-android-and-ios/ (p. 48).

[182] Mark Huasong Meng, Qing Zhang, Guangshuai Xia, Yuwei Zheng, Yanjun Zhang, Guangdong Bai, Zhi Liu, Sin G Teo, and Jin Song Dong. Post-GDPR Threat Hunting on Android Phones: Dissecting OS-level Safeguards of User-unresettable Identifiers. In: NDSS. 2023 (p. 30).

[183] Weizhi Meng, Fei Fei, Wenjuan Li, and Man Ho Au. Harvesting Smartphone Privacy Through Enhanced Juice Filming Charging Attacks. In: ISC. Vol. 10599. Lecture Notes in Computer Science. 2017 (pp. 8, 34).

[184] Weizhi Meng, Lee Wang Hao, Murali Srirangam Ramanujam, and S. P. T. Krishnan. JuiceCaster: Towards automatic juice filming attacks on smartphones. In: J. Netw. Comput. Appl. 68 (2016), pp. 201–212 (pp. 8, 34).

[185] Ulrike Meyer and Susanne Wetzel. A man-in-the-middle attack on UMTS. In: Proceedings of the 3rd ACM Workshop on Wireless Security. 2004 (p. 7).

[186] Xianghang Mi, Siyuan Tang, Zhengyi Li, Xiaojing Liao, Feng Qian, and XiaoFeng Wang. Your phone is my proxy: Detecting and understanding mobile proxy networks. In: Proceeding of ISOC Network and Distributed System Security Symposium (NDSS), 2021. 2021 (p. 39).

[187] Reham Mohamed, Arjun Arunasalam, Habiba Farrukh, Jason Tong, Antonio Bianchi, and Z. Berkay Celik. ATTention please! an investigation of the app tracking transparency permission. In: USENIX Security. 2024 (p. 30).

[188] Zahra Mousavi, Chadni Islam, Muhammad Ali Babar, Alsharif Abuadbba, and Kristen Moore. Detecting Misuse of Security APIs: A Systematic Review. In: ACM Comput. Surv. 57.12 (2025) (p. 10).

[189] Mark Murphy. Initial Thoughts on Code Transparency. 2021. URL: https://commonsware.com/blog/2021/06/29/initial-thoughts-code-transparency.html (p. 41).

[190] Mark Murphy. Uncomfortable Questions About App Signing. online. Sept. 2020. URL: https://commonsware.com/blog/2020/09/23/uncomfortable-questions-app-signing.html (p. 41).

[191] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. Source Attribution of Cryptographic API Misuse in Android Applications. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIA CCS). 2018 (p. 43).

[192] Marc Newlin. Hi, My Name Is Keyboard. 2023. URL: https://github.com/marcnewlin/hi_my_name_is_keyboard (p. 4).

[193] Tao Ni, Guohao Lan, Jia Wang, Qingchuan Zhao, and Weitao Xu. Eavesdropping mobile app activity via radio-frequency energy harvesting. In: USENIX Security. 2023 (p. 32).

[194] Tao Ni, Jianfeng Li, Xiaokuan Zhang, Chaoshun Zuo, Wubing Wang, Weitao Xu, Xiapu Luo, and Qingchuan Zhao. Exploiting Contactless Side Channels in Wireless Charging Power Banks for User Privacy Inference via Few-shot Learning. In: MobiCom. 2023 (p. 32).

[195] Tao Ni, Xiaokuan Zhang, and Qingchuan Zhao. Recovering Fingerprints from In-Display Fingerprint Sensors via Electromagnetic Side Channel. In: CCS. 2023 (p. 32).

[196] Karsten Nohl and Jakob Lell. BadUSB - On Accessories That Turn Evil. In: Black Hat USA. 2014 (pp. 8, 34).

[197] Mathias Oberhuber, Martin Unterguggenberger, Lukas Maar, Andreas Kogler, and Stefan Mangard. Power-related side-channel attacks using the Android sensor framework. In: NDSS. 2025 (p. 32).

[198] Lina Ochoa, Muhammad Hammad, Görkem Giray, Önder Babur, and Kwabena Bennin. Characterising harmful API uses and repair techniques: Insights from a systematic review. In: Comput. Sci. Rev. 57.C (2025) (p. 44).

[199] Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications. In: 30th USENIX Security Symposium. 2021 (p. 6).

[200] Marten Oltrogge, Nicolas Huaman, Sabrina Klivan, Yasemin Acar, Michael Backes, and Sascha Fahl. Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications. In: USENIX Security. 2021 (p. 42).

[201] Oppo. ColorOS 16. 2025. URL: https://www.oppo.com/en/coloros16/ (p. 16).

[202] Jonathan Bar Or, Sang Shin Jung, Michael Peck, Joe Mansour, and Apurva Kumar. Android apps with millions of downloads exposed to high-severity vulnerabilities. 2022. URL: https://www.microsoft.com/en-us/security/blog/2022/05/27/android-apps-with-millions-of-downloads-exposed-to-high-severity-vulnerabilities/ (p. 4).

[203] Oversecured. Oversecured automatically discovers persistent code execution in the Google Play Core Library. 2020. URL: https://blog.oversecured.com/Oversecured-automatically-discovers-persistent-code-execution-in-the-Google-Play-Core-Library/ (p. 4).

[204] Gerald Palfinger. AndroPROTECT: Hardening the Android API Against Fingerprinting. In: NSS. 2024 (p. 46).

[205] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In: AsiaCCS. 2012 (pp. 38, 39).

[206] André Pereira, Manuel Eduardo Correia, and Pedro Brandão. USB Connection Vulnerabilities on Android Smartphones: Default and Vendors' Customizations. In: Communications and Multimedia Security. Vol. 8735. Lecture Notes in Computer Science. 2014, pp. 19–32 (p. 34).

[207] Luca Piccolboni, Giuseppe Di Guglielmo, Luca P. Carloni, and Simha Sethumadhavan. CRYLOGGER: Detecting Crypto Misuses Dynamically. In: 42nd IEEE Symposium on Security and Privacy (SP). 2021 (pp. 10, 44).

[208] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In: 21st Annual Network and Distributed System Security Symposium (NDSS). 2014 (pp. 6, 44).

[209] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, but verify: A longitudinal analysis of android oem compliance and customization. In: S&P. 2021 (p. 31).

[210] Andrea Possemato, Dario Nisi, and Yanick Fratantonio. Preventing and Detecting State Inference Attacks on Android. In: NDSS. 2021 (p. 31).

[211] Sajjad Pourali, Nayanamana Samarasinghe, and Mohammad Mannan. Hidden in Plain Sight: Exploring Encrypted Channels in Android Apps. In: CCS. 2022 (p. 30).

[212] Sajjad Pourali, Xiufen Yu, Lianying Zhao, Mohammad Mannan, and Amr Youssef. Racing for {TLS} Certificate Validation: A Hijacker's Guide to the Android {TLS} Galaxy. In: USENIX Security. 2024 (p. 42).

[213] Sajjad Pourali, Xiufen Yu, Lianying Zhao, Mohammad Mannan, and Amr Youssef. Racing for TLS Certificate Validation: A Hijacker's Guide to the Android TLS Galaxy. In: USENIX Security. 2024 (p. 9).

[214] Qualcomm. Snapdragon 8 Elite Gen 5 Mobile Platform. 2025. URL: https://www.qualcomm.com/smartphones/products/8-series/snapdragon-8-elite-gen-5#specs (p. 17).

[215] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In: ACM SIGSAC Conference on Computer and Communications Security (CCS). 2019 (pp. 43, 44).

[216] Mishaal Rahman. Breaking: Google will now only release Android source code twice a year. 2026. URL: https://www.androidauthority.com/aosp-source-code-schedule-3630018/ (p. 16).

[217] Ali Ranjbar, Tianchang Yang, Kai Tu, Saaman Khalilollahi, and Syed Rafiul Hussain. Stateful analysis and fuzzing of commercial baseband firmware. In: S&P. 2025 (p. 33).

[218] Reef Spektor, Eran Vaknin. Vulnerabilities in CocoaPods Open the Door to Supply Chain Attacks Against Thousands of iOS and MacOS Applications. 2024. URL: https://www.evasec.io/blog/eva-discovered-supply-chain-vulnerabities-in-cocoapods (pp. 4, 40).

[219] Zeinab El-Rewini, Zhuo Zhang, and Yousra Aafer. Poirot: Probabilistically Recommending Protections for the Android Framework. In: CCS. 2022 (p. 30).

[220] Nils Rollshausen, Alexander Heinrich, Matthias Hollick, and Jiska Classen. WatchWitch: Interoperability, Privacy, and Autonomy for the Apple Watch. In: PETS (2025) (p. 36).

[221] Matthew Rossi, Dario Facchinetti, Enrico Bacis, Marco Rosa, and Stefano Paraboschi. SEApp: Bringing Mandatory Access Control to Android Apps. In: USENIX Security. 2021 (pp. 32, 39).

[222] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In: USENIX Security. 2020 (p. 34).

[223] Samsung. Galaxy Store. 2018. URL: https://galaxystore.samsung.com/apps (p. 25).

[224] Samsung. Samsung Brings AirDrop Support to Quick Share with Galaxy S26 Series. 2026. URL: https://news.samsung.com/us/samsung-airdrop-quick-share-galaxy-s26-series (p. 48).

[225] Jeremy Scahill and Josh Begley. The great SIM heist. 2015. URL: https://theintercept.com/2015/02/19/great-sim-heist/ (p. 38).

[226] Domien Schepers, Aanjhan Ranganathan, and Mathy Vanhoef. Framing frames: bypassing Wi-Fi encryption by manipulating transmit queues. In: USENIX Security. 2023 (p. 7).

[227] David Schmidt, Alexander Ponticello, Magdalena Steinböck, Katharina Krombholz, and Martina Lindorfer. Analyzing the iOS Local Network Permission from a Technical and User Perspective. In: S&P. 2025 (p. 30).

[228] David Schmidt, Sebastian Schrittwieser, and Edgar R. Weippl. Supply Chain Insecurity: Exposing Vulnerabilities in iOS Dependency Management Systems. In: CoRR abs/2601.20638 (2026) (p. 40).

[229] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. FLEXDROID: Enforcing In-App Privilege Separation in Android. In: NDSS. 2016 (p. 39).

[230] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust dies in darkness: Shedding light on samsung's {TrustZone} keymaster design. In: USENIX Security. 2022 (p. 32).

[231] Haoqi Shan, Boyi Zhang, Zihao Zhan, Dean Sullivan, Shuo Wang, and Yier Jin. Invisible finger: Practical electromagnetic interference attack on touchscreen-based electronic devices. In: S&P. 2022 (p. 32).

[232] Shashi Shekhar, Michael Dietz, and Dan S Wallach. {AdSplit}: Separating smartphone advertising from applications. In: USENIX Security. 2012 (p. 39).

[233] Bingyu Shen, Lili Wei, Chengcheng Xiang, Yudong Wu, Mingyao Shen, Yuanyuan Zhou, and Xinxin Jin. Can Systems Explain Permissions Better? Understanding Users' Misperceptions under Smartphone Runtime Permission Model. In: USENIX Security. 2021 (p. 30).

[234] Yun Shen, Pierre-Antoine Vervier, and Gianluca Stringhini. A large-scale temporal measurement of android malicious apps: Persistence, migration, and lessons learned. In: USENIX Security. 2022 (p. 40).

[235] Yun Shen, Pierre-Antoine Vervier, and Gianluca Stringhini. Understanding Worldwide Private Information Collection on Android. In: NDSS. 2021 (p. 30).

[236] Yizhe Shi, Zhemin Yang, Kangwei Zhong, Guangliang Yang, Yifan Yang, Xiaohan Zhang, and Min Yang. The Skeleton Keys: A Large Scale Analysis of Credential Leakage in Mini-apps. In: NDSS. 2025 (p. 30).

[237] Lukasz Siewierski. New APVI entry: platform certificates used to sign malware. 2022. URL: https://x.com/maldr0id/status/1598068216391405568 (pp. 4, 38).

[238] Siguza. Psychic Paper. 2020. URL: https://blog.siguza.net/psychicpaper/ (p. 4).

[239] SiOS-Submission. Security issue of GCDWebUploader. 2019. URL: https://github.com/swisspol/GCDWebServer/issues/433 (p. 4).

[240] Pallavi Sivakumaran and Jorge Blasco. A Study of the Feasibility of Co-located App Attacks against BLE and a Large-Scale Analysis of the Current Application-Layer Security Landscape. In: USENIX Security. 2019 (p. 34).

[241] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In: 21st Annual Network and Distributed System Security Symposium (NDSS). 2014 (p. 42).

[242] Statcounter. Mobile Operating System Market Share Worldwide. 2026. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide (p. 15).

[243] Lukas Stefanko. Cursed tapes: Exploiting the EvilVideo vulnerability on Telegram for Android. 2024. URL: https://www.welivesecurity.com/en/eset-research/cursed-tapes-exploiting-evilvideo-vulnerability-telegram-android/ (p. 4).

[244] Milan Stute, Alexander Heinrich, Jannik Lorenz, and Matthias Hollick. Disrupting Continuity of Apple?s Wireless Ecosystem Security: New Tracking, DoS, and MitM Attacks on iOS and macOS Through Bluetooth Low Energy, AWDL, and Wi-Fi. In: USENIX Security. 2022 (pp. 5, 36).

[245] Milan Stute, Sashank Narain, Alex Mariotto, Alexander Heinrich, David Kreitschmann, Guevara Noubir, and Matthias Hollick. A Billion Open Interfaces for Eve and Mallory: MitM, DoS, and Tracking Attacks on iOS and macOS Through Apple Wireless Direct Link. In: USENIX Security. 2019 (pp. 5, 35).

[246] Ke Sun, Chunyu Xia, Songlin Xu, and Xinyu Zhang. Stealthy-IMU: Stealing permission-protected private information from smartphone voice assistant using zero-permission sensors. In: NDSS. 2023 (p. 32).

[247] Zhichuang Sun, Ruimin Sun, Long Lu, and Alan Mislove. Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps. In: USENIX Security. 2021 (p. 30).

[248] Suzanne Frey. A new layer of security for certified Android devices. 2025. URL: https://android-developers.googleblog.com/2025/08/elevating-android-security.html (p. 25).

[249] Mehdi Talbi and Etienne Helluy-Lafont. Paint It Blue: Attacking The Bluetooth Stack. 2025. URL: https://www.synacktiv.com/en/publications/paint-it-blue-attacking-the-bluetooth-stack (p. 4).

[250] Zi Fan Tan, Gulshan Singh, and Eugene Rodionov. Attacking Android Binder: Analysis and Exploitation of CVE-2023-20938. 2024. URL: https://androidoffsec.withgoogle.com/posts/attacking-android-binder-analysis-and-exploitation-of-cve-2023-20938/ (p. 4).

[251] Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan Van Schaik, Daniel Genkin, and Yuval Yarom. Hot pixels: Frequency, power, and temperature attacks on {GPUs} and arm {SoCs}. In: USENIX Security. 2023 (p. 32).

[252] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin R. B. Butler. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In: USENIX Security. 2018 (p. 34).

[253] Jianwen Tian, Wei Kong, Debin Gao, Tong Wang, Taotao Gu, Kefan Qiu, Zhi Wang, and Xiaohui Kuang. Density Boosts Everything: A One-stop Strategy for Improving Performance, Robustness, and Sustainability of Malware Detectors. In: NDSS. 2025 (p. 29).

[254] Dimitrios Valsamaras. Dirty stream attack: Discovering and mitigating a common vulnerability pattern in Android apps. 2024. URL: https://www.microsoft.com/en-us/security/blog/2024/05/01/dirty-stream-attack-discovering-and-mitigating-a-common-vulnerability-pattern-in-android-apps/ (p. 4).

[255] Tom Van Eyck, Stefan More, Florian Draschbacher, Sam Michiels, and Danny Hughes. Don't Stop Receiving: Ensuring Availability of Critical Services on Compromised Mobile Devices. In: NCA. 2025 (pp. 13, 32).

[256] Mathy Vanhoef. Fragment and Forge: Breaking Wi-Fi Through Frame Aggregation and Fragmentation. In: USENIX Security. 2021 (p. 33).

[257] Mathy Vanhoef and Frank Piessens. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In: CCS. 2017 (pp. 7, 33).

[258] Mathy Vanhoef and Frank Piessens. Release the Kraken: new KRACKs in the 802.11 Standard. In: CCS. 2018 (p. 33).

[259] Mathy Vanhoef and Eyal Ronen. Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd. In: S&P. 2020 (p. 33).

[260] Vivo. Funtouch OS 15. 2024. URL: https://www.vivo.com/eu/funtouch (p. 16).

[261] Parjanya Vyas, Haseeb Ur Rehman Faheem, Yousra Aafer, and N. Asokan. Ariadne: navigating through the labyrinth of data-driven customization inconsistencies in android. In: USENIX Security. 2025 (p. 30).

[262] Parjanya Vyas, Asim Waheed, Yousra Aafer, and N. Asokan. Auditing framework APIs via inferred app-side security specifications. In: USENIX Security. 2023 (p. 30).

[263] Alan Wang, Pranav Gopalkrishnan, Yingchen Wang, Christopher W Fletcher, Hovav Shacham, David Kohlbrenner, and Riccardo Paccagnella. Pixnapping: Bringing pixel stealing out of the stone age. In: CCS. 2025 (p. 32).

[264] Chao Wang, Yanjie Zhao, Jiapeng Deng, and Haoyu Wang. Born with a Silver Spoon: On the (In)Security of Native Granted App Privileges in Custom Android ROMs. In: S&P. 2025 (p. 31).

[265] Haoyu Wang, Hao Li, and Yao Guo. Understanding the Evolution of Mobile App Ecosystems: A Longitudinal Measurement Study of Google Play. In: WWW. 2019 (pp. 20, 39, 40).

[266] Haoyu Wang, Hao Li, Li Li, Yao Guo, and Guoai Xu. Why are android apps removed from google play? a large-scale empirical study. In: MSR. 2018 (p. 40).

[267] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond google play: A large-scale comparative study of chinese android app markets. In: Proceedings of the Internet Measurement Conference 2018. 2018 (p. 40).

[268] Jice Wang, Yue Xiao, Xueqiang Wang, Yuhong Nan, Luyi Xing, Xiaojing Liao, JinWei Dong, Nicolas Serrano, Haoran Lu, XiaoFeng Wang, et al. Understanding malicious cross-library data harvesting on android. In: USENIX Security. 2021 (p. 39).

[269] Jice Wang, Yue Xiao, Xueqiang Wang, Yuhong Nan, Luyi Xing, Xiaojing Liao, Jinwei Dong, Nicolás Serrano, Haoran Lu, XiaoFeng Wang, and Yuqing Zhang. Understanding Malicious Cross-library Data Harvesting on Android. In: 30th USENIX Security Symposium, USENIX Security. 2021 (p. 9).

[270] Kai Wang, Richard Mitev, Chen Yan, Xiaoyu Ji, Ahmad-Reza Sadeghi, and Wenyuan Xu. GhostTouch: Targeted Attacks on Touchscreens without Physical Touch. In: USENIX Security. 2022 (p. 32).

[271] Mona Wang, Jeffrey Knockel, Zoë Reichert, Prateek Mittal, and Jonathan Mayer. WireWatch: Measuring the security of proprietary network encryption in the global Android ecosystem. In: S&P. 2025 (p. 30).

[272] Xianbo Wang, Shangcheng Shi, Yikang Chen, and Wing Cheong Lau. PHYjacking: Physical Input Hijacking for Zero-Permission Authorization Attacks on Android. In: NDSS. 2022 (p. 31).

[273] Xueqiang Wang, Yifan Zhang, XiaoFeng Wang, Yan Jia, and Luyi Xing. Union under Duress: Understanding Hazards of Duplicate Resource Mismediation in Android Software Supply Chain. In: 32nd USENIX Security Symposium, USENIX Security. 2023 (pp. 9, 39).

[274] Yuanda Wang, Hanqing Guo, and Qiben Yan. GhostTalk: Interactive Attack on Smartphone Voice System Through Power Line. In: NDSS. 2022 (pp. 8, 32, 34).

[275] Zhirui Wang, Chengwei Zhang, and Guohui Zhong. Comprehensive Analysis of Security Risks in Android File Sharing Applications: Reverse Engineering, Methodologies, and Optimization. In: International Conference on Computing and Artificial Intelligence (ICCAI). 2025 (p. 37).

[276] Zihao Wang, Jiale Guan, XiaoFeng Wang, Wenhao Wang, Luyi Xing, and Fares Alharbi. The danger of minimum exposures: Understanding cross-app information leaks on ios through multi-side-channel learning. In: CCS. 2023 (p. 32).

[277] Ralf-Philipp Weinmann. The Baseband Apocalypse: All your baseband are belong to us. In: Chaos Computer Congress. 2010 (p. 7).

[278] Tim Willis. Multiple Internet to Baseband Remote Code Execution Vulnerabilities in Exynos Modems. 2023. URL: https://projectzero.google/2023/03/multiple-internet-to-baseband-remote-rce.html (p. 4).

[279] Jiangrong Wu, Yuhong Nan, Luyi Xing, Jiatao Cheng, Zimin Lin, Zibin Zheng, and Min Yang. Leaking the Privacy of Groups and More: Understanding Privacy Risks of Cross-App Content Sharing in Mobile Ecosystem. In: NDSS. 2024 (p. 30).

[280] Jianliang Wu, Patrick Traynor, Dongyan Xu, Dave (Jing) Tian, and Antonio Bianchi. Finding Traceability Attacks in the Bluetooth Low Energy Specification and Its Implementations. In: USENIX Security. 2024 (p. 34).

[281] Claud Xiao. Novel Malware XcodeGhost Modifies Xcode, Infects Apple iOS Apps and Hits App Store. 2015. URL: https://unit42.paloaltonetworks.com/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/ (p. 40).

[282] Yue Xiao, Zhengyi Li, Yue Qin, Xiaolong Bai, Jiale Guan, Xiaojing Liao, and Luyi Xing. Lalaine: measuring and characterizing non-compliance of apple privacy labels. In: USENIX Security. 2023 (p. 31).

[283] Yue Xiao, Chaoqi Zhang, Yue Qin, Fares Fahad S Alharbi, Luyi Xing, and Xiaojing Liao. Measuring Compliance Implications of Third-party Libraries' Privacy Label Disclosure Guidelines. In: USENIX Security. 2024 (p. 31).

[284] Xiaomi. HyperOS 3. 2025. URL: https://www.mi.com/global/hyperos (p. 16).

[285] Xiaomi. Xiaomi GetApps. URL: https://global.app.mi.com/ (p. 25).

[286] XploitBengineer. Exploiting CVE-2025-21479 on a Samsung S23. 2025. URL: https://xploitbengineer.github.io/CVE-2025-21479 (p. 4).

[287] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. BadBluetooth: Breaking android security mechanisms via malicious bluetooth peripherals. In: NDSS. 2019 (p. 34).

[288] Haichuan Xu, Mingxuan Yao, Runze Zhang, Mohamed Moustafa Dawoud, Jeman Park, and Brendan Saltaformaggio. DVa: Extracting Victims and Abuse Vectors from Android Accessibility Malware. In: USENIX Security. 2024 (p. 29).

[289] Haichuan Xu, Runze Zhang, Mingxuan Yao, David Oygenblik, Yizhi Huang, Jeman Park, and Brendan Saltaformaggio. Lock the Door But Keep the Window Open: Extracting App-Protected Accessibility Information from Browser-Rendered Websites. In: CCS. 2025 (p. 31).

[290] Lei Xue, Hao Zhou, Xiapu Luo, Yajin Zhou, Yang Shi, Guofei Gu, Fengwei Zhang, and Man Ho Au. Happer: Unpacking Android Apps via a Hardware-Assisted Approach. In: S&P. 2021 (p. 29).

[291] Guanhua Yan and Stephan Eidenbenz. Bluetooth Worms: Models, Dynamics, and Defense Implications. In: ACSAC. 2006 (p. 7).

[292] Qing Yang, Paolo Gasti, Gang Zhou, Aydin Farajidavar, and Kiran S. Balagani. On Inferring Browsing Activity on Smartphones via USB Power Analysis Side-Channel. In: IEEE Transactions on Information Forensics and Security 12.5 (2017), pp. 1056–1066 (p. 8).

[293] Shishuai Yang, Guangdong Bai, Ruoyan Lin, Jialong Guo, and Wenrui Diao. Beyond the horizon: Exploring cross-market security discrepancies in parallel Android apps. In: ISSRE. 2024 (p. 41).

[294] Yuqing Yang, Mohamed Elsabagh, Chaoshun Zuo, Ryan Johnson, Angelos Stavrou, and Zhiqiang Lin. Detecting and Measuring Misconfigured Manifests in Android Apps. In: CCS. 2022 (pp. 45, 46).

[295] Yuqing Yang, Yue Zhang, and Zhiqiang Lin. Cross Miniapp Request Forgery: Root Causes, Attacks, and Vulnerability Detection. In: CCS. 2022 (p. 30).

[296] Zhi Yang, Zhanhui Yuan, Shuyuan Jin, Xingyuan Chen, Lei Sun, Xuehui Du, Wenfa Li, and Hongqi Zhang. FSAFlow: Lightweight and Fast Dynamic Path Tracking and Control for Privacy Protection on Android Using Hybrid Analysis with State-Reduction Strategy. In: S&P. 2022 (p. 30).

[297] Chang Yue, Kai Chen, Zhixiu Guo, Jun Dai, Xiaoyan Sun, and Yi Yang. What's Done Is Not What's Claimed: Detecting and Interpreting Inconsistencies in App Behaviors. In: NDSS. 2025 (p. 30).

[298] Zihao Zhan, Yirui Yang, Haoqi Shan, Hanqiu Wang, Yier Jin, and Shuo Wang. VoltSchemer: use voltage noise to manipulate your wireless charger. In: USENIX Security. 2024 (p. 32).

[299] Chennan Zhang, Shuang Li, Wenrui Diao, and Shanqing Guo. PITracker: Detecting Android PendingIntent Vulnerabilities through Intent Flow Analysis. In: WiSec. 2022 (p. 6).

[300] Lei Zhang, Keke Lian, Haoyu Xiao, Zhibo Zhang, Peng Liu, Yuan Zhang, Min Yang, and Haixin Duan. Exploit the last straw that breaks android systems. In: S&P. 2022 (p. 31).

[301] Runze Zhang, Mingxuan Yao, Haichuan Xu, Omar Alrawi, Jeman Park, and Brendan Saltaformaggio. Hitchhiking Vaccine: Enhancing Botnet Remediation With Remote Code Deployment Reuse. In: NDSS. 2025 (p. 29).

[302] Xiao Zhang, Amit Ahlawat, and Wenliang Du. AFrame: isolating advertisements from mobile applications in Android. In: ACSAC. 2013 (p. 39).

[303] Xin Zhang, Xiaohan Zhang, Zhichen Liu, Bo Zhao, Zhemin Yang, and Min Yang. An Empirical Study on Fingerprint API Misuse with Lifecycle Analysis in Real-world Android Apps. In: NDSS. 2025 (pp. 6, 45).

[304] Yinyuan Zhang, Cuiying Gao, Yueming Wu, Shihan Dou, Cong Wu, Ying Zhang, Wei Yuan, and Yang Liu. Fighting fire with fire: continuous attack for adversarial android malware detection. In: USENIX Security. 2025 (p. 29).

[305] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. Breaking Secure Pairing of Bluetooth Low Energy Using Downgrade Attacks. In: USENIX Security. 2020 (p. 34).

[306] Yue Zhang, Yuqing Yang, and Zhiqiang Lin. Don't Leak Your Keys: Understanding, Measuring, and Exploiting the AppSecret Leaks in Mini-Programs. In: CCS. 2023 (p. 30).

[307] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In: USENIX Security. 2021 (p. 32).

[308] Zhuowei Zhang. Proof-of-concept for CVE-2025-27363 that crashes FreeType 2.13.0. 2025. URL: https://github.com/zhuowei/CVE-2025-27363-proof-of-concept (p. 4).

[309] Zicheng Zhang, Wenrui Diao, Chengyu Hu, Shanqing Guo, Chaoshun Zuo, and Li Li. An empirical study of potentially malicious third-party libraries in android apps. In: WiSec. 2020 (pp. 39, 40).

[310] Zidong Zhang, Qinsheng Hou, Lingyun Ying, Wenrui Diao, Yacong Gu, Rui Li, Shanqing Guo, and Haixin Duan. MiniCAT: Understanding and Detecting Cross-Page Request Forgery Vulnerabilities in Mini-Programs. In: CCS. 2024 (p. 30).

[311] Min Zheng, Xiaolong Bai, Yajin Zhou, Chao Zhang, and Fuping Qu. POP and PUSH: Demystifying and Defending against (Mach) Port-oriented Programming. In: NDSS. 2021 (p. 32).

[312] Hao Zhou, Xiapu Luo, Haoyu Wang, and Haipeng Cai. Uncovering Intent based Leak of Sensitive Data in Android Framework. In: CCS. 2022 (p. 30).

[313] Hao Zhou, Haoyu Wang, Xiapu Luo, Ting Chen, Yajin Zhou, and Ting Wang. Uncovering cross-context inconsistent access control enforcement in android. In: NDSS. 2022 (p. 30).

[314] Hao Zhou, Shuohan Wu, Chenxiong Qian, Xiapu Luo, Haipeng Cai, and Chao Zhang. Beyond the Surface: Uncovering the Unprotected Components of Android Against Overlay Attack. In: NDSS. 2024 (p. 31).

Part II.

Publications

List of Publications

In total, I contributed to 8 publications in scientific conference proceedings, 3 of which were published at top-tier conferences. Of my 5 first-author papers, one was published at a top-tier conference, 4 at conferences ranked Core-A and 1 at a conference ranked Core-B, where it earned the best paper award.

Publications in this Thesis

1. **Florian Draschbacher**, Lukas Maar, Lorenz Schumm, Rene Denifl, Treffner Lukas, and Stefan Mangard. Clone2Pwn: A Systematic Security Analysis of Data Migration Tools in the Android Ecosystem. In: ESORICS. 2026.

2. **Florian Draschbacher**, Lukas Maar, Mathias Oberhuber, and Stefan Mangard. ChoiceJacking: Compromising Mobile Devices through Malicious Chargers like a Decade Ago. In: USENIX Security. 2025.

3. **Florian Draschbacher** and Lukas Maar. Manifest Problems: Analyzing Code Transparency for Android Application Bundles. In: ACSAC. 2024.

4. **Florian Draschbacher**. A2P2 - An Android Application Patching Pipeline Based On Generic Changesets. In: ARES. 2023.

5. **Florian Draschbacher** and Johannes Feichtner. CryptoShield: Automatic On-Device Mitigation for Crypto API Misuse in Android Applications. In: AsiaCCS. 2023.

Other Contributions

1. Lukas Maar, **Florian Draschbacher**, Lorenz Schumm, Ernesto Martínez García, and Stefan Mangard. The Doom of Device Drivers: Your Android Device (most likely) has N-day Kernel Vulnerabilities. In: USENIX Security. 2025.

2. Tom Van Eyck, Stefan More, **Florian Draschbacher**, Sam Michiels, and Danny Hughes. Don't Stop Receiving: Ensuring Availability of Critical Services on Compromised Mobile Devices. In: NCA. 2025.

3. Lukas Maar, **Florian Draschbacher**, Lukas Lamster, and Stefan Mangard. Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels. In: USENIX Security. 2024.

5

ChoiceJacking: Compromising Mobile Devices through Malicious Chargers like a Decade ago

Publication Data

Florian Draschbacher, Lukas Maar, Mathias Oberhuber, and Stefan Mangard. ChoiceJacking: Compromising Mobile Devices through Malicious Chargers like a Decade Ago. In: USENIX Security. 2025

Contributions

The author of this thesis is the main author of the paper. He came up with the general attack idea and all three attack techniques, and implemented the attacks against real-world state-of-the-art devices. For the power-line side-channel, the author's contributions are the design and implementation of the malicious charger platform including the power measurement circuitry, as well as the Python interfacing. The author also wrote most of the paper's text.

ChoiceJacking: Compromising Mobile Devices through Malicious Chargers like a Decade ago

Florian Draschbacher Lukas Maar Mathias Oberhuber
Stefan Mangard

Graz University of Technology

Abstract

JuiceJacking is an attack in which malicious chargers compromise connected mobile devices. Shortly after the attack was discovered about a decade ago, mobile OSs introduced user prompts for confirming data connections from a USB host to a mobile device. Since the introduction of this countermeasure, no new USB-based attacks with comparable impact have been found.

In this paper, we present a novel family of USB-based attacks on mobile devices, ChoiceJacking, which is the first to bypass existing JuiceJacking mitigations. We observe that these mitigations assume that an attacker cannot inject input events while establishing a data connection. However, we show that this assumption does not hold in practice. We present a platform-agnostic attack principle and three concrete attack techniques for Android and iOS that allow a malicious charger to autonomously spoof user input to enable its own data connection. Our evaluation using a custom cheap malicious charger design reveals an alarming state of USB security on mobile platforms. Despite vendor customizations in USB stacks, ChoiceJacking attacks gain access to sensitive user files (pictures, documents, app data) on all tested devices from 8 vendors including the top 6 by market share. For two vendors, our attacks allow file extraction from locked devices. For stealthily performing attacks that require an unlocked device, we use a power line side-channel to detect suitable moments, i.e., when the user does not notice visual artifacts.

We responsibly disclosed all findings to affected vendors. All but one (including Google, Samsung, Xiaomi, and Apple) acknowledged our attacks and are in the process of integrating mitigations.

1. Introduction

Mobile devices such as smartphones and tablets have grown into indispensable everyday companions, storing sensitive user data such as private pictures or documents. Until about a decade ago, a significant threat to these devices were JuiceJacking attacks [17, 20]. These attacks exploit the fact that mobile phones use a single physical connector for charging and for data exchange. A malicious party could hide a computer in a charger to stealthily establish a data connection during charging. Since all USB connections were trusted by default, the attacker could access arbitrary files stored on the device or gain code execution without the user's knowledge.

Mobile platform developers including Google, its Android partners and Apple quickly recognized the threat posed by JuiceJacking attacks. As a mitigation, modern versions of Android and iOS require user consent[1] before a *data connection from a computer* can be established. However, to date, no such requirement has been added for USB connections from a mobile device to *peripherals and accessories* such as keyboards or mice. Most USB-based attacks presented after the introduction of JuiceJacking countermeasures, therefore, act as malicious peripherals. Nohl et al. [30] proposed BadUSB, which reprograms the firmware of USB peripherals, such as thumb drives, to inject input events. However, as an input device, the approach requires an additional channel for extracting data. Meng et al. [24, 25] suggested a malicious charger that contains a USB-to-HDMI interface for accessing the screen content of mobile devices. Although this setup allows data extraction, it can only access the content visible on screen. Wang et al. [51] hide an audio interface in a malicious charger, but are only able to extract data accessible through the voice assistant. Some researchers exploit physical side-channels of cable connections to extract data from charging devices. However, passive side-channels can only extract information from events triggered by the user [7, 53], and active side-channels require detailed prior knowledge about the victim [13, 51]. In summary, none of the attacks published since the introduction of JuiceJacking mitigations matches the impact of JuiceJacking attacks. Additionally, none so far explored the potential of combining characteristics of malicious USB hosts and malicious USB devices (e.g., peripherals).

[1] In this paper, we use the term "consent" to refer exclusively to technical means that enforce a user's intent

In this paper, we present a novel attack family, ChoiceJacking, which has an impact comparable to JuiceJacking attacks, but works on modern mobile devices. It introduces the concept of hybrid USB attacks that combine aspects of both host-based and device-based attacks on mobile devices. ChoiceJacking attacks operate under the same threat model and attacker goals as JuiceJacking attacks. They leverage malicious chargers to gain *file access or code execution* on mobile devices. While we confirm the attack principle on iOS as well, this paper focuses on ChoiceJacking attacks for variants of Android, the most popular mobile platform.

We base our work on a fundamental flaw in the JuiceJacking mitigations implemented in major mobile platforms. The core idea of these mitigations is to require explicit user confirmation before a USB host can establish a data connection. This user confirmation is implemented by requiring certain user interface interactions, e.g., confirming a user prompt that explains what access is exposed. This countermeasure works under the assumption that a USB host cannot inject input events to autonomously confirm the user prompt. When considering the USB protocol in isolation, this is a valid assumption. The protocol dictates that a USB port can only operate as either a *USB host* (e.g., a computer) or a *USB device* (e.g., a mouse or keyboard) at a given time. However, from a system perspective, this assumption does not hold on mobile platforms. We identify three concrete attack techniques for Android and iOS that allow breaking this assumption. All attack techniques combine aspects of both a malicious USB host and USB device. They enable the attacker to take control of the user interface while a USB user confirmation prompt is shown and to autonomously complete the confirmation.

The most effective attack technique exploits flaws in the USB-based Android Open Accessory Protocol (AOAP). This protocol enables an accessory to register as an input device despite operating as a USB host. It does not require any user consent. According to the AOAP specification, Android devices are only supposed to accept AOAP messages while in a special accessory mode that does not allow data connections. However, we find that modern real-world Android devices accept certain AOAP messages at all times, allowing injection of input events while establishing USB data connections. This allows a malicious charger to trigger the user consent prompt intended to mitigate JuiceJacking attacks and autonomously confirm it. In our evaluation, this technique allows gaining file access on all Android devices.

For our second attack technique, we identify a race condition in Android's input subsystem. Android still trusts *USB peripherals* by default, so that a malicious charger can act as an input device without requiring user consent. This allows a malicious charger to flood the input event queue with key events as a *USB peripheral*. When the charger then (through USB Power Delivery) switches its USB interface to operate as a *USB host* and trigger the user consent prompt, the Android OS is still processing the injected input events. If the attacker specifically crafts the input events used for flooding the queue, this technique allows bypassing the user confirmation prompt. It succeeds on 9 of 10 evaluated Android devices.

In the third attack technique, a malicious charger can act as a USB peripheral to make the mobile device establish a connection to a Bluetooth input device integrated in the charger. The charger can then switch its USB interface to operate as a USB host, which triggers the user consent prompt intended to mitigate JuiceJacking attacks. However, the malicious charger can use the connected Bluetooth input device to autonomously accept the user consent prompt. 10 of 11 evaluated devices across both Android and iOS are susceptible to this technique.

On most evaluated devices, gaining data access through ChoiceJacking attacks requires the victim device to be unlocked at some point during charging. Once this condition is met, the attack only takes in the range of milliseconds on many devices, which is merely a short flickering on the screen that may remain unnoticed. After this brief flash of visual artifacts, the attacker has stealthy file access or code execution on the victim device until the USB connection is removed. For two vendors (Honor and Oppo), ChoiceJacking attacks gain MTP file access on *locked devices*, by exploiting further implementation flaws in USB handling logic. On devices from Xiaomi, ChoiceJacking attacks gain ADB development access (code execution) even if they are not development-enabled before the attack. For enabling entirely stealthy ChoiceJacking attacks, we present a power line side-channel for detecting suitable attack times, e.g., when the device screen is unobserved during a phone call.

Contributions. The main contributions of our work are

(1) We uncover fundamental flaws in the JuiceJacking mitigations of mobile platforms that render them largely ineffective on state-of-the-art devices.

(2) We introduce the novel concept of USB attacks on mobile platforms that combine traits of host-based and device-based attacks. We pro-

vide three concrete attack techniques that instantiate this concept to exploit JuiceJacking mitigation flaws and establish PTP, MTP or ADB connections on Android or iOS without user consent.

(3) We demonstrate the effectiveness of our attacks on 11 current-generation mobile devices from 8 vendors. Additionally, we present a power line side-channel that allows attackers to identify suitable moments for ChoiceJacking attacks.

Disclosure. We reported 16 vendor-specific security issues to 6 Android vendors, 4 upstream Android vulnerabilities to Google, and one vulnerability to Apple. All but one vendors confirmed our findings and are in the process of developing or rolling out patches. Google and Samsung already assigned CVEs[2]. Google plans to fix the upstream design flaws in an upcoming Android release.

Outline. Section 2 provides background. Section 3 introduces the threat model and principle of ChoiceJacking attacks. We present our attack techniques in Section 4. In Section 5, we detail the prototype used for the evaluations in Section 6. Section 7 presents our power line side-channel for detecting suitable attack moments. Section 8 discusses susceptible platforms, existing mitigations and related work, before Section 9 concludes this paper.

2. Background

This section briefly introduces USB and the Android OS, which this paper focuses on.

2.1. USB, USB Type-C and USB PD

The Universal Serial Bus (USB) allows connecting a large variety of different hardware peripherals to a computer through a single connector and low-level communication protocol. Since its first publication in 1996 [6], the standard has been widely adopted in all sorts of electronic devices. Fundamentally, USB connections on a logical level are always formed between a single USB host and a USB device. All communication is controlled by the USB host. The USB device may only transmit data to the host after the latter issues a corresponding request. Usually, the USB

[2] CVE-2024-43085 and CVE-2024-20900, respectively

host is a computer of some form, while the USB device is a peripheral such as a mouse or keyboard. Once a physical connection is established, the USB host queries the USB device for its USB descriptors. These are data structures that describe the supported functionality of the device, as well as metadata. This information allows the host computer to identify the attached peripheral and load suitable drivers. Originally, the role a communication partner assumed in a USB connection was hardcoded in the connector type it featured. Host devices used a USB Type-A connector that differed in shape from the Type-B connector used on the device side.

USB Type-C and USB Power Delivery. The Type-C connector [48] was added to the USB ecosystem to solve multiple issues with the previous Type-A and Type-B connector families. It features a dedicated Configuration Channel (CC) line that communication partners may use to advertise their port as either Upward Facing Port (UFP; what used to be a USB device port) or Downward Facing Port (DFP; USB host) through specific resistor configurations. Additionally, more complex role arrangements can be negotiated by exchanging USB Power Delivery (USB PD) [2] messages over the CC line. USB PD introduces a distinction between a port's power role (source or sink) and its data role (UFP or DFP). The communication partners advertise their capabilities as part of an initial PD handshake. On connection establishment, the DFP (as identified through resistors) always starts out as the power source and USB host. Later on, both partners can dynamically request power or data role swaps. For example, this enables the implementation of USB hubs that either act as power source or sink depending on whether their power supply is connected.

2.2. Android

With a market share of over 71 %[3], Android is the dominating mobile operating system. It is developed in an open-source fashion in the Android Open Source Project (AOSP) led by Google. The OS consists of a Linux kernel and a purpose-built userspace under the Apache 2.0 license. As a result of this licensing arrangement, device vendors may adapt large parts of the operating system. Almost all major Android phone manufacturers try to differentiate their products by shipping customized Android variants. These customizations have led to numerous security vulnerabilities in the past [22, 36, 43].

[3] As of Q1 2024: `https://gs.statcounter.com/os-market-share/mobile/worldwide`

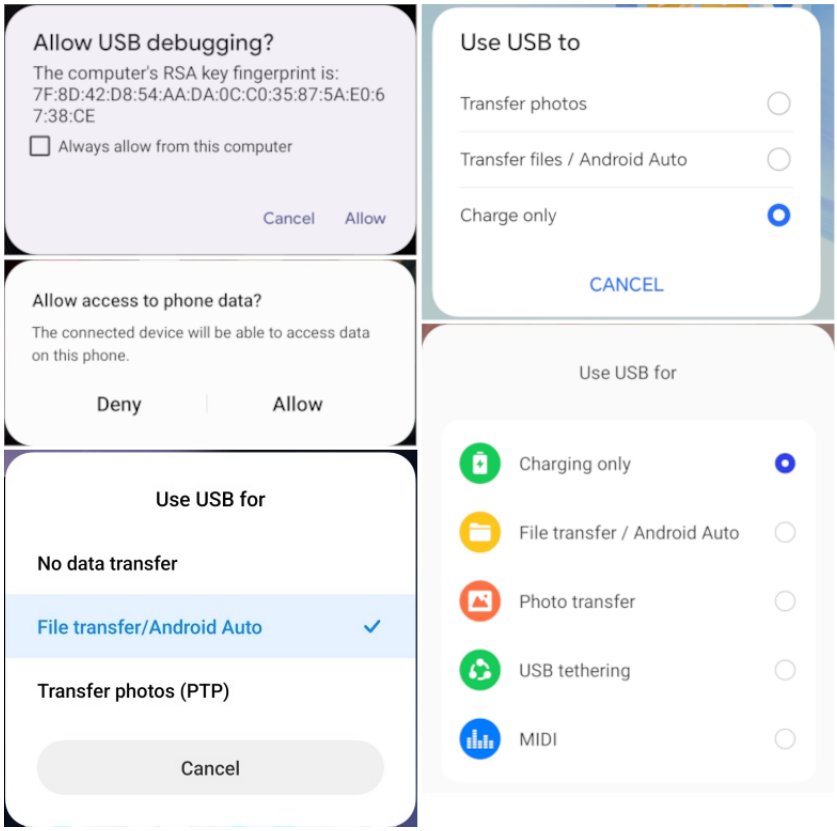


Figure 5.1: USB dialogs displayed by different Android variants

Modern Android devices feature a PD-enabled USB Type-C port that supports acting as DFP and UFP. When acting as a DFP, Android supports USB Human Interface Device (HID) class devices such as USB mice and keyboards. When acting as a UFP, Android exposes interfaces for exchanging files and for development access.

USB File Access. File exchange works through the Picture Transfer Protocol (PTP) for transferring photos and videos and the Media Transfer Protocol (MTP), an extension of PTP that enables transferring files of arbitrary formats. For every single USB connection that wishes to use any of these protocols, Android requires explicit user consent. Depending on the protocol and the vendor-specific Android variant, this works either through confirming a dialog or through changing a system setting.

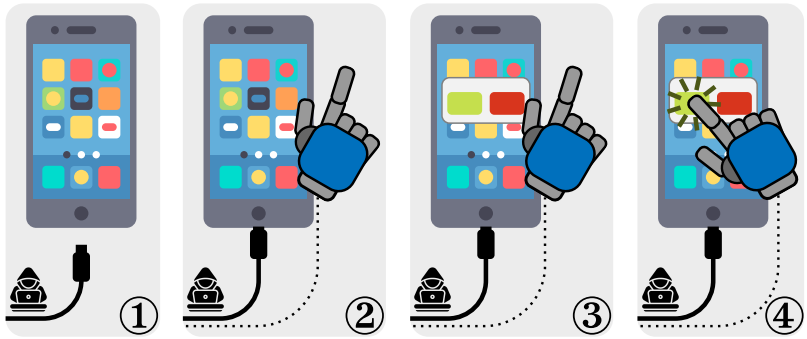


Figure 5.2: Principle of ChoiceJacking attacks. ① Victim device is attached to the malicious charger. ② The charger establishes an extra input channel. ③ The charger initiates a data connection. User consent is needed to confirm it. ④ The charger uses the input channel to spoof user consent.

USB Development Access. On Android, development access is implemented through the Android Debug Bridge (ADB) protocol and daemon. It grants the USB host shell access to the device, which can be used for installing apps, accessing the file system or executing arbitrary binaries. Using the ADB development access interface involves a three-staged setup procedure. As a first step, a hidden development system settings menu needs to be unlocked. This step is permanent and only needs to be carried out once, but requires user authentication, e.g., by presenting a valid unlock pin. Through the now unlocked development settings menu, it is possible to change configurations that lower the system's security. For example, it allows changing the default USB mode, so that the user consent described above is not needed for every MTP or PTP connection. As the second step for enabling ADB access, the user needs to enable USB debugging in the unlocked development settings menu. As the third and final step, Android requires user confirmation through a UI dialog to establish trust in a specific computer before it can interact with the ADB interface. When acting as a UFP, Android also supports platform-specific accessories that use the Android Open Accessory Protocol (AOAP) [1]. Like standard USB peripherals, AOAP accessories are trusted by default and may implement HID functionality. Still, as a major difference to standard USB peripherals, AOAP accessories act as USB hosts.

3. Threat Model and Attack Principle

In this section, we describe our threat model, which largely aligns with prior work [20, 45, 51], including the one from JuiceJacking. Subsequently, we discuss the flawed design of JuiceJacking mitigations and the principle of our ChoiceJacking attacks that bypass them.

3.1. Threat Model

Our threat model largely aligns with that of JuiceJacking attacks and various previous USB-based attacks on mobile devices [20, 45, 51]. We assume a scenario in which the victim uses phone charging infrastructure that has been tampered with by an attacker. Many public spaces offer mobile device charging possibilities that are anonymously shared between a large number of people. Examples are phone charging stations at airports or cafes, rentable power banks at museums, or USB wall sockets in hotel rooms. In our threat model, manipulations entail replacing the electronics in the charger or reprogramming the firmware of existing electronics. The goal of the attacker is to extract sensitive information from a mobile device that is connected to the malicious charger. Since the malicious charger integrates a wireless module (e.g., Wifi), the attacker does not need to be physically present for carrying out ChoiceJacking attacks. While some ChoiceJacking attacks require an unlocked screen, we alleviate this constraint through the power line side-channel in Section 7.

ChoiceJacking attacks are untargeted. An attacker can place malicious chargers in public places to harm as many victims as possible. As part of this untargeted setup, all information needed to carry out the attack is collected from the device once it is connected to the malicious charger. The attack will succeed on all susceptible charging devices (i.e., running Android or iOS and, for some vendors, unlocked while charging), and will thereby cause considerable damage to a large number of people over time. ChoiceJacking represents a primitive for obtaining access to users' private files stored on the mobile device. These files commonly include sensitive information such as private pictures, documents or app data. Through leaking this data, ChoiceJacking attacks can have severe consequences for affected invidiuals. Depending on the device configuration, ChoiceJacking attacks can also gain permanent code execution on the mobile device.

3.2. Design Flaws in JuiceJacking Mitigations

We observe that integrating user interaction as a countermeasure for JuiceJacking fundamentally conflicts with mobile platforms' default trust in *USB peripherals*. Mobile devices by default trust accessories and input devices connected to their USB port. However, at the same time, mobile OS developers introduced the requirement for explicit user consent for data connections to *USB hosts* as a countermeasure against JuiceJacking. On some Android variants, and also on iOS, a dialog informs users about the requested data access and offers the possibility to deny the connection. On other Android variants such as AOSP, the user has to manually change the USB mode in the system settings to enable data communication over the connection to the USB host. In all cases, when working as designed, this mitigation entails that (1) *the screen is unlocked*, and (2) *some UI interaction is performed* that explicitly confirms the data connection.

Screen Unlock. Before user consent can be expressed through the user interface, mobile platforms require the screen to be unlocked. On properly configured systems, unlocking the screen requires user authentication, for example through entering an unlock pin or presenting a fingerprint. The requirement to unlock the screen is therefore intended to ensure that all subsequent actions on the device are carried out by its legitimate owner. However, in the context of charger-based attacks, this guarantee does not necessarily hold. Users frequently unlock their device during charging, e.g., to make phone calls or look up information online. Prior work exploits this fact in malicious chargers to extract the screen unlock pin [7], observe web browsing activity [53] or read user input [24] of charging mobile devices. A malicious charger can also use this opportunity to stealthily act as an input device and interact with the phone's UI. This works because mobile platforms still trust USB *peripherals and accessories* by default, as described in Section 2. We conclude that not all UI interaction carried out while the device is unlocked and connected to a charger is necessarily caused by or known to the user. This voids the first part of the JuiceJacking mitigations already on a design level. Further flaws in vendor-specific Android variants additionally void it on the implementation level. These allow attacks that entirely bypass the requirement for the screen to be unlocked prior to USB data access.

UI Interaction. The protection therefore hinges on the fact that UI interaction is required for activating a data connection to a USB host. This measure is intended to ensure that the user is aware of potential

consequences and consciously making the choice to enable the data connection. It assumes that a malicious charger cannot operate as a USB host and as an input device at the same time. Otherwise, the attacker could autonomously carry out the user interaction needed to confirm its own data connection. This assumption is based on limitations of the USB protocol, which only allows a USB port to either act as a USB host or a USB peripheral at one point in time. When considering USB communication in isolation, an attacker would therefore have to decide on one of these two roles. They could either establish the USB data connection as a USB host or carry out user interaction as a USB peripheral. However, we show that when considering a holistic view on mobile platforms, there are multiple options for a malicious charger to *conceptually* hold both these roles at the same time. This is the basis for ChoiceJacking attacks.

3.3. Attack Principle

ChoiceJacking attacks are the first USB attacks to combine aspects of host-based and device-based attacks. In all ChoiceJacking attacks, the attacker compromises mobile devices through a malicious charger. To hide its malicious intent, the charger initially behaves like a normal phone charger. Once it detects a suitable moment (e.g., through the power line side-channel described in Section 7), it mounts an attack. As a common pattern for all ChoiceJacking attacks, the malicious charger issues input events while initiating a data connection to the victim device as a USB host. Conceptually, this works by exploiting initial USB access to establish a second channel to the victim device. This second channel can subsequently be used for injecting input events while the primary USB channel operates as a USB host initiating a data connection. Figure 5.2 illustrates this principle. It works on all major mobile platforms.

We present three concrete attack techniques that instantiate this novel attack principle. While the presented techniques focus on the Android platform, one of them works on iOS as well. The first technique uses implementation flaws in the Android Open Accessory Protocol (AOAP) as its (second) input channel. The second technique exploits a race condition in the Android input subsystem as its conceptual input channel. The third technique uses initial USB peripheral access to establish a BT input device connection as a second input channel. Additionally, we present a power line side-channel that allows a malicious charger to detect suitable attack moments.

Two of our attack techniques require switching USB roles, which a malicious charger can accomplish through USB Power Delivery. For determining the most effective ChoiceJacking attack for a specific device, an attacker can use the device information exposed through USB descriptors and USB PD Discover Identity messages. As described in Section 2.1, a USB device sends its USB device descriptor to the USB host as part of the initial handshake. Among other data, the descriptor contains information on the device's vendor, product, and version, as well as its unique serial number. This information can also be obtained before the USB handshake, by using the USB PD Discover Identity mechanism. Alternatively, timing or power side-channel based fingerprinting approaches as presented by Bates et al. [5] and Spolaor et al. [38] can be used for accurately identifying devices.

3.4. Attack Impact

Through a ChoiceJacking attack, the malicious charger either gains access to the victim device's PTP/MTP file transfer interface or its ADB development interface. On Android, both interfaces permit read/write control over the entirety of the victim device's public storage. Gaining MTP file transfer access does not pose any special requirements on the victim device. In other words, virtually all Android devices are affected. Most intuitively, an attacker can exploit this access to extract personal data such as pictures, videos, voice recordings, downloads or documents. Prior research has also shown that applications commonly store sensitive app-specific data in public storage. As a result, access to public storage could in the past be exploited for e.g., tampering over-the-air updates [21], gaining code execution in privileged processes [21, 23] or mounting DoS attacks against apps [23]. On devices that are development-enabled (see Section 2.2), ChoiceJacking additionally grants a malicious charger shell access to the victim device. Among others, this allows an attacker to gain persistent code execution. On iOS, ChoiceJacking attacks gain access to the victim device's pictures and videos.

4. Attack Techniques

We identify three concrete attack techniques, T1 through T3, that exploit the fundamental flaws in JuiceJacking mitigations described in Section 3.

In a novel hybrid form, they combine aspects of malicious USB hosts and malicious USB devices. They allow a malicious charger to bypass JuiceJacking mitigations for access to either PTP/MTP or ADB. While the described attack techniques focus on Android, T3 is platform-agnostic and also works on iOS. As described in Section 3, the malicious charger can use information obtained from initial communication to identify the most suitable attack for a victim device.

4.1. T1: Input Accessory

By sending a particular USB control request, the AOAP protocol described in Section 2.2 allows a USB host to put an attached Android device into a special accessory mode. In this accessory mode, the USB port is no longer available to other USB interfaces such as MTP. The USB host can then send subsequent USB control requests to register as a HID device and inject HID input events. However, we find that all investigated Android devices violate the AOAP specification by accepting AOAP HID messages while not in accessory mode. This means that a USB host can inject input events while initiating an ADB or MTP connection. A malicious charger can exploit this implementation flaw to autonomously complete all user confirmations for these data connections. Usually, AOAP accessories can only charge the device with 500 mA, which slows down charging noticeably [1]. For comparison, modern phone chargers supply at least 2 400 mA. To work around this limitation, the attacker can carry out USB Power Delivery power negotiation before sending AOAP messages. The full steps for this attack technique are:

1. The unlocked victim device is connected to the charger.
2. The charger registers a HID input device through AOAP.
3. The malicious charger initiates an MTP or ADB data connection to trigger the user consent prompt.
4. The user consent prompt is shown on the device.
5. The charger autonomously accepts the user consent prompt by injecting the suitable HID events.

On some vendor-specific Android variants, further implementation flaws in AOAP can be exploited for gaining file access on locked devices. By triggering a fault in AOAP input event handling, it is possible to stealthily force re-initialization of the USB interface in MTP mode. Details about these attacks can be found in Section 6.

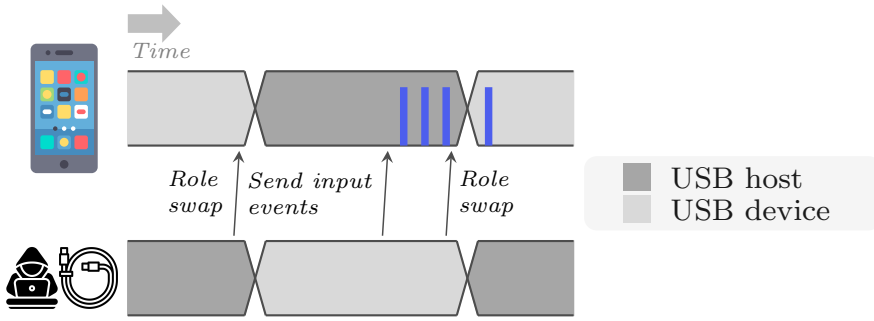


Figure 5.3: Race condition in Android's input dispatcher. A malicious charger can inject input events that are dispatched after its role swap to a USB host.

4.2. T2: Flooding Input Dispatcher

The second attack technique, T2, exploits a race condition in Android's input dispatcher for injecting input events while initiating a USB data connection. Android's input subsystem builds on its Linux counterpart. At the lowest level, Linux kernel drivers are responsible for interacting with connected input peripherals (e.g., USB HID devices) to obtain input events. The `InputReader` in the `InputManagerService` Android system service parses these Linux input protocol events and puts them into a queue for consumption by the input dispatcher. This queue acts as a buffer that ensures that input events are processed in order and are not lost even if the system is busy. We note that the Android input subsystem keeps input events in this queue even if the input device that generated them is no longer connected. Additionally, Android's input dispatcher intentionally serializes key events. It waits for all previous input events to have been fully processed before a key event is dispatched[4]. This means that a single process that performs overly complex logic in its key event handler will delay event dispatching for all other processes or global event handlers. We observe that such key event handlers are found in system processes or preinstalled applications on all evaluated Android devices. A malicious charger can exploit this by starting as a USB peripheral and flooding the event queue with a specially crafted sequence of key events. It then switches its USB interface to act as a USB host while the victim device is

[4] `InputDispatcher: https://android.googlesource.com/platform/frameworks/native/+/refs/heads/main/services/inputflinger/dispatcher/InputDispatcher.cpp#2217`

still busy dispatching the attacker's events. These events therefore accept user prompts for confirming the data connection to the malicious charger. The technique is illustrated in Figure 5.3. The exact steps are:

1. The unlocked device is connected to the charger.
2. At a suitable moment, the charger performs a USB PD Data Role Swap. The Android device now acts as a USB host, while the charger is a USB HID device.
3. The charger generates a large number of HID input events that are known to trigger complex handling logic and confirm the user prompt. Two options are possible for arranging events that delay processing (delayers) and those that confirm the user prompt (confirmers):
   (a) *Alternating arrangement*. By alternating delayers and confirmers, the user prompt is accepted as soon as it is shown.
   (b) *Sequential arrangement*. Filling the queue purely with delayers and only adding confirmers at the end ensures that confirmers do not interfere with the user interface before the user prompt is shown.
4. The charger performs a USB PD Data Role Swap. It is now the USB host, the mobile device is the USB device.
5. As the USB host, the charger triggers the user prompt.
6. The input generated previously confirms the user prompt.

4.3. T3: Bluetooth Input Device

In the third attack technique, T3, the malicious charger acts as a USB peripheral to autonomously pair a Bluetooth (BT) HID device to the victim device. Modern BT chips have very small footprints and can be easily fitted into maliciously modified chargers. The malicious charger can use this BT input device to confirm the data connection initiated by switching its USB interface into USB host mode. For pairing the BT input device to the victim device, the charger injects events for (1) making the device discoverable, and (2) confirming the pairing dialog. On Android, making a device discoverable simply requires opening the BT pairing screen in the system settings. The full steps for this attack technique through a PD-capable and BT-capable malicious charger are therefore:

1. The victim device is connected to the malicious charger. The device has its screen unlocked.

2. At a suitable moment, the charger performs a USB PD Data Role (DR) Swap. The mobile device now acts as a USB host, the charger acts as a USB input device.
3. The charger generates input to ensure that BT is enabled.
4. The charger navigates to the BT pairing screen in the system settings to make the mobile device discoverable.
5. The charger starts advertising as a BT input device.
6. By constantly scanning for newly discoverable Bluetooth devices, the charger identifies the BT device address of the mobile device and initiates pairing.
7. Through the USB input device, the charger accepts the Yes/No pairing dialog appearing on the mobile device. The Bluetooth input device is now connected.
8. The charger sends another USB PD DR Swap. It is now the USB host, and the mobile device is the USB device.
9. As the USB host, the charger initiates a data connection.
10. Through the Bluetooth input device, the charger confirms its own data connection on the mobile device.

5. Proof-Of-Concept Implementation

ChoiceJacking attacks are mounted from a malicious phone charger. Since we consider reverse-engineering and modifying existing commercial charger products out of scope for this work, we simulate such an environment using a custom printed circuit board (PCB) and a Raspberry Pi single-board computer (SBC). The PCB operates the hardware-level components needed for the attacks, while the Raspberry Pi SBC runs the attack logic and controls the PCB. The interaction between the components is shown in Figure 5.4.

Custom PCB. Attack techniques T2 and T3 described in Section 4 require a malicious charger capable of (1) operating as both USB device and USB host, (2) providing power to the attached mobile device in both modes, and (3) switching the USB data role of both the charger and mobile device. Technique T1 only requires the charger to send AOAP messages as a USB host. It is worth noting that T1, therefore, can be carried out from any commodity hardware that supports USB host mode (e.g., off-the-shelf computers or MCUs). To fulfill the requirement for all attack techniques, we designed a custom PCB around the Raspberry Pi RP2040

microcontroller unit (MCU). During operation, the PCB is connected to the USB PD power supply, the mobile device, and the Raspberry Pi SBC. The MCU controls all components on the board. The chip's native USB interface implements a USB HID keyboard and mouse, while a second bit-banged USB port allows serial communication to the external Raspberry Pi SBC. The PCB also contains a USB Power Delivery controller that acts as a DFP to the mobile device. A USB multiplexer connects the mobile device's USB data lines to either the MCU or the Raspberry Pi SBC.

Raspberry Pi. The Raspberry Pi 4 SBC in our implementation serves three purposes. First, it communicates with the MCU on the PCB to control its hardware-level functionality. Second, it acts as the USB host that establishes the data connection to the mobile device. Third, it operates as a Bluetooth HID device required for attack technique T3.

Attack Costs. Overall, the cost of our prototype is less than 100 USD. This means ChoiceJacking attacks can be mounted even by unsophisticated attackers. Although larger in physical dimensions than an off-the-shelf charger, the prototype can be hidden behind a wall, e.g., behind the enclosure of a charging station. Alternatively, the electronics on our PCB can further be miniaturized so that they even fit into the USB-C connector. This allows to form a malicious charger simply by attaching a malicious cable to an otherwise benign USB-C socket. A similar miniaturization has been accomplished in commercial products for BadUSB attacks[5].

6. Evaluation

Using the implementation described in Section 5, we evaluate the susceptibility of mobile devices from the most popular vendors to ChoiceJacking attacks. While we focus our evaluation on different Android devices, we also include a device running iOS, the other major mobile platform. Since every Android manufacturer ships a customized OS variant, different Android devices are affected in different ways by our attacks. This section demonstrates that despite the diversity of tested devices, all are vulnerable to ChoiceJacking attacks.

Our selection of test devices is shown in Table 5.1. It includes a recent device from each of the 5 Android vendors with the highest worldwide market share as of May 2024 as reported by Statcounter [39]. These are

[5] O.MG Cable: `https://shop.hak5.org/products/omg-cable`

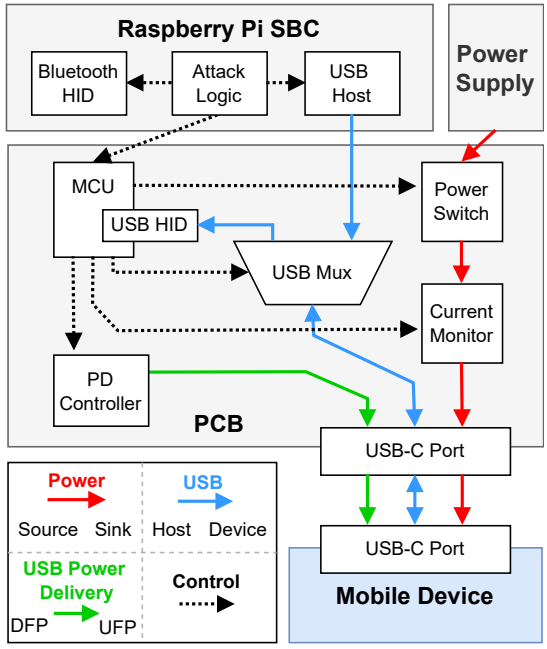


Figure 5.4: Our simulated malicious charger. Bidirectional USB lines represent the possibility for data role swaps.

Samsung, Xiaomi, Oppo, Vivo and Huawei. We additionally extend our evaluation to a recent device from Apple, the leader in mobile device market share. For Samsung, the leading Android manufacturer, we select 4 devices to include a range from low-end to high-end devices of one manufacturer. We also select a recent Pixel phone from Google, as these run a largely unmodified AOSP build. Finally, we also select a device from Honor to represent the long tail of smaller device manufacturers. For all of the covered devices, we evaluate on a recent available OS release (either Android 12, 13, or 14, and iOS 17).

For every device, we evaluate all attack combinations, i.e., using any of attack techniques T1 to T3 from Section 4 to gain MTP or ADB access. Since PTP is a strict subset of MTP, it was not evaluated separately for devices that support both.

We provide timings for the fastest attack technique per Android device and target protocol (ADB, MTP) in Figure 5.5. For all attacks, we measure the duration in which the malicious charger visibly interacts with the user interface of the victim device. This entails triggering UI changes e.g.,

Vendor	Device	OS	PD	Exploitable Attack Techniques		
				T1 Targets	T2 Targets	T3 Targets
Samsung	Galaxy A14	OneUI 5 (Android 13)	Yes	MTP, ADB	MTP, ADB	MTP, ADB
Samsung	Galaxy S20 FE	OneUI 5 (Android 13)	Yes	MTP, ADB	MTP, ADB	MTP, ADB
Samsung	Galaxy A33	OneUI 6 (Android 14)	Yes	MTP, ADB	MTP, ADB	MTP, ADB
Samsung	Galaxy S23	OneUI 6 (Android 14)	Yes	MTP, ADB	MTP	MTP, ADB
Xiaomi	12	MIUI 14 (Android 13)	Yes	MTP, ADB[a]	MTP, ADB[a]	MTP, ADB[a]
Vivo	Y36	Funtouch OS 13 (Android 13)	Yes[c]	MTP, ADB	-	-
Oppo	A58	ColorOS 13 (Android 13)	Yes	MTP[b], ADB	MTP	MTP
Huawei	nova 12i	EMUI 14 Android 12)	Yes	MTP, ADB	MTP, ADB	MTP, ADB
Honor	90 Lite	MagicOS 7.1 (Android 13)	Yes	MTP[b], ADB	MTP	MTP
Google	Pixel 7a	Android 14	Yes	MTP, ADB	ADB	MTP, ADB
Apple	iPad Pro 2022	iOS/iPadOS 17.4.1	Yes	-	-	PTP

[a] ADB access can be gained even if the device is not development-enabled before the attack
[b] Attack works on locked devices
[c] No Data Role Swaps

Table 5.1: Tested devices and their susceptibility to ChoiceJacking attacks based on our 3 different attack techniques.

through injected input events or through USB state changes. We measure timings by counting frames in video recordings taken at 30 frames per second and full HD resolution. In the following, we evaluate concrete ChoiceJacking attacks for each individual mobile device vendor.

6.1. Samsung

Samsung's OneUI Android variant by default acts as an MTP device when connected to a USB host. However, actual device storage contents only become visible once the user has confirmed a prompt displayed on the device following the MTP handshake from the host. The prompt can be confirmed by jointly pressing both hardware volume keys.

T1. Attack technique T1 succeeds on all Samsung devices to autonomously confirm Samsung's proprietary MTP prompt while the screen in unlocked. The attack takes 167 ms on the Galaxy A14, 133 ms on the Galaxy S20 FE, 267 ms on the Galaxy A33 and 300 ms on the Galaxy S23. For development-enabled devices, the ADB trust prompt can also be overcome via technique T1 within 233 ms, 200 ms, 233 ms and 267 ms on the Galaxy A14, S20 FE, A33 and S23, respectively.

T2. Attack technique T2 allows bypassing the MTP prompt. On Android 13 and earlier, volume key events delay the processing of subsequent input events and accept the user prompt. As a result, ChoiceJacking attacks can accept the MTP user prompt on the Galaxy A14 and S20 FE even before it becomes visible. Using T2 for gaining MTP access takes 17.5 s on average over the 4 Samsung devices. On development-enabled Galaxy A14, S20 FE and A33 devices, T2 also allows confirming the ADB trust prompt in less than 11 s. We could not find a suitable delayer sequence for the ADB trust prompt on the Galaxy S23.

T3. Technique T3 gains MTP access on an unlocked or ADB access on an unlocked, development-enabled Samsung device within 24.3 s and 25.3 s on average, respectively.

> **Takeaway:** All Samsung devices in our test set are susceptible to ChoiceJacking attacks for file access and code execution in *under* 0.3 s.

6.2. Xiaomi

Xiaomi's MIUI Android variant displays a proprietary dialog for selecting the USB mode for all connections to a USB host longer than a second. Xiaomi also heavily modified the developer settings on their OS variant. Enabling USB debugging effectively lowers an Android device's security, as explained in Section 2.2. To ensure that the user is consciously making the choice to enable USB debugging, its activation normally requires user authentication. However, Xiaomi entirely removed the authentication step from this procedure in their Android variant. Enabling the developer settings menu simply requires repeatedly tapping on the build version field in the system settings. Once enabled, the developer settings menu can already be used e.g., to change the default USB mode. For further activating USB debugging (ADB), the user needs to wait 10 s and accept a warning prompt.

T1. On the Xiaomi 12, attack technique T1 allows bypassing the USB mode selection dialog while the screen is unlocked to access MTP functionality. The attack takes 967 ms. If the device is development-enabled (USB debugging is activated), the same technique can also be used for confirming the ADB trust prompt in 933 ms. It is worth noting that attack technique T1 can also be used for gaining ADB access on an unlocked Xiaomi device that is not development-enabled. This works by using the AOAP input device to carry out the UI interaction for enabling development settings and USB debugging, before confirming the ADB trust dialog. Overall, this attack takes 36 s on the Xiaomi 12. The attack on a non-development-enabled device is comparatively slow, because it involves additional menu navigation and idle time caused by the warning prompt delay described above.

T2. Attack technique T2 is successful at gaining MTP or ADB access on an unlocked device. For MTP on an unlocked device, the attack takes 13 s and uses volume key events as delayers in a sequential arrangement. For ADB on a development-enabled and non-development-enabled device, the attack takes 27 s and 67 s, respectively. The difference in these timings can again be attributed to the additional steps needed on non-development-enabled devices.

T3. Attack technique T3 allows obtaining MTP or ADB access on an unlocked device. Reaching MTP access on an unlocked device takes 29 s. ADB access can also be reached on a development-enabled or non-development-enabled device within 23 s and 55 s, respectively. The attack

on the non-development-enabled device is slowed down by the additional steps described above.

> **Takeaway:** On the evaluated Xiaomi device, ChoiceJacking gains file access or code execution in under 1 s. Due to Xiaomi's OS customizations, ChoiceJacking attacks can gain code execution on unlocked *non-development-enabled* devices.

6.3. Vivo

Vivo's Funtouch OS Android variant does not display a user prompt for USB mode selection. Instead, the user can change USB mode by tapping on a particular system notification. Since this notification is always shown above all other notifications, it can be reliably selected and opened through keyboard inputs. It is worth noting that the entry-level Vivo Y36 device in our test set does not support data role switches through USB PD.

T1. Attack technique T1 on the Vivo Y36 allows to open the USB mode settings and enable MTP mode. This attack takes 1 267 ms. T1 can also gain ADB access on a development-enabled device within 367 ms.

T2 & T3. As the tested Vivo device only partially supports USB PD, it is not susceptible to attack techniques T2 and T3.

> **Takeaway:** ChoiceJacking attacks on the evaluated Vivo device gain file access or code execution in under 1.3 s.

6.4. Oppo

Oppo's ColorOS displays a user prompt for selecting the USB mode. The prompt is triggered by the USB host querying the device's USB descriptors immediately following USB cable attachment. It is worth noting that ColorOS disables USB debugging on every USB disconnect or PD data role swap. The attack cannot re-enable USB debugging, because the system settings UI does not accept keyboard input for the corresponding UI switch. Mouse input cannot be used because the screen position of the switch depends on the device configuration and screen orientation and is, therefore, impossible to predict.

T1. Attack technique T1 succeeds in gaining MTP access on the Oppo A58 both while the screen is locked and unlocked. On an unlocked screen, AOAP HID events are injected to autonomously select MTP in the USB mode user prompt within 667 ms. Gaining MTP access while the screen is locked can be accomplished by injecting an input event immediately after registering the HID device through AOAP. Because the OS takes some time to set up the registered HID device for use, an error in the AOAP implementation is raised. As part of error handling, the Android Auto service on the device (part of the AOAP implementation) interferes with Oppo's USB stack, resulting in MTP mode being enabled after about 35 s. Technique T1 also gains ADB access on unlocked development-enabled devices within 2 233 ms.

T2. Attack technique T2 can be exploited to bypass the USB mode user prompt for gaining MTP access on an unlocked Oppo device. The attack navigates to the launcher by injecting a home key press and uses the key event handler in the launcher application as a delayer in an alternating arrangement. Overall, the attack takes 14 s.

T3. Attack technique T3 allows access to MTP on the Oppo A58 within 27 s. Due to USB debugging being disabled automatically, we failed to exploit T2 and T3 for ADB access.

> **Takeaway:** ChoiceJacking attacks gain file access and code execution on the evaluated Oppo device in under 2.3 s. A bug in the implementation of accessory mode allows extracting files *while the screen is locked*.

6.5. Huawei

Huawei's EMUI Android variant displays a proprietary USB mode dialog for connections to a USB host that last longer than 2 seconds. The dialog is triggered by querying the device's USB descriptors. Among others, it offers the possibility to enable MTP access. It is worth noting that EMUI is not a licensed Android variant, so it does not undergo Google's rigourous compatibility testing. As a result, the OS's accessory mode is not fully compliant with Google's specification.

T1. Despite Huawei's AOAP implementation not being fully specification-compliant, EMUI accepts AOAP HID messages when not in accessory mode. As a result, the Huawei nova 12i is susceptible to ChoiceJacking

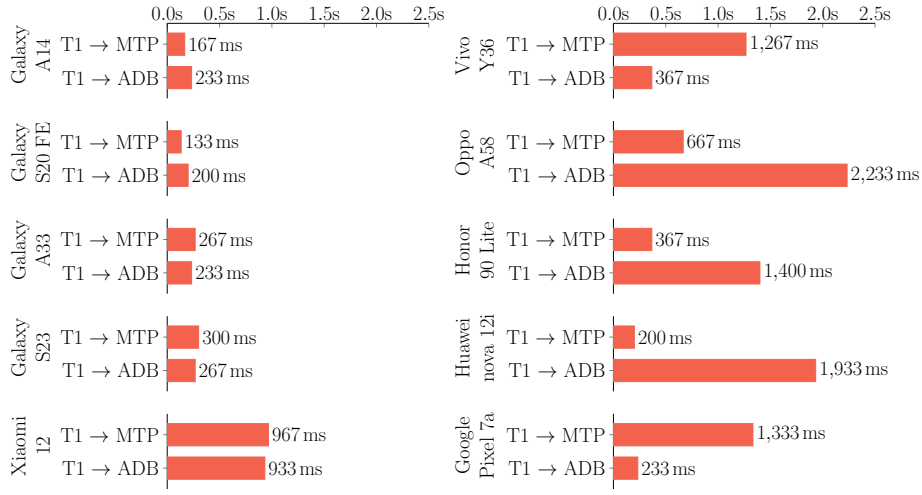


Figure 5.5: Timings for ChoiceJacking attacks on Android devices. In the best case, MTP file access is gained in 133 ms.

attacks for gaining MTP access while the screen is unlocked. This attack takes 200 ms to complete. Attack technique T1 can also be used for autonomously accepting the ADB trust dialog on a development-enabled device in 1 933 ms.

T2. This technique gains MTP access on an unlocked device and ADB access on an unlocked development-enabled device. These attacks take 20 s and 23 s for MTP and ADB, respectively. Volume key events are used as delayers in a sequential arrangement for MTP and a hybrid one for ADB.

T3. Via a Bluetooth HID connection, MTP and ADB access can be gained in 25 s and 27 s, respectively.

> **Takeaway:** The Huawei device is susceptible to ChoiceJacking attacks that gain file access and code execution in under 1.9 s.

6.6. Honor

In its MagicOS Android variant, Honor implemented a proprietary user prompt for choosing the USB mode. The prompt is triggered by querying the USB descriptors immediately following connection establishment as a USB device. Although the Android device presents as an MTP device to

the host immediately, the actual storage contents are only exposed once the user selects MTP in this USB mode prompt. Like Oppo's ColorOS, we observe that MagicOS automatically disables USB debugging after USB disconnects or PD data role swaps. Due to the same limitations in the system settings UI, it also cannot be re-enabled as part of the attack.

T1. The Android Open Accessory Protocol (AOAP) can be exploited to reach MTP access on the Honor 90 Lite independent of the screen lock state. While the screen is unlocked, the injected AOAP HID events select MTP in the displayed USB mode prompt. The attack completes in 367 ms. If the screen is locked, the same implementation flaw as found on the Oppo device can be exploited. It takes about 30 s for MTP access to be granted. Technique T1 also gains ADB access on an unlocked development-enabled device within 1 400 ms.

T2. Attack technique T2 can be exploited to bypass the USB mode user prompt to gain MTP access. Our attack uses the Google assistant shortcut as a delayer in an alternating arrangement. The attack briefly interrupts the device's power supply to trigger the USB mode dialog after the USB PD data role swap. Overall, the attack takes 28 s.

T3. Technique T3 allows gaining MTP file access. The Bluetooth pairing screen can be reached through the search bar in Honor's app launcher. Due to the USB debugging option being reset automatically, T2 and T3 cannot be exploited for ADB access.

> **Takeaway:** The evaluated Honor device is susceptible to ChoiceJacking attacks for file access and code execution within 1.4 s. A bug in the implementation of accessory mode allows extracting files *while the screen is locked*.

6.7. Google

Devices by manufacturer Google run unmodified AOSP Android. As a result, these devices do not display a user prompt when a connection to a USB host is established. To activate MTP mode, the user needs to open the USB mode selection screen either through the system settings or through a system notification and select the corresponding option. In contrast to Vivo's customized OS, AOSP does not permit keyboard input to reliably select the notification if other notifications are shown as well, because their order cannot be predicted.

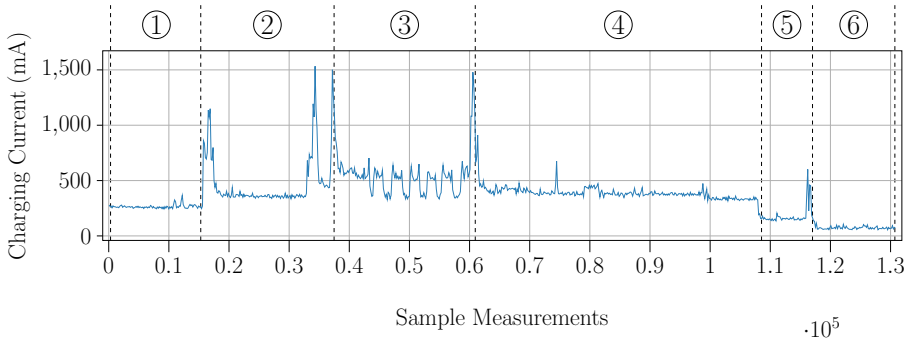


Figure 5.6: Power trace for phone call cycle on Xiaomi 12. ①: Screen Locked, ②: Screen Unlocked, ③: Call, ④: Screen Unlocked, ⑤: Screen Dimmed, ⑥: Screen Locked

T1. The Pixel 7a is susceptible to input events injected through AOAP for autonomously activating MTP within 1 333 ms. Since the exact position of the USB mode system notification among other notifications (e.g., from emails) is unpredictable, the attack launches the USB mode selection screen through the system settings. Technique T1 can also be used for confirming the ADB trust dialog within 233 ms.

T2. The race condition in Android's `InputDispatcher` can be exploited for gaining ADB access on a development-enabled device within 10 s. Our attack opens the preinstalled calculator app using a global key shortcut, then uses number key presses as delayers in an alternating arrangement.

T3. Attack technique T3 gains MTP access on an unlocked device and ADB access on an unlocked development-enabled device. Both attacks take 19 s.

> **Takeaway:** On the Google Pixel 7a, ChoiceJacking attacks can extract files or gain code execution in 1.3 s.

6.8. Summary of Results

Our evaluation highlights the scope of ChoiceJacking attacks. All tested devices across platforms and manufacturers are susceptible. On devices from Oppo and Honor (18 % of all tested devices), it is possible to exploit a flaw in the accessory mode implementation to gain file access on locked devices. On all Android devices, our attacks can gain MTP file access in

under 1.4 s, for a median of 333 ms over all evaluated Android devices. Code execution through ADB on all unlocked development-enabled Android devices can be reached in less than 1.3 s, for a median of 316 ms. On devices from Xiaomi, it is possible to gain code execution through ADB on an unlocked but non-development-enabled device.

Attack technique T1 proved best in terms of susceptible devices and attack speed. It succeeds on all evaluated Android devices. In the best case (Samsung Galaxy S20 FE), it only takes 133 ms, which is less than half the duration of a human blink [18]. At this speed, the attack only leaves a short flickering on the screen that may remain unnoticed. The median attack duration for T1 was 334 ms. Technique T3 exhibits the same success rate across platforms (91 %), but performs slower at a median of 24.5 s. This increase in duration can be attributed to the BT pairing process. At a median duration of 13 s, technique T2 only succeeded on 8 devices for MTP and 6 devices for ADB access. 9 devices (81 %) from 6 manufacturers (75 %) are susceptible to all 3 attack techniques for at least some protocol.

The results for the 4 Samsung devices indicate that attacks based on T1 work almost unchanged for different devices from the same vendor. Although the devices cover a range from low-end to high-end and span two major OS versions, the identical attack sequence works on all of them. This finding indicates that an attacker can target many different device models through a small number of different attack sequences.

7. Power Line Side-Channel

This section presents a power line side-channel (PLSC) for detecting a suitable moment to perform ChoiceJacking attacks, i.e., when the victim device is left unobserved.

The PLSC exploits the fact that any operation carried out on a mobile device creates unique fluctuations in its power consumption. A malicious charger that measures the charging device's power consumption can, therefore, learn details about its activity (e.g., user interactions). Prior work has demonstrated that PLSCs reveal various information, e.g., button-accurate screen tap locations [7] or web browsing activity [53]. The work by Cronin et al. [7] is particularly relevant for our PLSC since it infers touch locations from current fluctuations caused by screen content changes

as part of visual feedback. Their findings already show that such PLSCs exist on a wide range of mobile devices and are invariant to the battery charging level and various device configurations such as screen brightness. Screen content changes not only indicate user interaction but also allow learning about the user's focus of attention. Therefore, such measurements are a basis for detecting suitable moments for ChoiceJacking attacks.

7.1. Desired Detection Scenario

For operating stealthily, an attacker is interested in moments when a user does not notice the UI state changes incurred by ChoiceJacking attacks on an unlocked device. We observe that there are scenarios in which mobile devices are unlocked, yet the user is unable to observe the screen (see Section 8.1). Phone calls are particularly illustrative examples of such scenarios, so we use them for our proof-of-concept PLSC. During phone conversations, users typically hold their mobile device close to their face, so they can communicate through the integrated microphone and speaker. To conserve energy and prevent unintended touch events, most devices include proximity sensors that detect these events and turn off the screen. Crucially, while the screen is turned off, we notice that external input devices can still fully interact with the UI in this scenario. Furthermore, mobile phone calls usually take about two orders of magnitude longer than the 1.4 seconds ChoiceJacking attacks need at worst for gaining *file access* [32]. We therefore aim for a PLSC that detects when the screen is turned off during a phone call. We extend the findings of previous works [7, 53] and show that a PLSC can be used to detect this scenario.

7.2. Power Trace Analysis

As an initial indication for the existence of a suitable PLSC, we collect power traces using the malicious charger prototype described in Section 5. We base our experiments on the Xiaomi 12 from the test set described in Section 6, expecting similar results for other phones (as per prior work [7, 53]). The Xiaomi 12 was chosen because of the possibility for ChoiceJacking attacks to gain code execution *on non-development-enabled devices*. This represents the maximum attack gain among all evaluated ChoiceJacking attacks, yet it requires the device to be unattended for the longest time (36 s for technique T1). In line with prior work [7, 53], we deduce power from measuring current. The power trace collected at

1 kHz from the Xiaomi 12 is shown in Figure 5.6, illustrating the device's power consumption before, during and after a call. ① and ② show the consumption in the locked and unlocked states respectively before the call, while ⑤ and ④ show the consumption after the call. In the transition from ② and ③, the power traces indicate the possibility to distinguish the start of the call from other events, such as screen lock and unlock. During the call in ③, we repeatedly turned the screen off and on by triggering the device's proximity sensor. These events can be observed as visually distinguishable fluctuations in current draw between 600 mA and 400 mA in ③.

7.3. PLSC Evaluation

We implement and evaluate a neural network for automatically detecting the desired scenario from a malicious charger. This entails (1) detecting phone calls, i.e., call start and end events, as well as (2) screen on and off events. To demonstrate that the approach by Cronin et al. [7] can be adapted for detecting calls, we construct a similar one-dimensional convolutional neural network (CNN) classifier. The model distinguishes call start and end events from other events that commonly occur during mobile device usage. We consider 16 other common events (e.g., web browsing, doing calculations, messaging, ...). For training the model, we collected 20 power traces for each of the 18 events of 5 seconds each (360 traces total). Evaluating the resulting model on a test set of 10 fresh traces per event (180 traces total) yields an accuracy of 92.78 %. The resulting confusion matrix can be found in Figure 5.7. While our model does not report any false negatives for call start or end events, some other events are reported as call ends. The main goal for the PLSC is to avoid launching ChoiceJacking attacks while the user might notice them. False negatives (not starting the attack although it would be possible) at the identified prevalence are therefore negligible for our application. We conclude that the CNN allows determining when the user is in a phone call. For further determining when the screen is turned off during the phone call, a simple threshold comparator can be used, as observable from the trace in Figure 5.6.

8. Discussion and Related Work

In this section, we elaborate attack scenarios and techniques, discuss mitigations to ChoiceJacking attacks and introduce related literature.

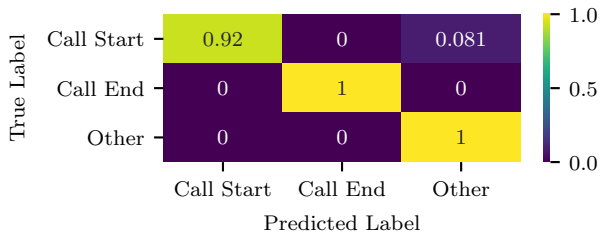


Figure 5.7: Confusion matrix of our PLSC CNN model.

8.1. Suitable Attack Scenarios

While some ChoiceJacking attacks work on locked devices or succeed within a matter of milliseconds, others require an unlocked device and cause on-screen artifacts for a duration of multiple seconds. These attacks are effective in any charging scenario where the unlocked phone screen is outside the user's focus of attention. A very illustrative example of such a scenario can be found in phone calls, as discussed in Section 7. Similarly suitable are scenarios where the user listens to a music video while the device charges from a rented power source (e.g., a hotel room's USB wall socket). It is worth noting that Youtube, which accounts for almost half of all music streaming [12], does not allow locked-screen playback for non-paying customers, and users are typically immersed in other tasks while listening to music [12]. Mobile devices are also commonly attached to rented chargers (power bank, car, etc.) while using navigation apps, where the user's attention is focused on the surroundings rather than the screen. These scenarios or further variations thereof are experienced by most mobile device users on a daily basis.

8.2. Rationale for Multiple Attack Techniques

Our paper presents three attack techniques (T1 through T3), which might appear redundant at first glance. However, presenting several distinct attack techniques serves two important missions. First, it expands the scope of ChoiceJacking attacks, both in terms of affected devices and platforms. Attack technique T3 is relatively slow due to the BT pairing procedure, but it succeeds on both iOS and Android. Attack technique T1 performs fastest, but it requires support for AOAP, which can be missing in unlicensed AOSP-derived platforms. Attack technique T2 exploits a

race condition in a core OS component, so it is likely to succeed even on heavily customized Android variants that lack support for AOAP. Second, presenting multiple techniques illustrates that the attacks are rooted in a systemic lack of considering the security repercussions of dual-role USB connectivity. It therefore intends to provoke a more wholistic way of thinking about dual-role USB and connectivity adversaries in mobile platforms.

8.3. Existing Mitigations

A number of mitigations have been suggested for JuiceJacking attacks that may also be effective against ChoiceJacking. Additionally, mobile OSs already integrate some means to limit the damage an attacker with control over the unlocked device's UI may do. All of these existing mitigations suffer from limitations to their effectiveness in practical scenarios.

USB Data Blockers. A frequent suggestion for mitigating malicious chargers are USB data blockers, which break USB data lines. However, this mitigation assumes that the user is aware of the potential for attacks, rendering it ineffective for the average end user. Data blockers also interfere with modern power negotiation schemes, thereby degrading charge speed.

User Authentication for Security-critical Functions. Android and iOS require user authentication for security-critical functionality such as enabling the USB debugging interface. While this is an effective mitigation against malicious chargers carrying out these operations, there are flaws in Android's implementation. When asking for user authentication, the UI does not provide any clue about the operation it is needed for. To the average user, the authentication screen might look like an innocuous screen unlock, which they routinely authenticate to. A malicious charger can exploit this weakness by triggering the authentication prompt and simply waiting for the user to confirm it.

Lockdown Mode. Both Android (since Android 15) and iOS (since iOS 16) include a Lockdown mode that can be activated to thwart immediate danger in insecure environments. On both OSs, enabling Lockdown mode shuts down USB lines entirely while the lock screen is employed, so that any kind of USB communication is disabled. However, USB protections are restricted to the lock screen. As soon as the device is unlocked, USB connectivity is re-enabled. Most ChoiceJacking attacks operate while the device is unlocked, so that Lockdown mode is ineffective against them.

Additionally, this mitigation needs to be enabled manually, and, therefore, requires the user's awareness of potential attacks.

8.4. Suggested Mitigation

We propose user prompts for all kinds of USB access (host, device, accessory mode) as a mitigation to both our novel ChoiceJacking and existing JuiceJacking attacks. ChoiceJacking attacks exploit the fact that mobile OSs by default trust USB and accessory input devices, although these can be used for elevating the attacker's privileges by interacting with the UI to enable data connections. Since there may be many ways to elevate privileges beyond the flaws we highlighted, we argue that the default trust in USB input devices and accessories needs to be cut. This can be realized by introducing prompts that ask for explicit user consent before an input device or accessory is allowed to interact with the system. Once the USB Type-C Authentication Specification [47] has been widely adopted, the user consent may be displayed for each device upon first attachment and saved for subsequent connections. A similar solution was proposed by Tian et al. [44] and has been adopted in desktop OSs, e.g., in macOS [3].

8.5. Related Work

This section briefly surveys existing work on other USB-based and general connectivity threats.

USB Threats. Prior works have discovered numerous other threats related to USB connectivity on mobile devices. After JuiceJacking and related attacks [17, 20] had been mitigated, several works exploited USB peripherals to extract screen contents [24, 25] or inject voice commands [51]. Tian et al. [43], Imtiaz et al. [14] and Pereira et al. [35] shed light on the consequences of vendor customizations to mobile devices' USB interfaces. They uncover possibilities for flashing firmware files or extracting user data through AT commands. Recent works by Kim et al. [15, 16], Wu et al. [52] and Peng et al. [34] present fuzzers for uncovering memory safety issues in USB and USB PD stacks. In contrast to the design flaws our work focuses on, all these works identify implementation issues such as memory safety issues and logic bugs. Su et al. [42] and Dumitru et al. [9] exploit physical properties of USB to observe or inject off-path communication. Finally,

Tischer et al. [46] and Meng et al. [25] present user studies showing end users are unaware of USB-related threats.

Exploiting Mobile Device Connectivity. Design and implementation weaknesses have recently been found in various other communication interfaces of mobile devices. In 2023, Marc Newlin [27] showed a family of attacks that allows bypassing Bluetooth pairing authentication to inject input events on Android phones. Vanhoef et al. [49, 50] and Schepers et al. [37] have shown multiple attacks on WPA protocols that allow intercepting and hijacking Wifi communication. Cellular standards GSM [4] and UMTS [26] have long been known to allow eavesdropping. Stute et al. [40, 41] proposed multiple eavesdropping attacks on Apple's proprietary wireless protocols. Fahl et al. [10], Oltrogge et al. [33] and others reveal flaws in mobile apps' use of TLS, allowing Man-In-The-Middle attacks on server communication. None of these interfaces allow arbitrary file access as possible over USB.

Power Side-Channels On Mobile Devices. Wired, wireless and standalone power side-channels (PSC) for mobile devices are known in literature. Cronin et al. [7] exploit a PSC attack that allows a malicious charger to infer unlock pins. They detect button-accurate screen presses through the effects of visual button tap feedback on the screen's power consumption. Yang et al. [53] and Wang et al. [51] show that PSCs allow inferring a user's browsing activity and loudspeaker output. Genkin et al. [11] use the USB PSC to extract ECDSA keys from mobile devices. Ni et al. [28] and Oberhuber et al. [31] demonstrate that many of these PSCs can also be exploited indirectly, from neighbouring devices in multi-port chargers or from software. La Cour et al. [19] prove that website fingerprinting is also possible through a PSC from a wireless charger. Ni et al. [29] further improve on this, identifying the running app and entered keystrokes of a mobile device charging from a wireless power bank.

9. Conclusion

Since user confirmations for USB data connections were introduced on mobile platforms, their effectiveness against JuiceJacking attacks remained unchallenged. In this paper, we observed that the design of this countermeasure is based on the flawed assumption that a USB host cannot inject input events. We presented a novel attack family, ChoiceJacking, and three concrete attack techniques that break this assumption on Android

and iOS. They allow bypassing JuiceJacking mitigations and gaining file access or code execution through a malicious charger. We constructed a prototype of a malicious charger to evaluate ChoiceJacking attacks on 11 recent mobile devices from 8 manufacturers. Our evaluation showed that ChoiceJacking attacks succeed across platforms and vendors and can gain USB-based file access in as little as 133 milliseconds in the best case. While most attacks require an unlocked device, we demonstrated attacks for two manufacturers that work while the screen is locked. Finally, we presented a power line side-channel that can be used for detecting suitable attack moments. Most affected manufacturers have already acknowledged the threat posed by ChoiceJacking attacks. We are hopeful they will all integrate the countermeasures we suggest, which will lead to a lasting improvement to USB security in the mobile landscape.

Acknowledgements

We are grateful to Jakob Heher and Stefan More for their valuable ideas, feedback and encouragement, in particular during the early stages of this project. We also extend our thanks to the anonymous reviewers and our shepherd for their time, effort, and insightful contributions, which helped bring this work to publication. This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant number 888087). Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

Ethics Considerations

We responsibly disclosed all identified vulnerabilities to affected vendors. In addition to the 16 Android issues mentioned in Section 1, we also disclosed our findings regarding susceptibility of iOS devices to ChoiceJacking attacks to Apple. We filed our reports between June and July of 2024, and have since been assisting vendors in reproducing the issues on their side and integrating suitable mitigations. Samsung assigned us moderate severity CVE-2024-20900 for the attack principle and rolled out first improvements to their USB mode selection implementation. Apple added user authentication prompts for USB connections in iOS 17.5, but is yet to integrate this mitigation into iOS 18. Xiaomi, Huawei, Vivo and Honor confirmed our findings and are still working on corresponding patches to their Android variants. Google assigned us high severity CVE-2024-43085 for the underlying flaws that caused MTP to be exposed on locked Oppo and Honor devices. A corresponding fix has been rolled out in the November 2024 Android Security Bulletin (ASB). They are also working on fixes for the reported design flaws in upstream AOSP. All evaluations were carried out in a lab environment, on dedicated research devices. We anticipate that the publication of our work will positively impact USB security of mobile platforms and help raise user awareness for USB-based threats.

Open Science

We provide all artifacts for our paper to the public[6]. The artifact package includes our prototype charger, the corresponding firmware, the Python framework for controlling it through USB, the Python code for all attacks and the PLSC. Additionally, we publish video recordings demonstrating our attacks on 11 different devices (10 Android devices and 1 iOS device), which we used for measuring attack timings.

References

[1] Android Open Source Project. Android Open Accessory 2.0. 2012. URL: https://android.googlesource.com/platform/docs/source.android.com/+/03fbc41/src/tech/accessories/aoap/aoa2.md (pp. 98, 103).

[2] Apple Inc, Hewlett-Packard Company, Intel Corporation, Microsoft Corporation, Renesas, STMicroelectronics, and Texas Instruments. USB Power Delivery Specification Revision 2.0. 2017. URL: https://www.usb.org/document-library/usb-power-delivery (p. 96).

[3] Apple, Inc. macOS Ventura 13 Release Notes: Accessory Security. 2022. URL: https://developer.apple.com/documentation/macos-release-notes/macos-13-release-notes#Accessory-Security (p. 122).

[4] Elad Barkan, Eli Biham, and Nathan Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. In: CRYPTO. 2003 (p. 123).

[5] Adam Bates, Joe Pletcher, Tyler Nichols, Braden Hollembaek, Dave Tian, Kevin R. B. Butler, and Abdulrahman Alkhelaifi. Securing SSL Certificate Verification through Dynamic Linking. In: ACM SIGSAC Conference on Computer and Communications Security (CCS). 2014 (p. 102).

[6] Compaq, Digital Equipment Corporation, IBM PC Company, Intel, Microsoft, NEC, Northern Telecom. Universal Serial Bus Specification 1.0. 1996. URL: https://web.archive.org/web/20180130144424/https://fl.hw.cz/docs/usb/usb10doc.pdf (p. 95).

[6] https://zenodo.org/records/14726532

[7] Patrick Cronin, Xing Gao, Chengmo Yang, and Haining Wang. Charger-Surfing: Exploiting a Power Line Side-Channel for Smartphone Information Leakage. In: USENIX Security. 2021 (pp. 92, 100, 117–119, 123).

[8] Florian Draschbacher, Lukas Maar, Mathias Oberhuber, and Stefan Mangard. ChoiceJacking: Compromising Mobile Devices through Malicious Chargers like a Decade Ago. In: USENIX Security. 2025 (p. 89).

[9] Robert Dumitru, Daniel Genkin, Andrew Wabnitz, and Yuval Yarom. The Impostor Among US(B): Off-Path Injection Attacks on USB Communications. In: USENIX Security. 2023 (p. 122).

[10] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: an analysis of android SSL (in)security. In: CCS. 2012 (p. 123).

[11] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. In: CCS. 2016 (p. 123).

[12] International Federation of the Phonographic Industry. Music Consumer Insight Report. 2018. URL: https://www.ifpi.org/wp-content/uploads/2020/07/091018_Music-Consumer-Insight-Report-2018.pdf (p. 120).

[13] Yan Jiang, Xiaoyu Ji, Kai Wang, Chen Yan, Richard Mitev, Ahmad-Reza Sadeghi, and Wenyuan Xu. WIGHT: Wired ghost touch attack on capacitive touchscreens. In: S&P. 2022 (p. 92).

[14] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. Opening Pandora's box through ATFuzzer: dynamic analysis of AT interface for Android smartphones. In: ACSAC. 2019 (p. 122).

[15] Kyungtae Kim, Sungwoo Kim, Kevin R. B. Butler, Antonio Bianchi, Rick Kennell, and Dave (Jing) Tian. Fuzz The Power: Dual-role State Guided Black-box Fuzzing for USB Power Delivery. In: USENIX Security. 2023 (p. 122).

[16] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin R. B. Butler, Antonio Bianchi, and Dave Jing Tian. FuzzUSB: Hybrid Stateful Fuzzing of USB Gadget Stacks. In: S&P. 2022 (p. 122).

[17] Brian Krebs. Beware of Juice-Jacking. 2011. URL: https://krebsonsecurity.com/2011/08/beware-of-juice-jacking/ (pp. 92, 122).

[18] Kyung-Ah Kwon, Rebecca J Shipley, Mohan Edirisinghe, Daniel G. Ezra, Geoff Rose, Serena M. Best, and Ruth E. Cameron. High-speed Camera Characterization of Voluntary Eye Blinking Kinematics. In: J R Soc Interface 10.85 (2013) (p. 117).

[19] Alexander S. La Cour, Khurram K. Afridi, and G. Edward Suh. Wireless Charging Power Side-Channel Attacks. In: CCS. 2021 (p. 123).

[20] Billy Lau, Yeongjin Jang, Chengyu Song, Tielei Wang, Pak Ho Chung, and Paul Royal. Mactans: Injecting Malware into iOS Devices via Malicious Chargers. In: Black Hat USA. 2013 (pp. 92, 99, 122).

[21] Yu-Tsung Lee, William Enck, Haining Chen, Hayawardh Vijayakumar, Ninghui Li, Zhiyun Qian, Daimeng Wang, Giuseppe Petracca, and Trent Jaeger. PolyScope: Multi-Policy Access Control Analysis to Compute Authorized Attack Operations in Android Systems. In: USENIX Security. 2021 (p. 102).

[22] Lukas Maar, Florian Draschbacher, Lukas Lamster, and Stefan Mangard. Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels. In: USENIX Security. 2024 (p. 96).

[23] Slava Makkaveev. Man-in-the-Disk:Android Apps Exposed via External Storage. 2019. URL: https://research.checkpoint.com/2018/androids-man-in-the-disk/ (p. 102).

[24] Weizhi Meng, Fei Fei, Wenjuan Li, and Man Ho Au. Harvesting Smartphone Privacy Through Enhanced Juice Filming Charging Attacks. In: ISC. Vol. 10599. Lecture Notes in Computer Science. 2017 (pp. 92, 100, 122).

[25] Weizhi Meng, Lee Wang Hao, Murali Srirangam Ramanujam, and S. P. T. Krishnan. JuiceCaster: Towards automatic juice filming attacks on smartphones. In: J. Netw. Comput. Appl. 68 (2016), pp. 201–212 (pp. 92, 122, 123).

[26] Ulrike Meyer and Susanne Wetzel. A man-in-the-middle attack on UMTS. In: Proceedings of the 3rd ACM Workshop on Wireless Security. 2004 (p. 123).

[27] Mark Newlin. Hi, My Name Is Keyboard. 2023. URL: https://github.com/skysafe/reblog/tree/main/cve-2023-45866#hi-my-name-is-keyboard (p. 123).

[28] Tao Ni, Yongliang Chen, Weitao Xu, Lei Xue, and Qingchuan Zhao. XPorter: A Study of the Multi-Port Charger Security on Privacy Leakage and Voice Injection. In: MobiCom. 2023 (p. 123).

[29] Tao Ni, Jianfeng Li, Xiaokuan Zhang, Chaoshun Zuo, Wubing Wang, Weitao Xu, Xiapu Luo, and Qingchuan Zhao. Exploiting Contactless Side Channels in Wireless Charging Power Banks for User Privacy Inference via Few-shot Learning. In: MobiCom. 2023 (p. 123).

[30] Karsten Nohl and Jakob Lell. BadUSB - On Accessories That Turn Evil. In: Black Hat USA. 2014 (p. 92).

[31] Mathias Oberhuber, Martin Unterguggenberger, Lukas Maar, Andreas Kogler, and Stefan Mangard. Power-Related Side-Channel Attacks using the Android Sensor Framework. In: NDSS. 2025 (p. 123).

[32] ofcom. Mobile Matters: Using crowdsourced data to assess people's experience of using mobile networks. 2021. URL: https://www.ofcom.org.uk/siteassets/resources/documents/research-and-data/telecoms-research/mobile-matters/2021/mobile-matters-2021-report.pdf (p. 118).

[33] Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications. In: USENIX Security. 2021 (p. 123).

[34] Hui Peng and Mathias Payer. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In: USENIX Security. 2020 (p. 122).

[35] André Pereira, Manuel Eduardo Correia, and Pedro Brandão. USB Connection Vulnerabilities on Android Smartphones: Default and Vendors' Customizations. In: Communications and Multimedia Security. Vol. 8735. Lecture Notes in Computer Science. 2014, pp. 19–32 (p. 122).

[36] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, but verify: A longitudinal analysis of android oem compliance and customization. In: S&P. 2021 (p. 96).

[37] Domien Schepers, Aanjhan Ranganathan, and Mathy Vanhoef. Framing frames: bypassing Wi-Fi encryption by manipulating transmit queues. In: USENIX Security. 2023 (p. 123).

[38] Riccardo Spolaor, Hao Liu, Federico Turrin, Mauro Conti, and Xiuzhen Cheng. Plug and Power: Fingerprinting USB Powered Peripherals via Power Side-channel. In: IEEE INFOCOM. 2023 (p. 102).

[39] statcounter GlobalStats. Mobile Vendor Market Share Worldwide - May 2024. 2024. URL: https://gs.statcounter.com/vendor-market-share/mobile (p. 107).

[40] Milan Stute, Alexander Heinrich, Jannik Lorenz, and Matthias Hollick. Disrupting Continuity of Apple?s Wireless Ecosystem Security: New Tracking, DoS, and MitM Attacks on iOS and macOS Through Bluetooth Low Energy, AWDL, and Wi-Fi. In: USENIX Security. 2022 (p. 123).

[41] Milan Stute, Sashank Narain, Alex Mariotto, Alexander Heinrich, David Kreitschmann, Guevara Noubir, and Matthias Hollick. A Billion Open Interfaces for Eve and Mallory: MitM, DoS, and Tracking Attacks on iOS and macOS Through Apple Wireless Direct Link. In: USENIX Security. 2019 (p. 123).

[42] Yang Su, Daniel Genkin, Damith Ranasinghe, and Yuval Yarom. USB Snooping Made Easy: Crosstalk Leakage Attacks on USB Hubs. In: USENIX Security. 2017 (p. 122).

[43] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin R. B. Butler. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In: USENIX Security. 2018 (pp. 96, 122).

[44] Dave Jing Tian, Adam Bates, and Kevin Butler. Defending Against Malicious USB Firmware with GoodUSB. In: ACSAC. 2015 (p. 122).

[45] Jing (Dave) Tian, Nolen Scaife, Deepak Kumar, Michael D. Bailey, Adam Bates, and Kevin R. B. Butler. SoK: "Plug & Pray" Today - Understanding USB Insecurity in Versions 1 Through C. In: S&P. 2018 (p. 99).

[46] Matthew Tischer, Zakir Durumeric, Sam Foster, Sunny Duan, Alec Mori, Elie Bursztein, and Michael Bailey. Users Really Do Plug in USB Drives They Find. In: S&P. 2016 (p. 123).

[47] USB 3.0 Promoter Group. Universal Serial Bus Type-C Authentication Specification. 2019. URL: https://www.usb.org/document-library/usb-authentication-specification-rev-10-ecn-and-errata-through-january-7-2019 (p. 122).

[48] USB3.0 Promoter Group. Universal Serial Bus Type-C Cable and Connector Specification Release 2.3. 2023. URL: https://www.usb.org/document-library/usb-type-cr-cable-and-connector-specification-release-23 (p. 96).

[49] Mathy Vanhoef. Fragment and Forge: Breaking Wi-Fi Through Frame Aggregation and Fragmentation. In: USENIX Security. 2021 (p. 123).

[50] Mathy Vanhoef and Eyal Ronen. Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd. In: S&P. 2020 (p. 123).

[51] Yuanda Wang, Hanqing Guo, and Qiben Yan. GhostTalk: Interactive Attack on Smartphone Voice System Through Power Line. In: NDSS. 2022 (pp. 92, 99, 122, 123).

[52] Yilun Wu, Tong Zhang, Changhee Jung, and Dongyoon Lee. DevFuzz: Automatic Device Model-Guided Device Driver Fuzzing. In: S&P. 2023 (p. 122).

[53] Qing Yang, Paolo Gasti, Gang Zhou, Aydin Farajidavar, and Kiran S. Balagani. On Inferring Browsing Activity on Smartphones via USB Power Analysis Side-Channel. In: IEEE Transactions on Information Forensics and Security 12.5 (2017), pp. 1056–1066 (pp. 92, 100, 117, 118, 123).

6

Clone2Pwn: A Systematic Security Analysis of Data Migration Tools in the Android Ecosystem

Publication Data

Florian Draschbacher, Lukas Maar, Lorenz Schumm, Rene Denifl, Treffner Lukas, and Stefan Mangard. Clone2Pwn: A Systematic Security Analysis of Data Migration Tools in the Android Ecosystem. In: ESORICS. 2026

Contributions

The author of this thesis is the main author of the paper. He came up with the idea for the paper, developed the attack scenarios and detection routines and implemented most of the end-to-end attacks. The thesis author also wrote most of the paper's text.

Clone2Pwn: A Systematic Security Analysis of Data Migration Tools in the Android Ecosystem

Florian Draschbacher Lukas Maar Lorenz Schumm
Rene Denifl Lukas Treffner Stefan Mangard

Graz University of Technology

Abstract

To simplify the transfer of user data to a new device, all major smartphone manufacturers offer data migration (or *phone clone*) tools. Despite the sensitivity of the involved data, the security of current-generation data migration tools has not been analyzed comprehensively.

In this paper, we systematically analyze state-of-the-art data migration tools from 7 major Android vendors (Google, Samsung, Vivo, Xiaomi, Oppo, Huawei, Honor) with a combined download count exceeding 12 billion. We identify generic attack scenarios, and check concrete apps for realizability of these scenarios based on a list of requirements. Our analysis uncovers severe security vulnerabilities in all tools that allow attackers within Wi-Fi range to eavesdrop sensitive user data. The eavesdropped data includes communication records, personal media and documents, and login credentials, effectively representing a sizable dump of a user's digital life. In practice, attackers can exploit these vulnerabilities by placing cheap equipment near places such as mobile carrier stores, where data migrations are typically done multiple times per day. Moreover, five tools are vulnerable to data injection attacks, allowing attackers within Wi-Fi range to achieve code execution on a freshly set-up device. Finally, three tools are also susceptible to data extraction or injection attacks from local attackers. We responsibly disclosed our results to affected vendors, so far leading to eight CVEs, five of which are high-severity.

1. Introduction

To encourage device sales, mobile device manufacturers provide tools that facilitate data migration from an old device to a newly purchased one.

Through tight OS integration, these tools can bypass sandbox boundaries to, e.g., seamlessly place imported messages into the suitable data store on the new device.

The data transmitted in a device migration typically represents a sizable dump of a person's digital life. In the wrong hands, this data could be misused to cause considerable harm. Data migration tools must therefore make sure that transmissions cannot be eavesdropped or manipulated. All data migration tools from major device vendors operate without Internet connection, either transmitting data through a peer-to-peer Wi-Fi connection or a cable. However, simply omitting a remote server from the system does not ensure secure transmission.

Despite the security sensitivity of data migration tools, they remain largely underexplored. The only prior study, conducted by Ma et al. [27] in 2019, reported that five of 8 tested tools leaked transmission data to disk and four were susceptible to wireless eavesdropping. While it provided an important first look at data migration security, it (i) excluded Android Switch (Google) and Smart Switch (Samsung)—the current market leaders—and (ii) reflected the landscape from about six years ago. More broadly, no subsequent work has re-evaluated these tools at scale or examined their security in today's ecosystem. Still, data migration tools have evolved substantially in recent years and expanded their user bases by orders of magnitude, rendering any vulnerabilities relevant to a large portion of smartphone users.

In this paper, we close this research gap through a comprehensive security analysis of today's data migration tool landscape in the Android ecosystem. We reveal that tools from the major vendors suffer from multiple vulnerabilities that can lead to leakage of sensitive data, and in some cases even code execution. Methodologically, we go beyond Ma et al. [27] by defining generic attack scenarios against these tools as cross-products of attacker goals and attack surfaces, enabling a systematic and extensible evaluation across tools. For each attack scenario, we then identify the attack requirements, i.e., necessary conditions that cause susceptibility to the scenario in a data migration tool. For example, wirelessly eavesdropping all transmitted user data requires a combination of weak link-level protection and lack of app-level encryption.

Next, we evaluate seven of the most popular data migration tools in the Android ecosystem based on the identified attack scenarios. As shown in Table 6.1, these tools have been downloaded between one million and

Brand	Tool	Version	Downloads
Google	Android Switch	1.0.740410803	10B+
Samsung	Smart Switch	3.7.64.10	1B+
Vivo	EasyShare	5.10.5.3_gp	1B+
Xiaomi	Mi Mover	4.2.8	100M+
Oppo	Clone Phone	15.9.3 / 14.7.22_third_gp	10M+
Huawei	Phone Clone	14.5.0.550	10M+
Honor	Device Clone	14.8.5.306 / 11.6.1.320	1M+

Table 6.1: Data migration tools evaluated. Differing app versions for third-party devices are indicated separately.

10 billion times[1]. Most Android devices ship preinstalled with two of these tools. Besides Android Switch, which is shipped with every Google-certified Android device independent of manufacturer [21], the analysed tools account for 75 % of the Android market [9]. For each tool, we determine whether our attack scenarios can be realized, i.e., whether the tool's implementation fulfills the attack requirements causing susceptibility. Since most tools are vendor-specific and only preinstalled (with privileged data access) on that vendor's devices, we consider two constellations for each tool. These are first-party migrations between two devices from the same manufacturer, and third-party migrations, where the tool is only privileged on the new device. First-party migrations usually include more sensitive data only accessible to privileged apps, such as, e.g., Wi-Fi credentials, app-private data, and more. Throughout this paper, we use the vendor name (e.g., Google) to refer to their respective data migration tool (e.g., Android Switch).

Our analyses reveal that all inspected apps are susceptible to wireless attacks. This is because six of seven apps use no or trivially bypassable app-level encryption or integrity checks together with a weak WPA2 Pre-Shared Key (PSK). The remaining app uses Wi-Fi Direct with a peer identification scheme that can be spoofed. The resulting wireless attacks either allow eavesdropping on the transmitted user data, or even manipulating it on the fly, e.g., for code execution in the context of transmitted apps. An attacker can carry out these attacks by placing cheap equipment within Wi-Fi range of places like mobile carrier stores, where data migrations are a frequent occurrence [36, 43, 44]. Moreover, for three of seven apps, a local attacker, e.g., a malicious app, can also

[1]Data according to Google Play or Huawei AppGallery listings.

extract or manipulate user data due to lack of authentication for open TCP sockets. Lastly, we implement end-to-end exploits for all identified attacks and detail two of them for Samsung Smart Switch, a particularly popular data migration tool.

Contributions. The main contributions of our work are

(1) We define attack scenarios for data migration tools and derive analysis procedures for checking susceptibility of a tool based on attack requirements.
(2) Through these analysis procedures, we identify attacks against 7 data migration tools from the most popular Android manufacturers.
(3) We implement end-to-end exploits for all identified attacks and discuss particularly interesting cases in Samsung Smart Switch.

Disclosure. We responsibly disclosed 16 variants of 14 end-to-end exploits to all analysed vendors. They acknowledged 13 of our reports (1 response still pending)[2], and either already rolled out or are rolling out fixes. So far, Samsung, Oppo, Vivo, and Huawei issued us 8 CVEs (5 high, 3 moderate severity)[3].

2. Background and Related Work

This section briefly introduces Wi-Fi and Android background, and discusses related work.

2.1. Background

Wi-Fi refers to protocols defined in IEEE 802.11 for wireless local data transmission over radio [22]. In a typical local network setup, one router functions as the station (AP) while connected client devices operate in STA mode. Wi-Fi operates primarily in 2.4 GHz and 5 GHz bands, which exhibit different propagation characteristics. In an outdoor setting, 2.4 GHz transmissions can achieve ranges of more than 100 meters [48]. In practice, the expected Wi-Fi range in indoor settings is commonly estimated between 20 and 45 meters for the 2.4 GHz band and about 75 % of that for the 5 GHz band [24, 30].

[2]Oppo could not reproduce S4 despite our PoC; Honor did not acknowledge S2.
[3]CVE-2025-{21060, 21061, 21062, 21064, 21078, 27387, 15515, 68963}. CVEs promised by Google, Xiaomi still pending as of the final manuscript of this paper.

WPA/WPA2/WPA3 are successors to the Wired Equivalent Privacy (WEP) incorporated in the original 802.11 standard, which is considered broken nowadays [18]. Wi-Fi Protected Access (WPA) addresses WEP's flaws through improved key management and authentication mechanisms. The succeeding WPA2 protocol uses the AES block cipher and a four-way handshake to establish secure communications. However, WPA2 does not offer forward secrecy, such that a compromised pre-shared key enables decryption of previously captured communication. The most recent WPA3 finally added forward secrecy.

Wi-Fi Direct is an 802.11 standard extension for peer-to-peer communication between Wi-Fi devices, without requiring the traditional AP infrastructure. This protocol establishes ad-hoc networks through group owner negotiation, where one device assumes coordination as the group owner, while connected peers operate as clients. Wi-Fi Direct implementations inherit the security protocols of the underlying Wi-Fi standard, typically WPA2 or WPA3.

Android is an open-source mobile OS developed by Google that dominates the smartphone industry [38]. To ensure a coherent ecosystem despite vendor customizations, devices have to undergo Google's Compatibility Test Suite [1]. Devices that pass are eligible for a license to ship with Google Mobile Services, which includes ecosystem components such as the Google Play Store. The Android Platform Security Model [28] mandates that apps on Android are sandboxed. Their access to system resources such as the file system is tightly restricted. The restrictions can only be selectively relaxed by an app requesting permissions. Particularly powerful permissions can only be granted to privileged apps that are signed with the same certificate as the system. Every app is assigned a private data folder that no other app can access. Apps typically use this folder to store information such as login tokens or other app-specific user data. Usually, not even privileged apps can directly access data in other apps' private data folder. However, privileged apps with the `BACKUP` permission can request a dump of the contained data. Unless an app opts out, this request is granted. For the remainder of this paper, we assume Android's security model remains uncompromised by other privilege escalation attacks that have been presented before [11, 15, 17, 33].

Wi-Fi APIs allow Android apps to host or join Wi-Fi networks. On modern Android versions, unprivileged apps can start a Wi-Fi hotspot through the modern `LocalOnlyHotspot` API [4] or through the legacy Wi-Fi Direct Hotspot API [3]. Both establish a local Wi-Fi hotspot with

a configurable network name (SSID) and PSK. The latter allows Wi-Fi Direct peers to join, in addition to Wi-Fi devices in STA mode that are also supported by the former. Privileged apps can also utilize the Wi-Fi Tethering API [6] originally intended for sharing the device's Internet connection via Wi-Fi. For joining a Wi-Fi network, unprivileged apps can choose between the `WifiNetworkSpecifier` [7] and the `WifiNetworkSuggestion` [8] APIs. While the former only establishes a temporary app-local link without Internet access, the latter works on a device-global scale, supporting auto-reconnect and Internet access. Both require user consent through a system dialog. Privileged apps can still use the otherwise deprecated `WifiManager.addNetwork()` [2] API. It does not require user consent but otherwise operates like the `WifiNetworkSuggestion` API. Finally, apps can use Wi-Fi Direct APIs [5] for negotiating a Wi-Fi connection between two devices without needing the APIs described above.

2.2. Related Work

Prior works have explored the security and privacy implications of wireless communication and external resource interactions in mobile systems. Naveed et al. [32] highlighted the mis-binding threats from external devices in Android, showing how malicious peripherals can be misrepresented as trusted components. Building on this, Demetriou et al. [14] further dissected both mandatory and discretionary protections for external resources, demonstrating vulnerabilities in Android's security framework when managing peripheral interactions. A broader survey by Wang and Yan [45] systematically reviewed the landscape of device-to-device communication security, identifying numerous threat vectors and proposing general countermeasures. More specialized work by Li et al. [23] exploited vulnerabilities in smart provisioning protocols, specifically SmartCfg, to capture Wi-Fi credentials from air-based transmissions, revealing how initial network setups can leak sensitive information. Cristalli et al. [13] focused on the absence of authentication protocols in app-to-app communications over short-range channels, emphasizing the risks in cross-device scenarios where trust is implicitly assumed. Similarly, Liu et al. [25] analyzed security threats in device-to-device apps, discussing attack surfaces related to permission abuse and communication mismanagement in proximity-based interactions. Recent work has shifted attention to data cloning and device impersonation. Ma et al. [27] and Song et al. [37] exposed vulnerabilities in hotspot-based cloning services and Android

system customization, respectively, showing how attackers could replicate or siphon user data via network configurations or system-level modifications. Extending beyond Android, Stute et al. [39, 40] examined Apple's wireless ecosystem, uncovering tracking, denial-of-service, and machine-in-the-middle vulnerabilities through BLE, AWDL, and Wi-Fi. While these studies underline a persistent pattern of insufficient authentication, weak isolation, and exploitable trust assumptions in wireless and cross-device communication, none systematically evaluate the security of current-generation data migration tools, as we do in this study.

3. Threat Models

In line with prior work [27], we assume an adversary who attacks a data migration between two devices. We observe that the new device typically starts out empty and the old device is abandoned after the migration. We therefore consider two distinct attacker goals: (1) exfiltrating data from the old device and (2) injecting (manipulated) data into the new device. Finally, we rule out implementation bugs in the OS or in the underlying protocols, such as WPA2.

We distinguish between two different attacker models based on the privileges prior to the attack. We explicitly consider hybrids between these two models (i.e., a proximity attacker who also has local access) unrealistic.

M1: Proximity Attacker. In this model, the attacker starts with no access to the target device. The only assumption is that the attacker is physically positioned in proximity to the data migration. We observe there are certain physical places that have a high occurrence of data migrations per day. For example, these include retail locations of phone carriers [36, 43, 44], electronics stores [10] or mobile phone repair shops [12, 41]. In these locations, people not only set up their newly purchased devices, but can also get this done as a service. A high volume of data migrations is also likely to be found close to a company's IT department, if the employees' mobile devices are centrally managed and maintained. We assume the distance between this attacker and the data migration does not exceed the range of Wi-Fi radio signals (cf. Section 2). The attacker can hide a small self-contained computer, such as a Raspberry Pi, anywhere within this range. This computer scans the Wi-Fi beacons of nearby Wi-Fi networks until one of them reveals a data migration app (e.g., through its SSID pattern). Given the low cost of the involved equipment (less than $100),

this attack can easily be scaled. Through planting a large number of such units, an adversary can collect data from or infect many devices in a short time span. For attacks that require near-realtime PSK brute-forcing, each unit can access cheap remote GPUs via an Internet connection, e.g., from Amazon Web Services.

M2: Local Attacker. Here, the attacker already has code execution capabilities on the old device. For example, an otherwise benign app might integrate a malicious library. Due to the Android Platform Security Model [28], the app is still sandboxed, i.e., can only access limited system resources. Attacking a data migration might thus allow elevating its privileges, i.e., bypassing sandbox restrictions. This can, e.g., include gaining code execution in another app on the new device, or accessing resources not otherwise reachable for the malicious app.

Attack Impact. Through any of the described attacks, the attacker gains access to sensitive user information. All examined data migration tools transmit messages, call logs, contacts, pictures, videos, and documents. For first-party migrations (between two devices from the same vendor), transmissions often also include Wi-Fi credentials and app-private data, which commonly contains session tokens or login credentials (see Section 5.3). Any of this data can be misused to cause serious harm to the device owner, their social circle, or employer. Some of the attacks additionally allow manipulation of the transmitted data. Manipulating transmitted app packages, i.e., a variant of an app repackaging attack [29], allows gaining code execution in the context of third-party apps. As a result, the attacker can, e.g., access app-private data even if it was not included in the migration itself. Additional privilege escalations can be accomplished, e.g., by overwriting system configurations such as granted permissions for an app.

4. Attack Scenarios for Data Migration Tools

This section defines attack scenarios for data migration tools as cross-products of attacker goals and attack surfaces. Per scenario, we identify attack requirements that need to be fulfilled for the scenario to be realizable for a specific app.

General Operation of Data Migration Tools. An initial analysis indicates that all data migration tools operate in 4 basic steps. (1) They

start with a pairing phase, in which a direct link between the old and new device is established. In the examined apps, this link is either a Wi-Fi connection (one device acts as Wi-Fi AP), Wi-Fi Direct, or USB connection. (2) Next, the app on the old device prepares the data to be transmitted. Involved steps may include data export from databases, compression, and/or copying to a central place. (3) The prepared data is sent to the new device over the previously established direct link. For links based on Wi-Fi, this involves communication over TCP. (4) Finally, the received data is reinstated on the new device, roughly inversing (2).

Attack Surfaces. The different attacker models (cf. Section 3) reach different attack surfaces of the operation described above. An **M1** (proximity) attacker can interact with steps (1) and (3), i.e., the pairing and transmission procedures. The main attack surface reachable to an **M1** attacker in these operations is the link layer. It is worth noting that an **M1** attacker is only effective against the most common scenario of wireless transmissions (cf. Section 7). For **M2** (local) attackers (which we restrict to the old device as explained in Section 3), only steps (2) and (3) are accessible. The main attack surfaces involved here are the file system or open TCP/IP sockets.

4.1. Definition of Attack Scenarios

Based on the attack surfaces discussed above, we define attack scenarios for data migration tools. To this end, we consider cross-products of main attack surfaces (proximity Wi-Fi, local file system and TCP/IP sockets) and attacker goals (data exfiltration and injection). The attack scenarios incorporate known attack techniques against Wi-Fi [42], TCP/IP [26], general device-to-device connectivity [13] and data transfer tools [27]. For every scenario, we list requirements, i.e., design or implementation details that need to be present in a data migration tool for the scenario to be realizable. We emphasize on realistic scenarios based on slightly refined attacker models from prior works (see Section 3) rather than on exhaustiveness. In particular, we only consider attacks that do not visibly interfere with the data migration, i.e., do not cause alertness of the user due to errors or warnings visible in the UI. Table 6.2 shows an overview of the requirements per scenario. In the following, we discuss the scenarios in detail.

Requirement	S1	S2	S3	S4	S5	S6
R1: App is identifiable through Wi-Fi beacons	✓	✓	-	-	-	-
R2: Wireless protection weak	✓	-	-	-	-	-
R3: No app-level payload encryption	✓	-	✓	✓	-	-
R4: Wireless protection very weak (Wi-Fi AP)	-	✓[1]	-	-	-	-
R5: Pairing identification ambiguous (Wi-Fi Direct)	-	✓[1]	-	-	-	-
R6: Payload encryption not derived from out-of-band secret	-	✓	-	-	-	-
R7: Payload integrity not protected via out-of-band secret	-	✓	-	-	-	-
R8: Server socket on old device accepts extra clients	-	-	✓	-	-	-
R9: No authentication for payload connections	-	-	✓	✓	-	-
R10: Wi-Fi join is device-global	-	-	-	✓	-	-
R11: Server socket on new device accepts extra clients	-	-	-	✓	-	-
R12: No encrypted or separately transmitted checksums	-	-	-	✓	-	-
R13: User data (temporarily) stored in public file system	-	-	-	-	✓	✓
R14: No encryption for data stored in public file system	-	-	-	-	✓	✓
R15: No integrity checks for data in public file system	-	-	-	-	-	✓

[1] Either **R4** or **R5** is necessary for **S2**.

Table 6.2: Summary of requirements for different attack scenarios **S1** to **S6**.

S1: Eavesdropping on wireless communication. The **M1** attacker captures Wi-Fi frames and later (offline) extracts the data exchanged in the transmission. Many Wi-Fi chips support monitor mode, which allows passively (therefore stealthily) capturing Wi-Fi frames destined for other devices. For this scenario to be viable, the Wi-Fi beacons of the data migration app must allow identification of the specific tool, e.g., through the SSID of the hotspot it creates. For the attack to yield user data, the attacker needs to be able to decrypt the captured Wi-Fi frames. This means the app's Wi-Fi hotspot needs to be either WPA2-protected with a weak PSK or unprotected. We consider a WPA2 PSK weak if it can be brute-forced from a captured WPA2 handshake in a matter of hours. For reference, an Nvidia GeForce RTX5090 commodity GPU can brute-force WPA2 at a rate of 3.4 MH/s. Finally, this scenario is only realizable if there is no effective app-level encryption for payloads, i.e., encryption based on an asymmetric key exchange or an out-of-band secret prevents this attack.

S2: Manipulating wireless communication. The **M1** attacker gains a mediating position between the old and new device to alter the payloads exchanged between them. For normal AP Wi-Fi connectivity, it can be accomplished, e.g., through a multi-channel machine-in-the-middle (MC-MITM) attack (cf. Section 9.1), ARP spoofing, or DHCP spoofing. The attacker needs to be able to decrypt and re-encrypt the transmission in real-time, both at the link and app layers. This means the app needs to use

WPA2 or no protection for its hotspot. The WPA2 key must leak to the attacker or be very weak, i.e., brute-forceable in near-realtime. Specifically, we consider a WPA2 PSK very weak if it can be realistically brute-forced within the 15 seconds that Android waits for a DHCP lease. During this time, a MC-MITM can artificially delay connection establishment by intentionally not forwarding Wi-Fi frames. A detailed description of this technique can be found in Section 9.1. If Wi-Fi Direct is used, an attacker can achieve a MITM position if the target device is not unambiguously identified prior to forming a group. Finally, for this attack scenario to be realizable, any app-level payload encryption or integrity checking must not be cryptographically linked to an out-of-band (OOB) secret. If an OOB secret is only used for authentication, an attacker can still perform a relay attack without knowing the secret.

S3: Extracting data via local socket connection. A malicious app (**M2** attacker) running on the old device might be able to connect to a listening socket on the same device opened by the data migration tool. If needed, SYN scanning [26] allows identifying dynamic ports. For a connection attempt to succeed, the listening socket must accept additional payload connections. The attacker has no information about the transmission session, so an attack can only work if payload connections are unauthenticated and unencrypted.

S4: Injecting data via socket connection to new device. If a data migration tool opens a listening socket on the new device, a malicious app (**M2** attacker) on the old device might use it to inject data. For identifying the new device's IP address and port, the attacker can, e.g., use ARP scanning and SYN scanning. As for **S3**, the tool must allow extra payload connections that are unauthenticated and unencrypted. Finally, it needs to use a joining API that establishes a device-global connection (cf. Section 2).

S5: Extracting data through file system. If the data migration tool stores migration data in public storage, an **M2** attacker can extract it from there. For tools privileged on the old device (first-party data migrations), these files may leak data otherwise inaccessible to user-installed apps. This scenario is only viable if the data migration tool does not (effectively) encrypt its files.

S6: Manipulating data through file system. In addition to **S5**, a malicious app can manipulate migration data if there are no integrity

checks for the (unencrypted) files the data migration tool stores in public storage.

5. Systematic Analysis

In this section, we analyze the seven data migration apps described in Section 1 for their susceptibility to the six attack scenarios introduced in Section 4. We use dynamic testing and manual static analysis to systematically determine susceptibility through the presence of requirements **R1** to **R15**.

5.1. Evaluation Setup

Per app, we assess first-party data migration between two devices from the same manufacturer and third-party data migration between differing manufacturers. The exception is Google, where any Google-certified Android device (due to the app being preinstalled) is considered first-party (we use the Samsung Galaxy S23), and third-party migrations are not supported. We use Honor as source for first-party migrations on Huawei and vice versa for lack of a second device from these vendors. This is possible because the two apps use the same protocol, and reverse-engineering reveals strong similarities between their code bases. We thus anticipate the core findings for these apps align with first-party migrations.

Table 6.3 lists the Android devices we use for our analysis. For every data migration app manufacturer, we use a first-party device purchased in either 2024 or 2025 as the data migration target. We can therefore assume that our results accurately reflect real-world migration to a freshly purchased device. For analysis steps that require root access (see Section 5.2), we root the Google Pixel 8a using Magisk [46] and use it as the old device in a third-party migration.

If an analysis requires capturing Wi-Fi communication, we utilize a Raspberry Pi 5 running Kali Linux and an external Wi-Fi interface based on the Mediatek MT7921 chipset. For brute-forcing WPA2 PSKs, we use hashcat on a Ubuntu 22.04.4 LTS Linux host with an Nvidia GeForce RTX5090 commodity GPU. We use JADX [35] and Ghidra [31] for reverse-engineering managed and native components of app package (APK) files.

Brand	Device	OS	Year
Google	Pixel 8a[s,t]	15	2024
Samsung	Galaxy S23[s,t]	14	2025
Samsung	Galaxy A54[s]	14	2024
Vivo	V40 SE 5G[t]	15	2025
Vivo	Y36[s]	13	2024
Xiaomi	Redmi Note 12 5G[t]	14	2024
Xiaomi	12[s]	14	2023
Oppo	A58[t]	15	2024
Oppo	A94 5G[s]	11	2023
Huawei	nova 12i[s,t]	14	2024
Honor	90 Lite[s,t]	14	2024

Table 6.3: Devices for evaluation, their Android version and purchase year. s/t = migration source/target.

5.2. Analysis Methodology

We determine the susceptibility to attack scenarios **S1** through **S6** from Section 4 through the presence of respective requirements **R1** to **R15** in an app. Additionally, we evaluate the data each data migration tool can migrate. For efficiency reasons, we employ short-circuit evaluation. If one requirement for a given attack is not present in an app, we do not evaluate for the presence of other requirements for that attack. Where possible, we evaluate the presence of a requirement through dynamic testing procedures whose outcomes provide definitive proof for the presence or absence of the respective implementation detail. However, for some requirements, such a procedure cannot be found or is not feasible. In these cases, we employ reverse-engineering of the app code to arrive at a conclusion. Unless noted otherwise, we carry out every check for every tool for both first-party and third-party data migrations. As some vendors prevent rooting, we aim for analysis procedures that work without root access where possible. In the following, we describe the analysis procedure for each attack requirement, as well as for finding the per-app migration data scope.

Data included in migrations. For determining the scope of data migrations, we prepare the old device for each app with pictures, videos, PDF documents, phone calls, SMS messages, contacts, and Wi-Fi credentials. Additionally, we install a simple app that stores fake secrets in its private data folder. We then check for the presence of these data items in transmission captures (described for **R3** below). For apps where all traffic

is encrypted, we confirm data scope by checking data items present on the new device after the data migration.

Wi-Fi beacons identify data migration app (R1). We run multiple pairing phases per app, inspecting Wi-Fi beacons from captures and noting down the SSIDs of started hotspots. If we recognize identical sections in the SSIDs (beyond recognizable device model numbers), we use these strings for a search into the app code. This allows us to confirm if these strings are constants in the code, which allows identification of the app.

Weak (R2) or very weak (R4) wireless protection. We determine the used Wi-Fi protection scheme through a Wi-Fi scanner immediately after the pairing step of the data migration tool. Some apps perform connection upgrades to 5 GHz Wi-Fi channels, and change the Wi-Fi protection scheme in the process. We thus additionally scan during the data transmission. If we detect WPA2, we reverse-engineer the app to determine the method used for generating the PSK. For PSKs that are randomly generated, we determine the PSK length and used character set. This allows us to calculate the worst-case time it takes to brute-force the PSK on commodity hashcat setups or rentable cloud compute.

App-level payload encryption (R3). We capture the Wi-Fi traffic of a transmission covering all supported data types and (if necessary) decrypt the captured Wi-Fi frames. As needed, we either extract the PSK from the UI, from verbose Wi-Fi logs on unrooted devices, derive it from the SSID, or brute-force it. From the decrypted Wi-Fi communication, we re-assemble TCP streams, which we then manually inspect. We try to identify all transmitted data types through their content or unique format. If we cannot find some data in the capture, we reverse-engineer the app to see if the reason was encryption.

Pairing device identification (R5). For normal Wi-Fi connections between a STA and an AP, pairing is based on the knowledge of both the SSID and PSK. In these cases, the attacker can only interfere with the pairing if they know the PSK (cf. **R4**). We determine an app's use of Wi-Fi Direct through scanning for corresponding advertisement beacons. If we find such beacons, we reverse-engineer the app to identify the pairing information used for identifying the device with which to start Wi-Fi Direct group negotiation.

Cryptography based on OOB secrets (R6, R7). First, we check for OOB secrets in the implementation. We complete a data migration,

taking note of all communication channels between the two devices beyond the primary link (usually Wi-Fi). If the pairing phase involves scanning a QR code, we manually extract and analyse its contents. Next, we determine the presence of payload encryption (**R6**) or integrity checks (**R7**). For encryption, we use the procedure described for **R3**, additionally checking for involvement of the OOB secret. Equally, we extend the integrity checks procedure described for **R12**, reverse-engineering involved logic and correlating it with communication captures.

Sockets accepting additional connections (R8, R11). We monitor the procfs TCP tables via the Android Debug Bridge (ADB) to identify sockets used for the data migration. ADB can be enabled on any Android device (rooted or not). For every listening socket on either the new (**R8**) or old (**R11**) device, we observe whether it accepts multiple clients during a legitimate data migration. If this is the case, we use connection captures to determine whether these clients carry data payloads. Finally, we manually reverse-engineer the involved logic to determine whether arbitrary payload clients can connect.

Authentication for extra connections (R9). We reverse-engineer the protocol used for payload connections. We then send packets for sending or requesting a single file from an additional socket client during a legitimate data migration. If the file extraction or injection succeeds, no authentication is used.

App-level payload integrity checks (R12). We use Frida [34] to monitor `MessageDigest` and `Hmac` APIs, checking if they process transmitted data. We also hook framework APIs for retrieving APK signatures. If any hook fires, we identify and reverse-engineer the relevant logic in the app's code. Due to needing root access, we only run this check for third-party migrations.

Wi-Fi joins device-global (R10). During a data migration, we check if a ping to the new device succeeds from a separate process on the old device.

Data in public file system (R13, R14, R15). We use the `FileObserver` API from a service on the old device to monitor all files written to during a data migration. We manually inspect these files to see if they contain user data (**R13**) and are unencrypted (**R14**). For files whose contents cannot be classified, we use reverse-engineering to determine if and what encryption is used, and trace back the origin of the key. We test

App	🖼	🎥	📄	📞	💬	🪪	</>	🗄	📶	🧰
Google	✓	✓	-	✓	✓	✓	-	✓	✓	✓
Samsung	✓	✓	✓	✓	✓	✓	✓	-	✓[1]	-
Vivo	✓	✓	✓	✓	✓	✓	✓	-	✓[1]	-
Xiaomi	✓	✓	✓	✓	✓	✓	✓	-	✓[1]	-
Oppo	✓	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	-
Huawei	✓	✓	✓	✓	✓	✓	✓	-	-	-
Honor	✓	✓	✓	✓	✓	✓	✓	-	✓[1]	-

[1] Only first-party data migrations.

Table 6.4: Content included in data migrations per app. 🖼 = Pictures, 🎥 = Videos, 📄 = Documents, 📞 = Call Logs, 💬 = SMS/MMS, 🪪 = Contacts, </> = Apps (APK files), 🗄 = App Data, 📶 = Wi-Fi Credentials, 🧰 = Health Data.

for integrity checks (**R15**) by replacing files during a data migration and confirming that they are restored on the new device.

5.3. Analysis Results

Attack Impact. The data types each migration tool transmits can be found in Table 6.4. Beyond basic media and communication data, all except Google also support migrating documents and apps (APKs). It is worth noting that for Google in particular, the transmission of certain data types is not apparent in the UI and cannot be disabled. Specifically, all Google data migrations include highly sensitive health data (including sexual activity and other reproductive health data). Google also migrates app-private data for installed apps, even if the app (APK) the data belongs to is not migrated. First-party data migrations commonly include additional information related to manufacturer-specific services that we do not examine at scale. Unless otherwise noted, the attacks described in the following concern all data included in the data migration.

S1: Eavesdropping on Wireless Communication. Every analyzed data migration app supports wireless migration. Samsung uses Wi-Fi Direct with strong WPA2 keys, preventing passive eavesdropping (i.e., **S1**). The other tools establish a Wi-Fi AP with a unique SSID pattern (**R1**) on the new device, protected only with a weak (**R2**) or very weak WPA2 PSK. Vivo starts out with a WPA2-protected 2.4 GHz Wi-Fi hotspot, but switches to a completely unprotected 5 GHz hotspot for large transfers.

Google and Oppo use 8-character PSKs that allow brute-forcing within 30 minutes on a single RTX5090 GPU. Xiaomi deterministically derives its PSK from a random token embedded in the SSID, which an **M1** attacker can simply replicate. Exact Wi-Fi configurations for these apps are listed in Section 9.2. Only two apps use some payload encryption (**R3**). Vivo employs TLS for all payloads. Oppo only encrypts contacts, SMS/MMS, call logs, and Wi-Fi credentials using an RSA key negotiated during handshake.

S2: Manipulating Wireless Communication. Huawei, Honor, and Vivo use very weak all-numeric PSKs that can be realistically brute-forced within 15 seconds (**R4**). An attacker can, e.g., use 2 RTX5090 GPUs or rent a suitable AWS E2C instance for \$13/hr[4]. Samsung can be identified through vendor fields in Wi-Fi Direct beacons (**R1**), and target device identification is ambiguous (**R5**; see Section 6). Huawei and Honor do not use OOB secrets (**R6**), thus being unprotected against MITM attacks. Their pairing QR codes only contain the Wi-Fi SSID and PSK. Vivo's TLS implementation trusts any server certificate. Samsung encrypts certain payloads during transmission, but transmits the symmetric key in plain. Samsung and Vivo do not employ any integrity checks (**R7**). Xiaomi uses MD5 for some payloads, but not protected via OOB secrets.

S3: Extracting data via local socket connection. Only Samsung and Vivo open listening sockets on the old device. Samsung only accepts the first connection, which is used by the legitimate client. Vivo allows arbitrary payload connections (**R8**), and its TLS server uses no client authentication (**R3/R9**).

S4: Injecting data via socket connection to new device. All tools use app-local Wi-Fi connections for third-party data migrations. Oppo, Vivo, and Xiaomi additionally support manual pairing through the system's Wi-Fi settings, leading to device-global connections. For first-party migrations, all tools except Google use the legacy join API, resulting in device-global connections (**R10**). All these apps open a listening socket on the new device. Samsung only accepts a single connection. Huawei and Honor use limited parallel payload connections, but their legitimate clients exhaust the allowed number of connections. These apps are unsusceptible to **S4**, as consuming such a connection for the attack would visibly disturb the transmission, violating the goals from Section 4. Xiaomi accepts an arbitrary number of payload connections, but requires each client to prove

[4]AWS EC2 g6e.12xlarge with 4 NVIDIA L40S GPUs (9.4 MH/s total).

knowledge of the old device's random UUID, which the attacker cannot know. Oppo allows arbitrary payload connections (**R11**) without effective authentication (**R9**). Although Vivo also accepts arbitrary connections on the new device, these are not for data payloads. Oppo does not implement payload encryption (**R3**) for APK files, which means they can be manipulated for code execution. Although Oppo transmits SHA256 hashes for APK files, they are never enforced on the receiving side.

S5: Extracting data through file system. Huawei, Honor, and Samsung write files to public storage during a data migration. For Huawei and Honor, these files only store insensitive analytics data. Samsung uses public storage for caching user data on the old device prior to transmission **(R13)**. Specifically, this concerns all transmitted data except pictures, videos, and documents, for both wired and wireless data migrations. For first-party migrations, the stored user data includes, e.g., Wi-Fi credentials, which would otherwise never be accessible to third-party apps. Although some files are encrypted, the used symmetric encryption key is written to public storage after each data migration **(R14)**.

S6: Injecting data through file system. Only Samsung stores user data in public storage (**R13**). Some data, such as Wi-Fi credentials, is unencrypted. Other data, including APK files, is encrypted. Although the app writes the symmetric encryption key into public storage, it only does so at the end of a data migration session. The key is randomly generated per session. However, the APK encryption scheme only encrypts the first 1 MB of every APK file (**R14**). As shown in Section 6, an attacker can exploit properties of the APK file format to construct a modified APK file whose encrypted prefix stays unchanged. No integrity checks are included (**R15**).

5.4. Summary of Results

As the overview in Table 6.5 shows, all tools are susceptible to at least one of our attack scenarios. Notably, all apps are vulnerable to realizations of at least one scenario under attacker model **M1** (wireless), which is the attacker model that requires no prior privileges. The most realizable attack scenarios across vendors are **S1** and **S2**, i.e., wireless eavesdropping and MITM. Both work against 5 of the 7 (71 %) examined apps. Overall, the examined apps are much less susceptible to **M2** (local) attack scenarios, which in total affect 3 of the 7 (43 %) apps.

	Attack Scenario					
	Proximity (M1)		Local (M2)			
App	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**
Google	⚠	-	-	-	-	-
Samsung	-	⚠	-	-	⚠	⚠
Vivo	⚠	⚠	⚠	-	-	-
Xiaomi	⚠	⚠	-	-	-	-
Oppo	⚠[1]	-	-	⚠[2]	-	-
Huawei	⚠	⚠	-	-	-	-
Honor	⚠	⚠	-	-	-	-

[1] Not Wi-Fi Configs, SMS/MMS, Call Log, Contacts.
[2] Only first-party migrations or manual pairing.

Table 6.5: Attack susceptibility per data migration tool.

The most vulnerable data migration tools in our test set are Samsung and Vivo, each susceptible to 3 attacks. This is concerning, as besides Google, these have the highest download count in our set (both > 1 billion). The most secure of the analyzed apps is Google, which is only affected by one attack.

6. End-to-End Attacks

We implemented end-to-end attacks for all possible attack scenarios identified through our analyses (see Table 6.5). The implementations for **S1** attacks are very similar across tools. We capture Wi-Fi frames, derive or brute-force the weak WPA2 PSK to decrypt the frames, and extract payloads using binwalk. For **S2** attacks, we use app-specific techniques (described below) against Samsung, DHCP spoofing against Vivo, and MC-MITM (cf. Section 9.1) against Xiaomi, Huawei, and Honor. Due to space constraints, we focus this section on **S2**, **S5** and **S6** against Samsung, a particularly popular yet vulnerable app.

Manipulating Wireless Communication (S2). Samsung uses Wi-Fi Direct as the wireless link. The app on the new device starts advertising as a Wi-Fi Direct peer and displays a QR code for scanning through the app on the old device. The code contains a 32-byte random shared secret S, as well as $H_N = HMAC_S(\mathit{Suffix}(MAC_{New}, 3))$, an HMAC of the last three bytes of the new device's MAC address. After scanning the QR

code, the old device starts Wi-Fi Direct peer discovery, calculating $H_C = HMAC_S(Suffix(MAC_{Candidate}, 3))$ for each discovered MAC address. If $H_C == H_N$, the old device negotiates a Wi-Fi Direct group with this peer. Once connected, both devices prove possession of S through an authentication handshake that includes a random nonce N and HMAC $H_A = HMAC_S(N)$ from both parties. Additionally, the new device uses S to encrypt the subsequent message that contains its IP address and used port, which the old device then tries to connect a socket to.

This pairing scheme is vulnerable to a MITM attack. The attacker scans for Wi-Fi Direct peers until a device is discovered whose Wi-Fi beacons identify it as a Samsung smartphone. At this point, the attacker spoofs the last three bytes of the MAC address in the Wi-Fi beacons and starts to advertise as a Wi-Fi Direct peer as well. Although the attacker does not know S, their MAC address will now qualify them for group negotiation with the old device. Depending on which peer the old device discovers first, it will either negotiate a group with the new device or the attacker. The attacker can detect group formation with the new device by a new group SSID being broadcast. In this case, the attacker forces the user to repeat the pairing procedure by sending Wi-Fi de-auth frames. Once negotiation with the old device has succeeded, the attacker requests group negotiation with the new device on a second Wi-Fi interface. Through its gained MITM position, the attacker then relays the authentication handshake between the new and old device. Although the attacker does not know S, they are now authenticated to both legitimate devices. The attacker replicates the IP allocation scheme of Android's Wi-Fi Direct implementation, which always assigns the same address to the group owner. Next, the attacker opens a server socket on the hardcoded port Samsung uses. As a result, the attacker can simply relay the subsequent encrypted message containing the IP address and port unaltered. There is no cryptographic link between the OOB secret S and actual payload transmission. Some data payloads in the transmission are encrypted, but using a symmetric key that is transmitted in plain before the payload. We successfully used this attack to extract pictures, videos, documents, call logs, messages, contacts, and Wi-Fi credentials from a data migration. Additionally, we demonstrated that the attacker can get code execution in any migrated app by replacing its APK data as it is transmitted.

Reading (S5) & Manipulating (S6) Cache Files. Samsung Smart Switch caches most user data in public storage during a transmission. Our **S5** attack can extract unencrypted Wi-Fi credentials, or use the key

Samsung writes to disk after each transmission to, e.g., decrypt encrypted Samsung account credentials. Although cached APK files are encrypted, the encryption scheme only covers the first 2^{20} bytes (1 MiB) of data. Our **S6** attack constructs a special APK file that prepends the first 2^{20} bytes of the unencrypted target APK to the attacker's payload APK. The attacker can simply determine the package name of the target APK from its file name and use unprivileged framework APIs to retrieve its unencrypted APK contents. Next, the attacker adjusts the offsets in the ZIP Central Directory to account for the added bytes and signs the resulting APK. Finally, the attacker replaces the prefix with the corresponding ciphertext from the encrypted APK.

7. Discussion

This section discusses potential limitations of our analyses and solutions for mitigating attacks in device-to-device communication.

Analysis Precision. Some of our procedures for determining presence of individual attack requirements use manual reverse-engineering. Like with any static analysis technique, imprecisions cannot be ruled out. However, we implemented working end-to-end exploits for all identified attacks, which we included in vendor disclosures. We thus argue that the core findings of our analysis faithfully reflect the security posture of state-of-the-art Android data migration tools.

Wired Data Migration. Some data migration tools support wired operation via USB connections, which are insusceptible to our most powerful (**M1**) attacks. However, we anticipate that wireless connections are used in most data migrations for two reasons. (1) Only one tested tool (Samsung) supports (optional) wired migrations from Android devices. Although a legacy support document mentions wired migrations for Google [19], the rebranded Android Switch no longer offers this option [20]. (2) In a USB connection between two phones, one of them charges from the other, quickly draining its battery.

Mitigations. MITM or eavesdropping attacks on device-to-device communication can effectively be mitigated by encrypting all exchanged traffic through an OOB secret. An OOB secret of sufficient entropy can, e.g., be passed between devices through a QR code or derived from a short user-entered pin via a password-authenticated key exchange (PAKE) protocol

such as Secure Remote Password (SRP) [47]. At the time of our analysis, none of the evaluated apps integrated such a mitigation.

8. Conclusion

In this paper, we presented a systematic security analysis of data migration tools in the Android ecosystem. Based on an abstracted view of the operation of these apps, we defined six attack scenarios. We then identified necessary implementation details for a specific scenario to be realizable for a given app, arriving at 15 attack requirements. These requirements allowed us to systematically evaluate the susceptibility for each app and attack scenario pair. Our evaluation for seven widely used data migration tools uncovered severe vulnerabilities across the industry. All apps were susceptible to wireless attacks from an attacker within Wi-Fi range of the victim. Of the seven apps, three were also susceptible to local attacks, e.g., from malware. We presented end-to-end attacks on all analyzed data migration tools and detailed two of them for Samsung Smart Switch, a particularly popular yet vulnerable data migration tool. We anticipate our systematic cross-vendor study will provide value for practitioners and developers and motivate lasting improvements to this critical part of the Android ecosystem.

9. Appendix

9.1. Multi-Channel Machine-In-The-Middle Attack

In this section, we document our adaptations to multi-channel machine-in-the-middle (MC-MITM) attacks as presented by Vanhoef et al. [42]. These attacks allow an attacker within Wi-Fi range to gain a MITM position between a Wi-Fi STA and AP. They work by injecting Wi-Fi control frames for enforcing a frequency switch for STAs connected to the victim AP, but not for the AP itself. The attacker can then relay between the two different frequencies, effectively granting it a MITM position between the STAs and the AP. Vanhoef et al. [42] focused their efforts on link-level attacks operating on the encrypted Wi-Fi frames, enabling, e.g., denial-of-service attacks. However, the approach can be extended to scenarios where the attacker needs to manipulate the actual data payloads within Wi-Fi

frames. For this to work in general, the attacker needs to obtain the PTK, i.e., know the WPA2 PSK and observe the handshake. We further adapt the technique so that it offers the attacker a time window for brute-forcing the WPA2 PSK. This can be accomplished by intentionally dropping (i.e., not forwarding) all communication for a certain time span following the WPA2 handshake. During this time, the attacker brute-forces the PSK and PTK from the captured handshake. Once the PSK and PTK have been brute-forced, the attacker continues to relay between STA and AP, but can now decrypt, manipulate, and re-encrypt the transmitted payloads. In our experiments on multiple Android devices, we could consistently drop Wi-Fi communication for at least 15 s without any negative consequences. When using longer time windows, the OS (and therefore any app using the system's Wi-Fi APIs) considered the Wi-Fi network non-functional and disconnected from it. Accounting for overhead caused by the Python implementation of the attack, this observation matches the 18 s timeout Android implements for awaiting a DHCP lease[5]. For the attack to work, the attacker therefore needs to be able to brute-force the WPA2 PSK within 15 s. The (simplified) steps the attacker takes are:

1. Scan for Wi-Fi beacons until a data migration is detected through the app's unique SSID pattern.
2. Start a Wi-Fi AP on a rogue channel and send Channel Switch Announcement (CSA) beacons to force STAs onto the rogue channel.
3. Capture Wi-Fi frames on the corresponding channel until a WPA2 handshake is observed.
4. Once the link-level MITM position is established, stop forwarding frames to the AP for 15 s.
5. During this time, brute-force the WPA2 PSK based on the captured WPA2 handshake and known PSK pattern for the app.
6. Use the brute-forced PSK to decrypt, manipulate, and re-encrypt the Wi-Fi frames relayed between STA and legitimate AP.

We successfully implemented this attack technique in Python using the open-source Scapy library. For being able to carry out complex modifications to TCP streams contained in the Wi-Fi frames, we route the traffic

[5] Default timeout defined at `https://cs.android.com/android/platform/superproject/main/+/main:packages/modules/NetworkStack/common/networkstackclient/src/android/net/shared/ProvisioningConfiguration.java;drc=497ee73d99bf8331974955c844581efd46e83bea;l=85`

App	SSID Pattern	WPA2 PSK
Google	DIRECT-[0-9A-F]{2}-[0-9A-F]{6}	[0-9A-F]{8}
Vivo	vivo#[■]#[0-9A-Z]{2}(_RE)?	None or [0-9]{8}
Xiaomi	[■]_[0-9a-zA-Z]{7}_MI	Derived from SSID
Oppo	oplus_co_ap[a-z]{4}	[a-z]{4}[0-9]{4}
Huawei	[■]%[0-9]{4}%CloudClone	[0-9]{8}
Honor	[■]%[0-9]{4}%CloudClone	[0-9]{8}

Table 6.6: Wi-Fi Hotspot configurations used by data migration apps. ■ represents manufacturer-specific model identifiers.

through the kernel TCP/IP stack via TUN interfaces on the attacker machine running Linux.

9.2. Wi-Fi Hotspot Configurations

Table 6.6 displays the Wi-Fi hotspot configurations used by the analysed data migration apps. Samsung was omitted due to using Wi-Fi Direct.

References

[1] Android Open Source Project. Android Compatibility program overview. 2025. URL: https://source.android.com/docs/compatibility/overview (p. 139).

[2] Android Open Source Project. API Reference: WifiManager.addNetwork(). 2025. URL: https://web.archive.org/web/20250729133834/https://developer.android.com/reference/android/net/wifi/WifiManager#addNetwork(android.net.wifi.WifiConfiguration) (p. 140).

[3] Android Open Source Project. API Reference: WifiP2pManager.createGroup(). 2025. URL: https://developer.android.com/reference/android/net/wifi/p2p/WifiP2pManager#createGroup(android.net.wifi.p2p.WifiP2pManager.Channel,%20android.net.wifi.p2p.WifiP2pConfig,%20android.net.wifi.p2p.WifiP2pManager.ActionListener) (p. 139).

[4] Android Open Source Project. Use a local-only Wi-Fi hotspot. 2025. URL: https://developer.android.com/develop/connectivity/wifi/localonlyhotspot (p. 139).

[5] Android Open Source Project. Wi-Fi Direct (peer-to-peer or P2P) overview. 2025. URL: https://web.archive.org/web/20250729122558/https://developer.android.com/develop/connectivity/wifi/wifip2p (p. 140).

[6] Android Open Source Project. Wi-Fi hotspot (Soft AP). 2025. URL: https://source.android.com/docs/core/connect/wifi-softap (p. 140).

[7] Android Open Source Project. Wi-Fi Network Request API for peer-to-peer connectivity. 2025. URL: https://web.archive.org/web/20250729122716/https://developer.android.com/develop/connectivity/wifi/wifi-bootstrap (p. 140).

[8] Android Open Source Project. Wi-Fi suggestion API for internet connectivity. 2025. URL: https://developer.android.com/develop/connectivity/wifi/wifi-suggest (p. 140).

[9] AppBrain. Android Statistics: Top Manufacturers. 2025. URL: https://web.archive.org/web/20250702234143/https://www.appbrain.com/stats/top-manufacturers (p. 137).

[10] BestBuy. GeekSquad: Setup for New Devices. 2025. URL: https://www.bestbuy.com/site/setup-for-new-devices/9131012.p?skuId=9131012&intl=nosplash (p. 141).

[11] Sheng Cao, Hao Zhou, Songzhou Shi, Yanjie Zhao, and Haoyu Wang. Parcel Mismatch Demystified: Addressing a Decade-Old Security Challenge in Android. In: CCS. 2025 (p. 139).

[12] Cell Phone Repair by Assurant. Cell Phone Data Transfer. 2025. URL: https://www.cellphonerepair.com/common-issues/data-transfer (p. 141).

[13] Stefano Cristalli, Long Lu, Danilo Bruschi, and Andrea Lanzi. Detecting (absent) app-to-app authentication on cross-device short-distance channels. In: ACSAC. 2019 (pp. 140, 143).

[14] Soteris Demetriou, Xiaoyong Zhou, M Naveed, Y Lee, K Yuan, X Wang, and CA Gunter. What's in Your Dongle and Bank Account? Mandatory and Discretionary Protection of Android External Resources. In: NDSS. 2015 (p. 140).

[15] Florian Draschbacher and Lukas Maar. Manifest Problems: Analyzing Code Transparency for Android Application Bundles. In: ACSAC. 2024 (p. 139).

[16] Florian Draschbacher, Lukas Maar, Lorenz Schumm, Rene Denifl, Treffner Lukas, and Stefan Mangard. Clone2Pwn: A Systematic Security Analysis of Data Migration Tools in the Android Ecosystem. In: ESORICS. 2026 (p. 133).

[17] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware. In: USENIX Security. 2020 (p. 139).

[18] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. In: International Workshop on Selected Areas in Cryptography. 2001 (p. 139).

[19] Google. Copy apps & data from an Android to a new Android device. Accessed: October 08, 2025. 2023. URL: https://support.google.com/android/answer/13761358?hl=en (p. 155).

[20] Google. How to transfer data between Android phones. Accessed: October 08, 2025. 2025. URL: https://web.archive.org/web/20251008132752/https://www.android.com/transfer-data-android-to-android/ (p. 155).

[21] Inc Google. Google Play: Android Switch. Accessed: August 5, 2025. 2024. URL: http://web.archive.org/web/20250805184017/https://play.google.com/store/apps/details?id=com.google.android.apps.restore (p. 137).

[22] Institute of Electrical and Electronics Engineers. Wireless Local Area Networks. 2025. URL: https://www.ieee802.org/11/ (p. 138).

[23] Changyu Li, Quanpu Cai, Juanru Li, Hui Liu, Yuanyuan Zhang, Dawu Gu, and Yu Yu. Passwords in the Air: Harvesting Wi-Fi Credentials from SmartCfg Provisioning. In: WiSec. 2018 (p. 140).

[24] TP-Link. General questions about Wi-Fi range of TP-Link wireless products. Accessed: August 5, 2025. 2021. URL: https://www.tp-link.com/en/support/faq/2291/ (p. 138).

[25] Kecheng Liu, Wenlong Shen, Yu Cheng, Lin X. Cai, Qing Li, Sheng Zhou, and Zhisheng Niu. Security Analysis of Mobile Device-to-Device Network Applications. In: IEEE Internet of Things Journal (2019) (p. 140).

[26] Gordon Lyon. The Art of Port Scanning. In: Phrack Magazine (1997) (pp. 143, 145).

[27] Siqi Ma, Hehao Li, Wenbo Yang, Juanru Li, Surya Nepal, and Elisa Bertino. Certified Copy? Understanding Security Risks of Wi-Fi Hotspot based Android Data Clone Services. In: ACSAC. 2020 (pp. 136, 140, 141, 143).

[28] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The Android Platform Security Model. In: CoRR (2023) (pp. 139, 142).

[29] Alessio Merlo, Antonio Ruggia, Luigi Sciolla, and Luca Verderame. You Shall not Repackage! Demystifying Anti-Repackaging on Android. In: Computers & Security (2021) (p. 142).

[30] Bradley Mitchell. What Is the Range of a Typical Wi-Fi Network? Accessed: August 5, 2025. 2020. URL: https://www.lifewire.com/range-of-typical-wifi-network-816564 (p. 138).

[31] National Security Agency. Ghidra software reverse engineering (SRE) framework. Accessed: July 31, 2025. 2019. URL: https://github.com/NationalSecurityAgency/ghidra (p. 146).

[32] Muhammad Naveed, Xiao-yong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A Gunter. Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android. In: NDSS. 2014 (p. 140).

[33] Oversecured. Oversecured automatically discovers persistent code execution in the Google Play Core Library. Accessed: March 11, 2026. 2020. URL: https://blog.oversecured.com/Oversecured-automatically-discovers-persistent-code-execution-in-the-Google-Play-Core-Library/ (p. 139).

[34] Ole André V. Ravnås. Frida: Dynamic instrumentation toolkit. Accessed: July 31, 2025. 2014. URL: https://frida.re (p. 149).

[35] Skylot. JADX: Dex to Java decompiler. Accessed: July 31, 2025. 2015. URL: https://github.com/skylot/jadx (p. 146).

[36] SmarTone. Data Transfer Service. 2025. URL: https://www.smartone.com/en/privileges_and_support/support/device_support/data_transfer.jsp (pp. 137, 141).

[37] Wenna Song, Ming Jiang, Han Yan, Yi Xiang, Yuan Chen, Yuan Luo, Kun He, and Guojun Peng. Android Data-Clone Attack via Operating System Customization. In: IEEE Access (2020) (p. 140).

[38] statcounter. Mobile Operating System Market Share Worldwide. 2025. URL: https://web.archive.org/web/20250708194001/https://gs.statcounter.com/os-market-share/mobile/worldwide (p. 139).

[39] Milan Stute, Alexander Heinrich, Jannik Lorenz, and Matthias Hollick. Disrupting Continuity of Apple?s Wireless Ecosystem Security: New Tracking, DoS, and MitM Attacks on iOS and macOS Through Bluetooth Low Energy, AWDL, and Wi-Fi. In: USENIX Security. 2022 (p. 141).

[40] Milan Stute, Sashank Narain, Alex Mariotto, Alexander Heinrich, David Kreitschmann, Guevara Noubir, and Matthias Hollick. A Billion Open Interfaces for Eve and Mallory: MitM, DoS, and Tracking Attacks on iOS and macOS Through Apple Wireless Direct Link. In: USENIX Security. 2019 (p. 141).

[41] ubreakifix by Asurion. Smartphone Data Transfers. 2025. URL: https://web.archive.org/web/20250430135435/https://www.ubreakifix.com/repairs/smartphones/services/data-transfer (p. 141).

[42] Mathy Vanhoef and Frank Piessens. Advanced Wi-Fi attacks using commodity hardware. In: ACSAC. 2014 (pp. 143, 156).

[43] Verizon. Verizon Community: Data Transfer Fee. 2024. URL: https://community.verizon.com/t5/Other-Network-Discussions/Data-Transfer-Fee/m-p/1750512/highlight/true (pp. 137, 141).

[44] Vodafone UK. What our Tech Team can do for you. 2025. URL: https://web.archive.org/web/20250729134247/https://www.vodafone.co.uk/help-and-information/tech-team (pp. 137, 141).

[45] Mingjun Wang and Zheng Yan. A Survey on Security in D2D Communications. In: Mobile Networks and Applications (2017) (p. 140).

[46] John Wu. Magisk: The Magic Mask for Android. Accessed: July 31, 2025. 2018. URL: https://github.com/topjohnwu/Magisk (p. 146).

[47] Thomas D Wu. The Secure Remote Password Protocol. In: NDSS. 1998 (p. 156).

[48] Brennan Yamamoto, Allison Wong, Peter Joseph Agcanas, Kai Jones, Dominic Gaspar, Raymond Andrade, and A Zachary Trimble. Received Signal Strength Indication (RSSI) of 2.4 GHz and 5 GHz Wireless Local Area Network Systems Projected over Land and Sea for Near-Shore Maritime Robot Operations. In: Journal of Marine Science and Engineering (2019) (p. 138).

7

Manifest Problems: Analyzing Code Transparency for Android Application Bundles

Publication Data

Florian Draschbacher and Lukas Maar. Manifest Problems: Analyzing Code Transparency for Android Application Bundles. In: ACSAC. 2024

Contributions

The author of this thesis is the main author of the paper. He came up with the idea for the paper, carried out all analyses and implemented all the described experiments and proof-of-concepts. The author also wrote most of the paper's text.

Manifest Problems: Analyzing Code Transparency for Android Application Bundles

Florian Draschbacher Lukas Maar

Graz University of Technology

Abstract

In 2018, Google introduced a new app distribution format called AAB (Android Application Bundle), which replaced APK (Android Package) as the required format for all new app submissions to Google Play in 2021. Apps are still delivered to end users as APK files, but they are now generated and signed on the app store operator's infrastructure. Most crucially, this change requires developers to hand over their APK signing key to the app store operator, enabling them to arbitrarily manipulate apps prior to delivery to end users. To address this, Google has introduced the Code Transparency scheme to verify the integrity of APKs generated from AAB files. However, due to the lack of independent studies, the exact security properties of Code Transparency remain unclear.

In this paper, we present the first comprehensive analysis of the security of Code Transparency and the AAB format. We thoroughly investigate the design and implementation of the Code Transparency scheme, discussing in detail the technical possibilities attackers have for manipulating apps that use it. Additionally, we conduct a large-scale study on AAB and Code Transparency in practice. To this end, we evaluate the prevalence of both technologies among 3.5 million real-world apps, analyze their susceptibility to our attacks, and carry out a case study that demonstrates the practical security implications of attacks on Code Transparency.

Our analyses indicate that Code Transparency suffers from severe design and implementation flaws that allow app store operators to execute code in the context of any app without disturbing its Code Transparency signature.

1. Introduction

The adaptability to a wide range of hardware platforms has been a significant advantage of Android in the competitive mobile OS market. However, it has also led to a fragmented device landscape and a wide range of security and usability issues that continue to plague the Android ecosystem. To address these issues, Google has made considerable efforts, including initiatives such as *Project Treble* [27], which simplified adapting OS updates to devices, *Project Mainline* [11] to modularize the OS into components updatable through Google Play, or *Android Jetpack* [6] which provides developers with app building blocks that reliably work across Android versions.

Another recent change in this series of initiatives was the introduction of the Android Application Bundle (AAB) format for packaging applications for submission to app stores in 2018 [24]. Until that point, developers usually shipped each application to app stores as a single universal APK file containing resources for all supported device configurations. App stores would then serve these exact files to end-user devices, where the resources designed for other device configurations would waste storage space. Although the new AAB submission format contains compiled code and resources for all possible device configurations as well, it only serves as an intermediary format. The app store operator uses it to *"generate and serve optimized APKs for each device configuration"* [13], which saves storage space and network bandwidth.

Since the introduction of the AAB format, Google has put a lot of effort into establishing it as the new distribution standard for the entire Android app industry. In 2020, the format was made mandatory for any new app submissions to Google Play [10]. Since May 2023, all apps or updates that support Android TV need to use the AAB format [39]. The format has also been adopted by all major third-party Android app stores, including Samsung's Galaxy Store, Huawei AppGallery, and Amazon Appstore.

However, besides improving the user experience, the AAB format has considerable consequences for the security architecture of the Android OS. The Android Platform Security Model [28] considers the application developer one of the three stakeholders that need to agree on sensitive operations being carried out on the device. Specifically, it is the application (on behalf of the application developer) that decides on the access to login credentials or other valuable user data stored in its private data folder.

Since the application encodes the developer's intentions regarding sensitive operations affecting the app and its data, it must remain unmodified through the entire supply chain between the developer and the end user. Any code injected into an application somewhere along the supply chain would later be executed in the process context of the application itself, where it is able to alter its security policies.

The security consequences of AAB became particularly problematic when Google in 2020 made it mandatory for any new app submission to Google Play [10]. After the developer community voiced its concerns regarding that change [29], Google in 2021 introduced the optional Code Transparency (CT) scheme for AAB files. It is intended to provide cryptographically proofable integrity guarantees for key parts of an Android application. Despite the security-critical role of CT for the AAB format, the reliability of its integrity guarantees has never been examined. Furthermore, it remains entirely unclear how and if the scheme is used in practice, whether it is deployed correctly, and whether its design and implementation align with the reality of modern mobile applications.

In this paper, we present the first comprehensive analysis of the security of CT for the AAB format. We establish attack scenarios against CT based on Google's official documentation, related supply-chain hardening techniques and the Android Platform Security Model. From these attack scenarios, we identify three design flaws in the CT scheme's design, as well as three flaws in the *bundletool* reference implementation. As a key observation in our analysis, we note that CT's exclusive focus on Dalvik Executable (DEX) and Shared Object (SO) files entirely neglects the powerful role of resource files such as the app manifest on the Android platform.

Subsequently, we show how an attacker can exploit the identified flaws. We consider three attacker models of varying power that are in line with Google's Code Transparency threat model. In these scenarios, a supply chain attacker compromises the APK signing key to sign manipulated app builds. The CT is supposed to help identify these modifications. Yet, under all attacker models, the attacks we identify allow gaining code execution in the context of an application without invalidating its CT.

Furthermore, we investigate the use of AAB and CT in practice. To this end, we carry out automated analyses on more than 3.5 million Android applications from Google Play and Huawei AppGallery. Our results show that despite the serious security implications of the AAB

format it was designed to protect against, CT is almost non-existent (0.0014 % of apps submitted as AAB) in real-world applications. We also evaluate the eligibility for CT of the most popular apps from Google Play. We find that if they used CT, more than 50 % of them would be susceptible to code execution attacks even under the least powerful attacker model.

Overall, our results show that CT in its current form is mostly ineffective and virtually unused. This leaves developers unprotected against the security repercussions of the AAB format, which they are increasingly pressed to use. We therefore discuss possible improvements to CT and suggest app distribution solutions that offer the same advantages as AAB without compromising on security.

Contributions: To summarize, our key contributions are:

- We thoroughly analyze the design and implementation of Code Transparency for Android Application Bundles, identifying multiple flaws.
- We demonstrate attacks that exploit the identified flaws to gain code execution or data access in the context of target applications without invalidating their CT.
- We conduct the first large-scale survey of CT in practice, analyzing applications from Google Play and Huawei AppGallery. We evaluate their use of AAB and CT and their susceptibility to the vulnerabilities we identified in CT's integrity guarantees.

Disclosure: We disclosed our findings to Google and are currently discussing how to mitigate the attacks we identified, ultimately improving Android security. One of the reported issues already received a CVE[1] and was fixed in Android 14.

Outline: Section 2 provides background on the Android platform. Section 3 introduces the Android Application Bundle format and Code Transparency scheme. In Section 4, we discuss our attack scenarios, which we use in our security analysis in Section 5. Section 6 presents attacks based on the identified flaws, before Section 7 evaluates Code Transparency in practice. We discuss our findings in Section 8, elaborate on related work in Section 9, and conclude our paper in Section 10.

[1] CVE-2023-21387

2. Background

2.1. Android and its Security Architecture

Android is the most popular operating system for mobile devices, most notably for smartphones. To facilitate adoption by handset manufacturers, the platform is based on a Linux kernel and developed as an open-source project led by Google. Android supports user-installed software, which is executed inside a sandbox that enforces strict isolation between applications. Most users download apps from the platform's official Google Play app store, although the platform (through a user setting) allows installing applications from arbitrary sources. This possibility has led to the emergence of third-party app stores.

2.1.1. Android Platform Security Model

In 2018, key security architects in the Android team published the Android Platform Security Model [28], which formalizes the core principles of Android's security architecture. The platform uses a multi-party consent model, where operations that affect apps must be agreed upon by the end user, the operating system (OS), and the application developer (through policies expressed in the app code). This consent model spans the traditional notion of subjects (e.g., users and processes) accessing objects (e.g., files and network sockets). The party that creates an object is implicitly granted control over it. For example, the system automatically creates a private data folder for every app. The user's consent to this action is given through triggering the app installation. The OS ensures that no other application may access this folder without consent from the three main parties.

2.1.2. Permission System

The only way for an app to utilize re- sources shared with the rest of the system is through well-defined APIs under tight control of the OS kernel and system framework. For accessing most of these APIs, apps need to obtain the corre- sponding permission from the OS. Any permission an app may request at runtime needs to be statically declared to the OS in the app's application manifest (see Section 2.2). While the user

implicitly grants an app's declared permissions during installation, some particularly dangerous permissions require explicit user consent during runtime. The permissions known to the Android OS are grouped into different protection levels. Only a subset of these levels is accessible to user-installed applications.

2.1.3. Privileged Applications

The Android security architecture assumes some applications preinstalled on the device are more trustworthy than user-installed applications. They, therefore, are allowed to request more privileged permissions belonging to the *privileged* protection level. We refer to these applications as *privileged applications* in the following. Crucially, even if these apps hold a privileged position on the device, they may *not* access any other (less-privileged) app's process space, influence its execution, or access its private data directory.

2.2. Android Package Format

Android applications run in the Android-specific ART Java runtime, which requires code to be compiled to the special Dalvik bytecode format. Performance-critical functionality can be provided as native machine code in ELF shared object files. Applications need to be packaged into an Android Application Package (APK) file before they may be installed on a device. These files are signed ZIP archives whose contents follow a specific structure. The main elements are:

- *DEX Files*: Dalvik Executable files that contain *managed code*, i.e. the Dalvik bytecode encoding the app's functionality. Any file named `classes.dex` or `classes`$\{n\}$`.dex` (for any $n > 1$) in the APK root is loaded into the app's classpath.

- *Shared Libraries*: ELF binaries that contain *native code* usually compiled from C or C++. Since they contain machine code, they require adaptation to every Instruction Set Architecture (ISA). Native libraries typically reside in the `libs` folder in the APK, from where they must be loaded via managed code.

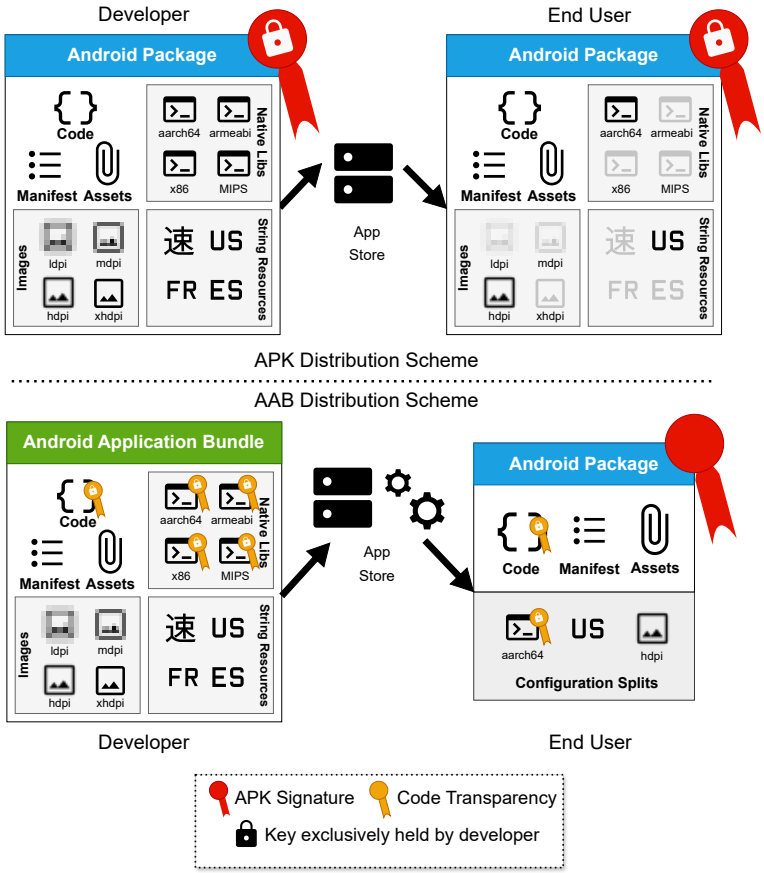


Figure 7.1: Comparison between old APK and new AAB distribution schemes. Unused resources and native libraries wasted storage space in the old APK distribution scheme. In the new AAB scheme, the app store serves optimized APKs for every device configuration. Note how the optional Code Transparency covers DEX files and native libraries, while the APK signature covers the entire APK file. APKs generated from AAB files are signed by the app store, which is now in possession of the app signing key. Only the Code Transparency key is exclusively held by the app developer.

- *Resources*: UI layout definitions, strings for localizations, and more that are compressed during compilation are stored in the `res` folder and indexed in `resources.arsc` in the APK's root. Resources carry a configuration qualifier, such as a language, screen density, or OS version. It is, e.g., possible to mark an image as high-dpi, so the Android framework loads it only on devices with a high pixel density.

- *Assets*: Arbitrary files that are not modified during compilation. These are stored in the `assets` folder in the APK.

- *Application Manifest*: `AndroidManifest.xml` in the APK root effectively serves as the contract between the app and the OS. It declares the app components (e.g., UI activities, broadcast receivers, content providers, etc) that are exposed to the rest of the system, as well as the system functionality (e.g., permissions) that the app wishes to use.

2.3. Universal APKs, Multi APKs, Split APKs

APK files that contain resources and native libraries for a variety of device configurations are called *Universal APKs*. Alternatively, developers may produce multiple APK builds of their app targeting different device configurations, called *Multi APKs*. Android devices (running Android 5.0 or newer) allow a single app to be installed from a collection of APK files, called *Split APKs*.

2.4. APK Signature

Before an APK file can be installed on an Android device, it needs to be signed. The signature covers all contents of the APK file and is only checked during installation. It serves as an integrity guarantee for an APK's delivery (no tampering between its creation and installation) and an app's installation (updates must come from the original app author). Since APK signing certificates are usually self-signed, the APK signature does not provide any means for authenticating the developer of an app. By ensuring that app updates are signed with the same certificate as the original installation, the signature protects the app's private data.

For the first decade of the Android OS's existence, developers typically were responsible for packaging and signing APK files for their apps. They then submitted the signed APK files to app stores, which would relay them to end users. Since they would retain exclusive control over the APK signing key, developers could use hardware-assisted app attestation [31] at runtime to ascertain that the executed APK file had not been manipulated.

3. Android Application Bundles and Code Transparency

With the Android Application Bundle (AAB) format, Google introduced a distinction between the publishing and delivery formats of Android applications. While the Android OS still only accepts app installations from APK files, developers now submit their apps to distributors (such as app stores) in the new AAB format. The distributor then generates APK files optimized for end-user devices from the submitted AAB file. The difference between the old APK and the new AAB distribution scheme is illustrated in Figure 7.1.

3.1. AAB File Structure

Like the APK file format, AAB is based on ZIP archives. However, an AAB file retains more of the high-level structure from the app project, facilitating the generation of optimized APK files later on. More specifically, data inside an AAB file is organized in modules. Every app is required to have a base module shipped with every installation. Further app functionality may be clustered into additional modules that can be installed on demand later. Every module inside the AAB is organized in its own top-level folder. Each module folder contains a file structure that is very similar to that inside an APK file. A noteworthy difference is that all DEX files that are to be copied into the root folder of generated APK files are stored in a subfolder named `dex` inside each module folder.

3.2. Bundletool

Google made all tooling required for building and processing AAB files available under an open-source license in the *bundletool* project[2] to facilitate adoption by third-party app stores. *bundletool* is used by Google Play and third-party app stores, as well as by the Android Gradle Plugin that compiles app projects in the official Android Studio IDE [15]. Additionally, it may be invoked directly through a command-line interface. The tool provides functionality for creating AAB files from a set of individual app modules or for generating a set of optimized APK files from an AAB file.

[2]bundletool. https://github.com/google/bundletool

3.3. APK Generation

When building APKs from an AAB file, *bundletool* will generate a set of *Split* APKs. These include a base APK for the base module and several configuration splits containing different variants of resources and native libraries. Finally, Split APK files will also be generated for any feature modules and their configuration splits. The possible dimensions for generating configuration splits are language, screen density, and processor Application Binary Interface (ABI)[3]. For example, for a complete installation of an app, it may be necessary to install the base split, a configuration split for high-density screens, one for the English language, and another for native libraries in the `arm64-v8a` ABI. Typically, app stores automatically download all Split APK files suitable for the user's device configuration. At the developer's discretion, feature modules may either be distributed as add-ons the base module dynamically downloads on demand or already as part of the initial installation.

The AAB format was designed to also support legacy devices incompatible with Split APKs. For apps targeting OS versions prior to Android 5.0, *bundletool* therefore also generates Multi APKs for all possible combinations of APK splits.

3.4. Code Transparency

This work refers to Google's definition of Code Transparency (CT): *"CT is an optional code signing and verification mechanism for apps published with the Android App Bundle"* [16]. This mechanism is only implemented in external tooling. There is no infrastructure in the Android OS itself for verifying the CT of APK files.

A developer can add CT to an application through *bundletool*'s command line interface or through its Android Gradle Plugin integration. This process injects a JSON Web Token (JWT) file into the app's AAB. This JWT file *"contains a list of DEX files and native libraries included in the bundle, and their hashes"* [16]. Listing 3.1 shows an example of the content of a CT JWT file. The list in the JWT file is signed with the developer's private CT key, which they are supposed never to share with another party. The CT JWT file is copied into all APK files generated from the AAB.

[3]The ABI here roughly corresponds to a processor ISA.

```
{
  "codeRelatedFile": [
    {
      "path": "base/dex/classes.dex",
      "type": "DEX",
      "sha256": "7b22...d19e"
    },
    {
      "path": "base/lib/arm64-v8a/libjsc.so",
      "type": "NATIVE_LIBRARY",
      "apkPath": "lib/arm64-v8a/libjsc.so",
      "sha256": "b63e....7a49"
    },
  ],
  "version": 1
}
```

Listing 3.1: The signed content of a Code Transparency JWT

CT was designed to be verifiable by both developers and end users. *Bundletool* provides functionality for checking whether the JWT in an APK contains a valid hash for all the app's DEX or SO files. End users also need to verify that the key used for signing the JWT is the legitimate developer's CT key. This requires an additional secure channel between the developer and user to communicate the legitimate key.

3.5. Code Transparency Attack Scenarios

Android CT is intended to thwart supply-chain attacks that happen in the app distribution channel between an app's developer and an end user. As stated by Google, verifying the CT gives the developer *"assurance that, even if the APK itself was re-signed during distribution, the code verified by CT hasn't been modified"* [14]. CT verification is used *"for the purpose of inspection by developers and end users, who want to ensure that code they're running matches the code that was originally built and signed by the app developer"* [16]. Since Google has not provided a definition for the addressed supply-chain scenarios in the documentation for CT, we refer to Google's Android Binary Transparency scheme [17]. It is the equivalent

of CT for protecting firmware images against supply-chain attacks. There, Google states that *"software supply chains are increasingly vulnerable to attacks, ranging from compromised signing keys to surreptitious code injection to insider attack"* [17]. In fact, similar events have happened in the past. For example, platform signing keys from Android vendors were found to have been leaked in 2022, enabling attackers to install apps with system privileges on affected devices[4]. There have also been examples in the past of CA key compromises [33], which play a similarly important role in the Internet's Public Key Infrastructure as app signing keys do on Android. As evident from the existence of CT for AAB, Google considers these realistic scenarios in app distribution as well.

4. CT Attacker Models

This section discusses the attacker goals and models we use in our analyses. All attacker models are based on the assumption that an attacker exploits an app store's access to APK signing keys (as necessary for AAB) to produce and distribute malicious APKs that bear the legitimate signing certificate. On paper, Code Transparency promises the possibility to detect such attacks.

4.1. Attacker Goals

In the subsequent analyses in Section 5, we consider a scenario in which a malicious party aims to compromise a concrete *app installation*, i.e., aims to deliver a manipulated target application to a specific victim user. Moreover, we assume that the attacker wishes to operate stealthily, i.e., by only temporarily deploying the manipulated app as an update to a legitimate installation and later rolling back all changes. It is worth noting that compromising an APK that bears the legitimate APK signature is particularly desirable for an attacker. APKs modified in this way cannot be detected through app attestation and can be installed as updates to a legitimate install, thereby gaining access to all its private data.

In line with the Android Platform Security Model [28], we assume the attacker wants to compromise the main assets of an application: First,

[4]Issue 100: Platform certificates used to sign malware. `https://bugs.chromium.org/p/apvi/issues/detail?id=100`

by gaining control over an application's execution flow (i.e., gaining code execution), the attacker can perform arbitrary actions from the target application's context. Second, by gaining control over a target application's private data directory, the attacker can extract sensitive data, e.g., credentials that they may misuse for user impersonation.

To summarize, we identify these two attacker goals that CT's integrity guarantees should prevent:

- **G1: Code injection.** Any attempt of an attacker to manipulate an APK for changing its runtime behavior should lead to the APK failing CT verification.
- **G2: Data access.** Any attempt of an attacker to manipulate an APK for gaining access to the application's private data directory should lead to the APK failing CT verification.

Please note that our analyses assume apps are installed and executed on an Android system that passes the Android Compatibility Test Suite, i.e., that runs a fully patched (as of August 2023) untampered (unrooted) specification-compliant OS.

4.2. Attacker Models

All attacker models we consider in our analyses assume an attacker holds some position in the app supply chain between the developer and the end user. Through this position, the attacker gets access to the APK signing key for producing APK files that can be installed as an update to an original legitimate APK. We assume that developers are aware of the possibility for app manipulations in the supply chain when using the AAB format and precautionarily employ CT as a means to detect these manipulations. We further distinguish the attacker models in the degree of access the attacker has to the victim's device. Two attacker models assume an attacker who has compromised an app store operator's infrastructure (e.g., as an inside attacker). The last attacker model considers scenarios in which only the APK signing keys are compromised. The existence of CT shows that Google considers these scenarios to be realistic.

- **M1: Privileged app store.** In addition to being in possession of the APK signing key, the attacker can execute code from a privileged application installed on a target device. It is crucial to note that without

modifying the target APK, even a privileged application is subject to the access policies described in the Android Platform Security Model [28]. In particular, it does not have any means to execute code in the context of another application or access its private data folder. Virtually all Android vendors operate their own privileged app stores, which, if compromised, allow an attacker to act under this model. Examples would be the *Google Play Store* on most Android devices, or *Huawei App Gallery* on devices produced by Huawei.

- **M2: Unprivileged app store.** The attacker may execute code in an unprivileged (user-installed) app store application on the victim device. A multitude of third-party app stores are available for Android, which, if compromised, can be exploited to the effect of this attacker model.
- **M3: Other role in the supply chain.** The attacker does not have any means to execute code on the victim device other than through injecting it into the target application.

5. Security Analysis of CT

In this section, we discuss the security flaws we identify in CT for AAB from analysing the official documentation and the source code of both *bundletool* and the Android Open Source Project. We distinguish between flaws inherent to the CT design and those that only affect the *bundletool* reference implementation.

5.1. Design Flaws in CT

Based on the security goals of CT we establish in Section 4, we identify the following flaws in its design.

F1: Optionality

Although intended to mitigate the severe security repercussions of the AAB distribution scheme, CT is only an optional feature. In fact, developers may only learn about its existence through subpages hidden in the Android documentation. While hampering the broad adoption of CT, its optionality also has serious consequences for the scheme's security. It means that

users cannot know whether an APK lacking CT was submitted like this by the developer or subjected to its malicious removal somewhere along the supply chain.

F2: Scope

CT lacks coverage of key possibilities for manipulating an app's behavior. More specifically, the application's manifest is not covered, nor are its assets or resources. Since applications are free to include executable code in formats of their choosing at arbitrary locations in their APKs, this leaves severe gaps in the integrity guarantees of CT.

F3: Communication Channel

The CT design leaves a key question unanswered. The official documentation notes that the *"public [CT] key of the developer [...] must be provided by the developer over a separate, secure channel"* [16]. While the general problem is not specific to CT but associated to public key distribution in general, the CT does not provide any clues as to how such a separate secure channel to the user may be established, given that all information published by the distributor must not be trusted.

5.2. Implementation Flaws in *bundletool*

In addition to the design flaws described above, we also identify several implementation flaws in the *bundletool* reference implementation of CT for AAB.

F4: Certificate Reuse

In the documentation for CT, Google states that the CT key *"should be a unique key that is different from the app signing key"* [16]. In fact, this is an essential requirement for the integrity guarantees the scheme aims to provide. Given that the app signing key must be shared with the app distributor, the latter may otherwise regenerate the CT after manipulating SO or DEX files. However, *bundletool* does not enforce this crucial requirement. The tool accepts the same key being used for both adding CT to an AAB and for generating APK files from the AAB. The

APK files produced in this way also pass all checks when verifying their CT through *bundletool*.

F5: DEX or SO in Assets

When an application includes DEX or SO files in its assets or other non-standard locations, verifying its CT fails even if the APK file has not been manipulated. This is due to an asymmetry between the implementations of CT generation and verification. While CT generation only considers DEX files in the root of the APK and SO files in the `lib` folder, verification ensures that all such files at any location in the APK have a valid entry in the JWT file.

As a result of this implementation flaw, developers of affected applications can not take advantage of CT to detect manipulations of their APKs. We note that, e.g., the *Facebook Audience Network SDK*[5], the most popular third-party advertisement library for Android [4], ships with a DEX file in its assets folder. As a result, about 13% of all apps on Google Play are affected by this problem [3].

F6: App Archives

By default, when generating APK files from an AAB, *bundletool* also generates an app archive APK. This is a lightweight placeholder APK sharing the app's package name and app signing key. It may be installed in place of the app's full APK to save storage space while retaining its private data directory. The placeholder APK contains code and resources necessary to redownload the full APK when needed.

The addition of app archiving to *bundletool* in 2022 and Google's integration of the feature in Google Play marked the first time an app store used the power gained through the AAB scheme to actively change app functionality. Given that app archiving is retroactively added during APK generation, the code for redownloading the full APK is not intuitively part of the CT. Indeed, when app archiving was added to *bundletool* in March 2022 (version 1.8.2), placeholder APKs did not contain any CT, even if the AAB did.

[5] Audience Network SDK. `https://developers.facebook.com/docs/audience-network`

Attack	Flaw	Attacker Goal	Attacker Models	App Conditions	Android Versions
A1: Stripping CT	**F1**	**G1**	**M1**, **M2**, **M3**	None	All
A2: Library Injection	**F2**	**G1**	**M1**, **M2**	None	8.0+
A3: Debuggable Flag	**F2**	**G1**	**M1**	None	All
A4: Resources / Assets	**F2**	**G1**	**M1**, **M2**, **M3**	Relevant files	All
A5: Backup Opt-In	**F2**	**G2**	**M1**	None	4.0 - 13[a]

[a]The vulnerability was fixed in Android 14 after our responsible disclosure

Table 7.1: Summary of the attacks we identified against CT.

This issue was solved in October 2022 (*bundletool* 1.12.0) by adding the hash of the placeholder APK's DEX file to every CT generated. The DEX file can, therefore, be later injected into generated APKs without tripping CT checks. Thus, with every CT generation, developers unknowingly agree a third-party DEX file to be executed in the context of their application. Even worse, Google does not disclose the source code of that DEX file. We conclude that app archiving effectively bypasses CT's integrity guarantees intended to protect against APK manipulations by app distributors.

6. Attacks Against CT

The attacks presented in this section take advantage of the CT flaws we discussed in previous Section 5. We experimentally confirmed all attacks on a Google Pixel 6a running Android 13.0. For simulating an **M1** attacker, we built Magisk[6] modules that installed our proofs of concept as privileged applications. An overview of the attacks, their exploited flaws, respective attacker model and goals may be found in Table 7.1.

A1: Code Execution by Stripping CT

Given the optionality of the feature, an attacker under any of our models may simply remove the CT of an AAB when generating a manipulated APK file for a victim user. Since no trace of CT is left in the generated APK file, the victim may only detect the manipulation if they have a separate secure channel to the developer. To summarize, a stripping attack works by:

[6]Magisk: The Magic Mask for Android. `https://github.com/topjohnwu/Magisk`

1. Removing the CT JWT from the AAB submitted by the developer.
2. Replacing or adding DEX or SO files in the AAB.
3. Generating and serving APKs from the manipulated AAB.

A2: Code Execution through Library Injection

CT by design covers only executable parts of an application. It does not protect the integrity of the application manifest, which means that any changes an attacker applies to this file go unnoticed. This enables an **M1** or **M2** attacker to inject code into any app's classpath and enforce this payload's execution during app launch. More specifically, injecting code may be accomplished through the undocumented `uses-static-library` manifest entry. During app launch, the system parses this entry and loads all DEX files of the referenced packages into the app's classpath. There are no restrictions on referenced packages other than that they must be installed on the device and that their signing certificate must be known to the dependent application. The application manifest also offers multiple options for an attacker to enforce the invocation of the injected code. The most straightforward approach is adding a `ContentProvider` element that references a class in the injected code. When launching an application, the system automatically invokes the `onCreate()` method of any `ContentProvider`s it exposes. From there, the attacker may then, e.g., inspect files stored in the app's private data folder or manipulate the internal data structures of the ART runtime to alter further app functionality. To exploit this vulnerability, an attacker may follow these steps:

1. Under attacker model **M1**, the attacker uses the privileged `android.permission.INSTALL_PACKAGES` permission to install an additional package containing the malicious code. Under attacker model **M2**, the attacker includes the malicious code in the app store app already installed on the device.
2. The attacker injects the `uses-static-library` entry referencing the malicious code package and a `ContentProvider` into the manifest of the target application. Alternatively, the referenced library may simply contain a class named identical to one of the target application's app components.

3. When launched on the victim device, the attacker's malicious code is executed in the context of the target application.

A3: Code Execution through Debuggable Flag

Modifying its manifest also allows an attacker to mark an application as debuggable without influencing its CT. The `debuggable` manifest flag is usually only set on applications during development. It weakens some of the app isolation security features in the OS to facilitate inspecting the app's runtime behavior. Most notably, the ART runtime opens a Java Debug Wire Protocol (JDWP) debug port for applications carrying this flag. An app developer may then connect the Java Debugger (JDB) to the application through the Android Debug Bridge (ADB). The ADB stack involves a server running on the Android device while the system-wide debug setting is active. ADB clients may connect to this server through USB or TCP/IP. In accordance with the Android Platform Security Model [28], initial connections need explicit consent by the user. For wireless connections, this involves supplying the client with a numeric code generated and displayed by the ADB server on the device. Subsequent connections utilize exchanged asymmetric keys for authentication.

Although ADB is intended for connecting to the Android device from an external computer, connections from a local process are possible as well. We also found that a privileged application carrying the `android.permission.MANAGE_DEBUGGING` permission may bypass the user consent requirement described above. These possibilities allow an attacker under model **M1** to gain code execution in any app while the device is connected to a Wifi network by carrying out the following procedure:

1. The attacker sets the debuggable flag in the manifest of the target application delivered to a victim user.

2. The attacker uses the privileged `android.permission.MANAGE_DEBUGGING` permission to enable wireless debugging for the currently connected Wifi SSID. This works by invoking the `allowWirelessDebugging()` and `enabledPairingByPairingCode()` methods on the `AdbManager` system service.

3. The attacker may listen for the `WIRELESS_DEBUG_PAIRING_RESULT_ACTION` broadcast to learn the server port and pairing code that the user normally needs to manually enter into the ADB client.

4. The attacker establishes an ADB connection to localhost, authenticating using the intercepted pairing code.

5. The attacker can now manipulate the target application's execution by establishing a JDB connection via ADB.

A4: Code Execution through Resources or Assets

While CT intends to cover executable parts of an application, it is entirely ignorant of the fact that many applications contain executable code in non-standard forms or locations. CT only covers SO and DEX files in their default locations. However, applications may load DEX files or native code from any location within their APK. Additionally, developers commonly take advantage of cross-platform app frameworks such as *React Native*, *Xamarin*, *Cordova*, or *Ionic*. Many of these frameworks ship the app's functionality in code formats other than DEX or SO. An attacker under any of our models may trivially compromise an affected application by manipulating these custom code formats.

A5: Data Access through Backup Opt-in

System backups on Android usually need to be explicitly started by the user, either through the UI or ADB. Once manually initiated, the system takes a copy of the private data directory of all applications that do not opt out through the `allowBackup` manifest entry.

We identified a vulnerability in the Android OS that allowed privileged applications to bypass user confirmation for backups. Confirmation is implemented by having the `BackupManager` system service generate a random token that is passed to system UI. The backup process is started only if this token is reported back to the `BackupManager` service. Unfortunately, while the token was transported to system UI through an adequately secured channel, it was also leaked to the system's Logcat log. Since privileged applications may obtain the `android.permission.READ_LOGS` permission to gain access to these log messages, they were able to bypass the user's consent for backups. We responsibly disclosed this vulnerability to Google. A fix was implemented and is being deployed to end-user devices in the Android 14 release. For unpatched devices, an **M1** attacker may:

1. Remove the `allowBackup` flag from the manifest of the target application delivered to a victim user.
2. Use the privileged `android.permission.BACKUP` permission for invoking `adbBackup()` on the `BackupManager` system service.
3. Use the privileged `android.permission.READ_LOGS` permission to extract the leaked user confirmation token.
4. Spoof user confirmation to the `BackupManager` by passing the token to `acknowledgeFullBackup()`.
5. Extract the contents of the target application's private data directory from the backup.

7. AAB and CT In Practice

In this section, we evaluate the prevalence and security of CT in real-world applications. We concentrate our analyses on Google Play and Huawei AppGallery. Google Play is the official app store on the Android platform and, therefore, largest in terms of offered apps and active users. Huawei AppGallery was the first third-party Android app store to add support for AAB (in 2020) [18].

7.1. Prevalence of AAB and CT

For determining the prevalence of real-world applications using the AAB distribution format and CT, we carried out large-scale automated analyses on 3.3 million free applications from Google Play and 230k free apps from Huawei AppGallery. It is worth noting that neither platform publishes an official index of available apps. Not even the exact number of listed packages is publicly disclosed, so we are unable to provide any information regarding the completeness of our analysis.

The Google Play dataset was obtained from the *AndroZoo* project [2] in April 2023 and filtered to only include apps listed on Google Play at the time of our analysis. We additionally retrieved metadata directly from Google Play servers for the resulting 3 265 096 apps to detect whether they had been submitted as an AAB file (AndroZoo did not provide app metadata at the time of our study). The dataset from Huawei AppGallery consists of 226 568 free apps indexed through brute-forcing application

identifiers on Huawei AppGallery in April 2024. Since no suitable information could be identified in AppGallery's app metadata, we interpreted the presence of the file `META-INF/BNDLTOOL.RSA` in served APKs as an indicator for original submissions as AAB. Unmodified *bundletool* versions store the APK signature at this location. However, since Huawei might have used a modified *bundletool* binary for some of the apps on AppGallery, we note that our results for Huawei AppGallery only represent a lower bound. The presence of CT in an app was assessed by scanning the ZIP central directory in the APK for an entry named `META-INF/code_transparency_signed.jwt`. Since this is the only allowed location for the CT file as per the scheme's specification, this analysis is precise. Carrying out the analyses took about 80 hours for Google Play and 5 hours for Huawei AppGallery.

7.1.1. Android Application Bundle Prevalence

Of the 3 265 096 applications analyzed from Google Play, 1 528 310 (46.8 %) had been submitted in the AAB format. The remaining apps must have been enlisted on Google Play before 2021 (even if they may since have been updated; AAB is only mandatory for *new apps* in general, not for updates). These numbers indicate an increase in AAB adoption compared to official numbers published by Google in 2021, when *"more than 1 million apps"* [9] were reported to have been submitted as AAB. Of the 226 568 analyzed apps from Huawei AppGallery, only 97 (0.04 %) had been submitted in the AAB format. We trace this low number back to the fact that the format is entirely optional for this app store. We note that all the apps that use AAB on any app store are susceptible to the supply chain attacks discussed in Section 6.

7.1.2. CT Prevalence

Only 21 (0.0014 %) of the applications submitted in the AAB format to Google Play contained a CT JWT. A full list of them may be found in Appendix 11.1. Of these 21 applications, 11 are plugins for a geospatial planning app, all listed under the same publisher account on Google Play. 4 more applications were published by a single company. Additionally, three apps are stock portfolio management apps that are so similar in UI and functionality that they likely originate from the same developer, even if listed under different publishers on Google Play (all are French banks;

two belong to the same enterprise group). We conclude that within the analyzed dataset, only four developer teams were aware of AAB's security implications and the (limited) countermeasure that exists in CT. None of the analyzed apps from Huawei AppGallery contained a CT JWT. We assume that CT's optionality (design flaw **F1**) and implementation flaw **F5** are major contributing factors to the low adoption of CT on both app stores.

It is worth noting that due to capacity constraints, we chose not to contact the developers of applications that lacked CT JWTs. This would have required us to establish secure channels (e.g., by finding contact information on developer websites; the data published in app stores may not be trusted) to the developers of almost all 3.5 million apps in our dataset. It would have allowed us to learn whether their submitted AABs had contained a JWT in the first place. We, therefore, potentially missed stripping attacks (See Section 6) in our analysis.

7.1.3. Verifying CT

We used *bundletool* to verify the integrity of the 21 samples in our dataset that contained a CT JWT. The tool confirmed that all JWTs contained matching hash entries for the DEX and SO files in the corresponding APKs. We also tried to ascertain that all CT JWTs were signed using the legitimate public key of the respective original developers. For the 11 plugin apps, the legitimate public key could be found in the open-source variant of the host app, which allowed us to completely verify these apps' CT with reasonable certainty. For the 10 remaining apps, we were unable to find trustworthy public information regarding their legitimate public key. We, therefore, contacted the support email addresses recorded in their respective Google Play listings. We could confirm the authenticity for all but one of the provided email addresses through the respective company websites. However, we did not receive any response to our emails within three months. Appendix 11.1 shows detailed results for this analysis.

7.1.4. Susceptibility to CT Attacks

All of the apps we found to use CT are susceptible to attacks **A1**, **A2**, **A3**, and **A5**. This simply follows from the fact that these attacks do not impose any requirements on affected apps.

	A4	
Package Name	JS	DLL
com.atakmap.android.bng.plugin		
com.atakmap.android.compassnav.plugin		
com.atakmap.android.datasync.plugin		
com.atakmap.android.firesurvey.plugin		
com.atakmap.android.geocam.plugin		
com.atakmap.android.grgbuilder.plugin		
com.atakmap.android.takchat.plugin		
com.atakmap.android.uastool.plugin		
com.atakmap.android.vns.plugin		
com.atakmap.android.wave.plugin		
com.microsoft.loop	⚠	⚠
com.neuflize.obc.bourse	⚠	
com.oracle.ebs.maintenance	⚠	
com.oracle.ebs.scm.mwa.MSCA10	⚠	
com.oracle.ebsapps.csm	⚠	
com.oracle.events	⚠	
com.portzamparc.bourse	⚠	
com.somewearlabs.swtak.plugin		
net.bnpparibas.bourse	⚠	

Table 7.2: Susceptibility to attack **A4** of apps that use CT. *JS*/*DLL*: Apps integrate JavaScript code or Dynamic Link Library files.

To also evaluate the susceptibility for app-dependent attack **A4** (Code Execution through Resources or Assets), we built a simple static analysis tool. This tool scans for files in an APK's assets or resources that carry the file signature or extension of the ELF, DEX, APK, or DLL formats. Additionally, all files in these folders that contain only printable characters are ran through the Esprima JavaScript syntax validator.

Our analysis indicates that 8 of the 21 (38 %) applications that use CT are susceptible to **A4**. 8 apps implement some of their functionality in JavaScript, while 1 includes code in a Dynamic Link Library (DLL). None of the apps contained DEX, SO, or APK files in their resources or assets. The exact findings for each application are listed in Table 7.2.

7.1.5. Eligibility for CT and Possible Attacks Among Popular Apps

We also ran our static analysis tool described in Section 7.1.4 against a representative set of the most popular applications from Google Play. Our goal was to determine their eligibility for adopting CT and their

susceptibility to our attacks once they do so. The dataset was assembled by collecting the package names of the 200 most popular free applications of all 36 categories on Google Play in December 2023. We again extracted the APKs for these applications from the *AndroZoo* [2] project. Since some apps were not in their collection, our final dataset consisted of 6 648 applications.

Our tool indicates that 3 935 (59.2 %) of the analyzed apps had been submitted to Google Play in the AAB format. One (0.02 %) app (*Microsoft Loop*, which we also found in Section 7.1.2) uses CT. 1 442 (21.7 %) of packages in our dataset are not compatible with *bundletool*'s CT implementation due to containing DEX or SO files in non-standard locations (**F5**). In 1 243 (18.7 %) of the analyzed apps, the integration of the Facebook Audience SDK at least contributed to this incompatibility. If they adopted CT, 3 467 of the analysed apps (52.2 %) would be susceptible to attack **A4**. All of them would be susceptible to attacks **A1**, **A2**, **A3**, and **A5**.

7.2. Case Study

To demonstrate the practicality of the attacks described in Section 6, we provide a case study on attacking CT in a real-world application.

7.2.1. Microsoft Loop

The Android application (com.microsoft.loop) for this collaborative creation environment has been downloaded from Google Play by more than 100 000 users as of December 2023. The app employs CT, likely in an attempt to thwart the potential for supply chain attacks enabled by the AAB format. However, this attempt proves ineffective against manipulations of the app's runtime behavior. For this case study, we demonstrate how attack **A2** may be used to inject code for stealing login data without invalidating the APK's CT. We implemented a custom stage for the *A2P2* APK patching pipeline [7] that injects a `uses-static-library` element into the application manifest. We also created an additional package exposing the referenced library to the system, which needs to be installed on the victim device (see Section 6). The library code defines a class named identical to the *Microsoft Loop* application's main UI activity. Since static libraries take precedence over app code in Android's class loading, this is

```
$ bundletool check-transparency --mode=apk
↪  --apk-zip=loop-patched-apks.zip
   APK signature is valid. SHA-256 fingerprint of the apk signing key
    ↪  certificate (must be compared with the developer's public key
    ↪  manually): 94 29 4F 23 A8 ... 50 6B CD B0 22 AB 1F D8 C9 3C
   Code transparency signature is valid. SHA-256 fingerprint of the
    ↪  code transparency key certificate (must be compared with the
    ↪  developer's public key manually): 52 D4 B1 8E CA 3F 78 62 FF
    ↪  ... 89 54 40 B7 C2 02
   Code transparency verified: code related file contents match the
    ↪  code transparency file.
```

Listing 7.1: The manipulated APK still successfully passes the CT verification in *bundletool*

enough to intercept the app's launch. Our malicious main activity displays a fake login screen that, in a real-world attack, could, for example, forward all collected credentials to a web server controlled by the attacker. Despite the ability for the attacker to execute arbitrary code in the context of the *Microsoft Loop* application, its CT remains valid, as can be confirmed through the *bundletool* output in Listing 7.1.

Please note that we do not have access to the private key for the APK's app signing certificate, so we cannot entirely accurately simulate a supply chain attacker. In a real-world attack, the manipulated APK would be signed with the developer's correct app signing certificate. As explained in Section 3, the AAB scheme requires the developer to share the app signing key with the distributor.

8. Discussion & Future Work

8.1. Improving AAB and CT

Given the number of serious design flaws in CT (see Section 5), we consider a complete secure design out of scope for this work. However, we would like to suggest improvements that mitigate the design problems raised in Section 5. We discuss fixes for the implementation flaws in *bundletool* in Appendix 11.2.

F1: Optionality. The simplest solution for this issue is making CT a mandatory part of the AAB format. This would enable trivial detection of stripping attacks. If for some reason (e.g. backward-compatibility), such a change cannot be realized, we note that the information obtainable to the user through the separate secure channel to the developer (as discussed in the CT documentation [16]) at least needs to include an indication whether the AAB contained a CT when submitted.

F2: Scope. The attacks described in Section 6 highlight the necessity for CT to cover the application manifest, assets, and resources. However, it is unclear how such a solution may be implemented, given that *bundletool* itself modifies these parts of the application during APK generation.

F3: Communication Channel. We argue the need for a standardized registry for legitimate public CT keys. A solution might be akin to Certificate Transparency for TLS certificates [22].

Practicality. Currently, verifying an application's CT requires a deep understanding of the Android OS and development tools. It is, therefore, only accessible to advanced users. Future work needs to identify possibilities for improving the user experience of CT verification.

8.2. Preventing infrastructure-level attacks

Given the complexity of app stores, it is unlikely that the possibility for attacks compromising app store infrastructure (attacker models **M1** and **M2**) can ever be entirely eliminated. For vendors whose devices go through Google's GMS certification, further security could be imposed by requiring security audits of privileged app stores as part of the certification process. However, since Android consciously allows third-party app stores with no ties to Google nor device vendors, attacker model **M2** will always remain well within the boundaries of possibility.

8.3. Server-Side APK Generation

The fundamental proposition of the AAB publishing format is the possibility for distributors such as app stores to generate APK files optimized for end-users' devices. While optimized APKs undoubtedly improve the user experience, we argue that the AAB format is not necessary for serving optimized APKs to end users.

First and foremost, generating APKs on the server would be necessary if the number of possible device configurations to optimize for was unknown at compile time. However, these optimized builds are simply composed of all possible combinations of resource variants already provided by the app developer. Therefore, the optimized APKs could already be packaged during app build by the developer. In fact, functionality for generating a container of all possible optimized APKs is already available in *bundletool*.

Server-side APK generation only possibly offers advantages when new features are to be globally retrofitted into already submitted apps. However, the crude workaround implemented for App Archives (see **F6**) displays that any such efforts drastically interfere with the developer's interest in provable integrity guarantees for their app's behavior.

Given the lack of apparent advantages of server-side APK generation and its negative security repercussions, we argue that app stores should at least offer the option for developers to submit a container of locally built optimized APKs for their apps. This would allow them to sign the APKs themselves without having to hand over the APK signing key. As a result, they could benefit from the strong integrity guarantees of the APK format and app attestation.

Security Impact. Due to the small number of apps that use CT, the practical security impact of the identified flaws in CT alone is relatively small. However, the security impact of our work becomes apparent when considering the bigger picture: Given the security consequences of AAB (without CT), the low prevalence of CT means that a very large number of apps are vulnerable to supply chain attacks. Even worse, due to the current state of CT as discussed in this paper, no effective mitigations for the severe security repercussions of the AAB distribution scheme are available.

9. Related Work

Despite their impact on the security architecture of the Android ecosystem, to the best of our knowledge, our work represents the first scientific publication concerned with Code Transparency and the Android Application Bundle distribution scheme. However, research has been ongoing in the related domains of Android Supply Chain Attacks, App Integrity

Checking, and App Misconfiguration. This section provides an overview of these fields.

Android Supply Chain Attacks. To the best of our knowledge, no other studies exist on the security of the global application supply chain *from the developer to the end user* on Android. However, several recent publications deal with various aspects of the software supply chain upstream of the developer, i.e., third-party app components and tools. The work that is most similar to ours in this category is the one by Wang et al. [38], which discusses the possibilities for malicious third-party libraries to stealthily override an application's security policies. A number of further studies elaborate on the perils of malicious app components in a benign app, such as Wang et al. [37], Kim et al. [21] and Zhang et al. [44]. Several research teams, including Wu et al. [40], Zhan et al. [43], and Backes et al. [4] investigated the prevalence of vulnerable third-party libraries integrated into applications.

App Integrity Checking. Since the introduction of the Android OS, repackaging attacks have been a major focus of research attention. These attacks exploit the fact that APK signing does not provide any authenticity guarantees, allowing anyone to redistribute a manipulated version of a legitimate application. Research in this field has mostly focused on either detecting repackaging attacks or repackage-proofing applications. Works on repackage detection propose approaches based on watermarking [45] or similarities in app resources [35] or code [12]. These approaches scale poorly since they need to compare each sample to a large database of applications. Shi et al. [36] present a solution that does not work through comparison, but only works for repackaging attacks that use virtualization techniques. Suggestions for repackage-proofing include Stochastic Stealthy Networks (SSN) [25] that spread obfuscated signature checks throughout app code, logic bombs [42] that only trigger when executed on end-user devices or a combination of native and self-decrypting code [32]. However, as recently shown by Ma et al. [26], none of these countermeasures are insurmountable by reasonably determined attackers due to being confined to the same sandbox as the malicious code. An effective mitigation for repackaging attacks in server-client scenarios can be found in hardware-assisted app attestation, as discussed by Prünster et al. [31]. Several publications recently investigated the security of hardware attestation implementations. As part of these efforts, Aldoseri et al. [1] and Shakevsky et al. [34] identified severe flaws affecting millions of devices. Complementarily, Ibrahim et al. [19] and Berlato et al. [5] studied the integration of these technologies

into real-world applications and discovered they were used by only 0.04 % of analyzed applications. All these protection methods are ineffective against the attacks discussed in our paper because they rely on the app developer having exclusive control over the APK signing key. However, AAB distribution requires the developer to share the APK signing key with the app distributor.

App Misconfigurations. Many of our attacks against CT exploit the powerful role of configuration files in Android apps. Several recent works investigated how mistakes in these configurations can introduce exploitable vulnerabilities. Yang et al. [41] and Jha et al. [20] constructed tools for automatically detecting manifest misconfigurations in compiled apps and found severe security flaws in widely used software. Oltrogge et al. [30] examined the Network Security Configuration in 1.3 million apps and uncovered that it was used for bypassing the default security policy in the overwhelming majority of cases. Lastly, Lenk et al. [23] analyzed the security implications of real-world `ContentProvider` configurations, reporting sensitive data leakage in a considerable number of apps.

10. Conclusion

In this paper, we presented the first comprehensive security analysis of Code Transparency (CT) for Android Application Bundles. We identified 3 design flaws and 3 implementation flaws in CT and its *bundletool* reference implementation. These flaws may be exploited through 7 different attacks for bypassing CT to gain code execution or data access in applications.

We further provided detailed statistics regarding CT in practice. We found that CT is almost non-existent in real-world applications despite the severe security consequences of the AAB format. We were also able to show that more than 20 % of the most popular applications on Google Play are not even able to use CT due to implementation flaws in *bundletool*. For apps that use CT, we provided a case study illustrating the consequences of the attacks we identified. Finally, we discussed possibilities for improvements to CT and AAB.

References

[1] Abdulla Aldoseri, Tom Chothia, José Moreira, and David F. Oswald. Symbolic modelling of remote attestation protocols for device and app integrity on Android. In: Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS. 2023 (p. 195).

[2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: collecting millions of Android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016. 2016 (pp. 187, 191).

[3] AppBrain. AppBrain Statistics: Facebook Audience Network. 2024. URL: https://www.appbrain.com/stats/libraries/details/facebook_ads/facebook-audience-network (p. 182).

[4] Michael Backes, Sven Bugiel, and Erik Derr. Reliable Third-Party Library Detection in Android and its Security Applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security CCS. 2016 (pp. 182, 195).

[5] Stefano Berlato and Mariano Ceccato. A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps. In: J. Inf. Secur. Appl. (2020) (p. 195).

[6] Karen Ng Chris Sells Benjamin Poiesz. Use Android Jetpack to Accelerate Your App Development. online. May 2018. URL: https://android-developers.googleblog.com/2018/05/use-android-jetpack-to-accelerate-your.html (p. 168).

[7] Florian Draschbacher. A2P2 - An Android Application Patching Pipeline Based On Generic Changesets. In: ARES. 2023 (p. 191).

[8] Florian Draschbacher and Lukas Maar. Manifest Problems: Analyzing Code Transparency for Android Application Bundles. In: ACSAC. 2024 (p. 165).

[9] Dom Elliott. The future of Android App Bundles is here. online. June 2021. URL: https://android-developers.googleblog.com/2021/06/the-future-of-android-app-bundles-is.html (p. 188).

[10] Dom Elliott and Yafit Becher. Recent Android App Bundle improvements and timeline for new apps on Google Play. online. Aug. 2020. URL: https://android-developers.googleblog.com/2020/08/recent-android-app-bundle-improvements.html (pp. 168, 169).

[11] Anwar Ghuloum. Fresher OS with Projects Treble and Mainline. online. May 2019. URL: https://android-developers.googleblog.com/2019/05/fresher-os-with-projects-treble-and-mainline.html (p. 168).

[12] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. CodeMatch: obfuscation won't conceal your repackaged app. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering ESEC/FSE. 2017 (p. 195).

[13] Google. Android Developers: About Android App Bundles. online. Accessed 2023-12-05. Nov. 2023. URL: https://developer.android.com/guide/app-bundle (p. 168).

[14] Google. Android Developers: Android App Bundle frequently asked questions. online. Accessed 2023-12-05. Nov. 2023. URL: https://developer.android.com/guide/app-bundle/faq (p. 177).

[15] Google. Android Developers: bundletool. online. Accessed 2023-12-05. July 2023. URL: https://developer.android.com/tools/bundletool (p. 175).

[16] Google. Android Developers: Code transparency for app bundles. online. Accessed 2023-12-05. July 2021. URL: https://developer.android.com/guide/app-bundle/code-transparency (pp. 176, 177, 181, 193).

[17] Google. Google Developers: Binary Transparency. online. Accessed 2023-12-05. Aug. 2023. URL: https://developers.google.com/android/binary_transparency/overview#threat_model (pp. 177, 178).

[18] Huawei. Huawei Developers: App Bundles Distribution. online. Accessed 2024-04-19. Dec. 2020. URL: https://web.archive.org/web/20201205143521/https://developer.huawei.com/consumer/en/agconnect/app-bundle/ (p. 187).

[19] Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. SafetyNOT: on the usage of the SafetyNet attestation API in Android. In: MobiSys '21: The 19th Annual International Conference on Mobile Systems, Applications, and Services. 2021 (p. 195).

[20] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. Developer mistakes in writing Android manifests: an empirical study of configuration errors. In: Proceedings of the 14th International Conference on Mining Software Repositories MSR. 2017 (p. 196).

[21] Joongyum Kim, Junghwan Park, and Sooel Son. The Abuser Inside Apps: Finding the Culprit Committing Mobile Ad Fraud. In: 28th Annual Network and Distributed System Security Symposium NDSS. 2021 (p. 195).

[22] Ben Laurie, Eran Messeri, and Rob Stradling. Certificate Transparency Version 2.0. In: RFC 9162 (2021) (p. 193).

[23] Christopher Lenk and Johannes Kinder. Poster: Privacy Risks from Misconfigured Android Content Providers. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023. ACM, 2023 (p. 196).

[24] Tian Lim. I/O 2018: Everything new in the Google Play Console. online. May 2018. URL: https://android-developers.googleblog.com/2018/05/io-2018-everything-new-in-google-play.html (p. 168).

[25] Lannan Luo, Yu Fu, Dinghao Wu, Sencun Zhu, and Peng Liu. Repackage-Proofing Android Apps. In: 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 2016 (p. 195).

[26] Haoyu Ma, Shijia Li, Debin Gao, Daoyuan Wu, Qiaowen Jia, and Chunfu Jia. Active Warden Attack: On the (In)Effectiveness of Android App Repackage-Proofing. In: IEEE Trans. Dependable Secur. Comput. 19.5 (2022) (p. 195).

[27] Iliyan Malchev. Here comes Treble: A modular base for Android. online. May 2017. URL: https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html (p. 168).

[28] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The Android Platform Security Model. In: ACM Trans. Priv. Secur. 24.3 (2021) (pp. 168, 171, 178, 180, 185).

[29] Mark Murphy. Uncomfortable Questions About App Signing. online. Sept. 2020. URL: https://commonsware.com/blog/2020/09/23/uncomfortable-questions-app-signing.html (p. 169).

[30] Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications. In: 30th USENIX Security Symposium. 2021 (p. 196).

[31] Bernd Prünster, Gerald Palfinger, and Christian Kollmann. Fides: Unleashing the Full Potential of Remote Attestation. In: Proceedings of the 16th International Joint Conference on e-Business and Telecommunications ICETE. 2019 (pp. 174, 195).

[32] Chuangang Ren, Kai Chen, and Peng Liu. Droidmarking: resilient software watermarking for impeding android application repackaging. In: ACM/IEEE International Conference on Automated Software Engineering ASE. 2014 (p. 195).

[33] Steven B. Roosa and Stephen Schultze. Trust Darknet: Control and Compromise in the Internet's Certificate Authority Model. In: IEEE Internet Comput. 17.3 (2013) (p. 178).

[34] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Keymaster Design. In: 31st USENIX Security Symposium, USENIX Security 2022. 2022 (p. 195).

[35] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a scalable resource-driven approach for detecting repackaged Android applications. In: Proceedings of the 30th Annual Computer Security Applications Conference ACSAC. 2014 (p. 195).

[36] Luman Shi, Jiang Ming, Jianming Fu, Guojun Peng, Dongpeng Xu, Kun Gao, and Xuanchen Pan. VAHunt: Warding Off New Repackaged Android Malware in App-Virtualization's Clothing. In: 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS). 2020 (p. 195).

[37] Jice Wang, Yue Xiao, Xueqiang Wang, Yuhong Nan, Luyi Xing, Xiaojing Liao, Jinwei Dong, Nicolás Serrano, Haoran Lu, XiaoFeng Wang, and Yuqing Zhang. Understanding Malicious Cross-library Data Harvesting on Android. In: 30th USENIX Security Symposium, USENIX Security. 2021 (p. 195).

[38] Xueqiang Wang, Yifan Zhang, XiaoFeng Wang, Yan Jia, and Luyi Xing. Union under Duress: Understanding Hazards of Duplicate Resource Mismediation in Android Software Supply Chain. In: 32nd USENIX Security Symposium, USENIX Security. 2023 (p. 195).

[39] Josh Wentz. App Bundles for Google TV and Android TV. online. Nov. 2022. URL: https://android-developers.googleblog.com/2022/11/app-bundles-for-google-tv-and-android-tv.html (p. 168).

[40] Yafei Wu, Cong Sun, Dongrui Zeng, Gang Tan, Siqi Ma, and Peicheng Wang. LibScan: Towards More Precise Third-Party Library Identification for Android Applications. In: 32nd USENIX Security Symposium, USENIX Security. 2023 (p. 195).

[41] Yuqing Yang, Mohamed Elsabagh, Chaoshun Zuo, Ryan Johnson, Angelos Stavrou, and Zhiqiang Lin. Detecting and Measuring Misconfigured Manifests in Android Apps. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security CCS. 2022 (p. 196).

[42] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, Zhoujun Li, Chin-Tser Huang, and Csilla Farkas. Resilient User-Side Android Application Repackaging and Tampering Detection Using Cryptographically Obfuscated Logic Bombs. In: IEEE Trans. Dependable Secur. Comput. 18.6 (2021) (p. 195).

[43] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. ATVHUNTER: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications. In: 43rd IEEE/ACM International Conference on Software Engineering ICSE. 2021 (p. 195).

[44] Zicheng Zhang, Wenrui Diao, Chengyu Hu, Shanqing Guo, Chaoshun Zuo, and Li Li. An empirical study of potentially malicious third-party libraries in Android apps. In: WiSec '20: 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks. 2020 (p. 195).

[45] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. AppInk: watermarking android apps for repackaging deterrence. In: 8th ACM Symposium on Information, Computer and Communications Security ASIA CCS. 2013 (p. 195).

11. Appendix

11.1. Details of apps using CT for AAB

Table 7.3 lists all apps in our dataset that use CT and their respective verification results.

11.2. Fixes to implementation flaws

In this section, we discuss how the implementation flaws in *bundletool* that we identified in Section 5.2.

F4: Certificate Reuse. Checks need to be added to APK generation in *bundletool* to ensure the CT key may not be used as the app signing key. Additionally, the APK verification result should include information on whether the two keys are distinct.

F5: DEX or SO in Assets. This issue may be mitigated by aligning the CT verification policy with its creation policy. The most sensible approach would be including all DEX or SO files in the CT, irrespective of their location in the AAB/APK.

F6: App Archiving. It is unclear whether the CT scheme should allow such radical modifications of an app as carried out through this feature. We believe that all source code of the placeholder APK should at least be available for public inspection.

Package Name	Version	Publisher	Valid CT	Original Key	Unique Key?
com.atakmap.android.bng.plugin	2.0.0-SNAPSHOT [4.8.1]	TAK Product Center	✓	✓	✓
com.atakmap.android.compassnav.plugin	1.0 (8a9728b2) - [4.8.1]	TAK Product Center	✓	✓	✓
com.atakmap.android.datasync.plugin	1.6.6 (1b388a2d) - [4.8.1]	TAK Product Center	✓	✓	✓
com.atakmap.android.firesurvey.plugin	2.9.01.2 (95e2d92a) - [4.8.1]	TAK Product Center	✓	✓	✓
com.atakmap.android.geocam.plugin	1.0 (0ffd4a57) - [4.8.1]	TAK Product Center	✓	✓	✓
com.atakmap.android.grgbuilder.plugin	1.1 (3a1fc26f) - [4.8.1]	TAK Product Center	✓	✓	✓
com.atakmap.android.takchat.plugin	1.0 (a43b4fbb) - [4.8.1]	TAK Product Center	✓	✓	✓
com.atakmap.android.uastool.plugin	12.2 (f2ccbf8d) - [4.8.1]	TAK Product Center	✓	✓	✓
com.atakmap.android.vns.plugin	3.6 (9d26d997) - [4.8.1]	TAK Product Center	✓	✓	✓
com.atakmap.android.wave.plugin	2.0 (4bca9c49) - [4.8.1]	TAK Product Center	✓	✓	✓
com.microsoft.loop	1.0.0321.00	Microsoft Corporation	✓	?	✓
com.neuflize.obc.bourse	5.6.8	Banque Neuflize OBC[a]	✓	?	✓
com.oracle.ebs.maintenance	10.0.0	Oracle America, Inc	✓	?	✓
com.oracle.ebs.scm.mwa.MSCA10	10.0.0	Oracle America, Inc	✓	?	✓
com.oracle.ebsapps.csm	10.0.0	Oracle America, Inc	✓	?	✓
com.oracle.events	1.0	Oracle America, Inc	✓	?	✓
com.portzamparc.bourse	5.6.8	Portzamparc SA[a]	✓	?	✓
com.somewearlabs.swtak.plugin	0.11.9 - [4.8.1]	TAK Product Center	✓	✓	✓
net.bnpparibas.bourse	5.6.11	BNP PARIBAS[a]	✓	?	✓

[a]These apps likely are maintained by the same developer

Table 7.3: The apps in our dataset that use CT for AAB and their respective verification results. All apps contained a valid Code Transparency (CT). In column *Original Key*, **?** signifies that the original developer's public CT key could not be obtained. It therefore was not possible to check whether it matched the certificate found in the CT JWT. *Unique Key* denotes that the app signing key was not reused as the CT key.

8

CryptoShield: Automatic On-Device Mitigation for Crypto API Misuse in Android Apps

Publication Data

Florian Draschbacher and Johannes Feichtner. CryptoShield: Automatic On-Device Mitigation for Crypto API Misuse in Android Applications. In: AsiaCCS. 2023

Contributions

The author of this thesis jointly develped the idea for this paper with Johannes Feichtner, and holds the sole authorship for its implementation and textual descriptions.

It is worth noting that this paper shares similarities in the high-level approach with the author's master thesis titled CryptoPatcher. However, there are key differences in concept, implementation and evaluation. First, CryptoPatcher is a pure mitigator for a small number of misuse cases, while CryptoShield covers a much more comprehensive systematically derived range of misuse classes not only for mitigation, but also for detection comparable in accuracy with state-of-the-art pure detectors. Second, on a technical level, CryptoShield uses a dynamic hooking approach compatible with method invocations through reflection or JNI calls, which remain unmitigated with CryptoPatcher's static rewriting approach. Finally, only CryptoShield's efficiency and efficacy were confirmed through large-scale automated and manual analyses, as well as synthetic detection benchmarks.

CryptoShield: Automatic On-Device Mitigation for Crypto API Misuse in Android Applications

Florian Draschbacher Johannes Feichtner

Graz University of Technology

Abstract

Misuse of cryptographic APIs remains one of the most common flaws in Android applications. The complexity of cryptographic APIs frequently overwhelms developers. This can lead to mistakes that leak sensitive user data to trivial attacks. Despite herculean efforts by platform provider Google, countermeasures introduced so far were not successful in preventing these flaws. Users remain at risk until an effective systemic mitigation has been found.

In this paper, we propose a practical solution that mitigates crypto API misuse in compiled Android applications. It enables users to protect themselves against misuse exploitation until the research community has identified an effective long-term solution. CryptoShield consists of generic mitigation procedures for the most critical crypto API misuse scenarios and an implementation that autonomously extends protection onto all applications on an unrooted Android device. Our on-device CryptoShield Agent injects an instrumentation module into application packages, where it can intercept crypto API calls for detecting misuse and applying mitigations. Our solution was designed for real-world applicability. It retains the update flow through Google Play and can be integrated into existing MDM infrastructure.

As a demonstration of CryptoShield's efficiency and efficacy, we conduct automated (1604 apps) and manual (99 apps) analyses on the most popular applications from Google Play, as well as measurements on synthetic benchmarks. Our solution mitigates crypto API misuse in 96 % of all vulnerable apps, while retaining full functionality for 92 % of all apps. On-device instrumentation takes roughly 11 seconds per application package on average, with minimal impact on package size (5 %) and negligible runtime overhead (571 ms on average app launches, 101 ms worst-case mitigation overhead per crypto API call).

1. Introduction

Several publications over the last years have uncovered that crypto API misuse causes severe vulnerabilities in a large number of mobile applications. Android, the most popular mobile operating system, was shown to be particularly affected. Most recently, Oltrogge et al. [13] found that *37 %* of 15,000 statically analyzed applications contained *mistakes in their code for validating TLS certificates*. For a broader definition of crypto API misuse that considers framework-provided TLS and crypto primitives[1], Rahaman et al. [16] even reported a misuse prevalence of 91 %.

As a reaction to the first alarming reports a decade ago, platform provider Google started multiple initiatives to eliminate TLS misuse and insecure parametrization of cryptographic APIs in general. The Android platform saw the introduction of improved APIs, reworked documentation, detailed linter rules, and automated checks for software uploaded to the Google Play Store. Still, researchers keep showing that all efforts so far were largely ineffective.

Despite the consequences of widespread crypto API misuse on data security and the ineffectiveness of official countermeasures, very few publications have so far attempted to mitigate crypto API misuse in compiled applications. However, until an effective long-term solution has been identified and implemented by Google, third-party solutions are user's only option for protecting against the severe security repercussions of crypto API misuse.

Buhov et al. [3] proposed a dynamic approach in the form of a Cydia Substrate module. However, their exclusive focus on retroactively adding TLS pinning to applications did not acknowledge the full scope of the problem, which affects a far broader range of cryptographic APIs. Additionally, the presented module utilized the obsolete Cydia Substrate hooking framework and required a *rooted device*, thereby considerably limiting its real-world applicability. An important first step towards mitigating other classes of crypto API misuse was made by Ma et al. [11]. They suggested an approach based on static analysis and static rewriting of Dalvik Executable (DEX) files. However, the reliance on static analysis of the control flow graph limited the practicality of their solution (19 seconds patch generation *per misuse*; only the first misuse per method could be mitigated).

[1] We use the term *cryptographic APIs* to refer to Java APIs for TLS or crypto primitives.

We therefore propose CryptoShield as the first *practical* on-device method that mitigates both TLS and crypto primitive misuse on unmodified Android systems.

Our key contributions are:

- We devise *application-agnostic mitigation procedures* for 10 common classes of crypto API misuse, including flaws in the integration of TLS, Ciphers, Password-Based Encryption (PBE), Key Storage and Cryptographically Secure Random Number Generators (CSRNG). Our mitigations operate transparently to application code and require only runtime information. Thus, our method does *not* suffer from the imprecisions and resource-intensiveness of traditional patching approaches that employ static analysis of the control flow graph.

- We propose a self-contained *on-device component* named CryptoShield Agent (CSA) for deploying our mitigations to all applications on a target Android device. An instrumentation module intercepts crypto API calls to detect misuse and employ our mitigation procedures. Our implementation works on unrooted Android devices and does not require any interaction from the user. Application updating through Google Play remains fully functional. [2]

- We demonstrate the efficiency and efficacy of CryptoShield through automated and manual analyses on the 1604 and 99 most popular free applications, respectively, from the Google Play store. Additionally, we evaluate CryptoShield's detection precision on a synthetic benchmark, provide exact per-API runtime overheads and carry out a case study that illustrates how our solution can mitigate critical security vulnerabilities in two widely-used Android applications.

The remainder of this paper is organized as follows. Section 2 gives an overview of the Android OS and its cryptographic APIs. Section 3 outlines the addressed problem and the assumed attacker model. Section 4 describes our mitigation procedures, before Section 5 covers our prototype implementation, which is subsequently evaluated in Section 6. We discuss limitations of our system and plans for future work in Section 7, highlight related publications in Section 8, and conclude this paper in Section 9.

[2] Source code is available at `https://extgit.iaik.tugraz.at/fdraschbacher/cryptoshield`

2. Background

In this section, we provide a short overview of cryptographic APIs in the Android framework, introduce the Android package format, and outline the OS's application runtime.

2.1. Android Crypto APIs & Common Misuse

Android applications can take advantage of Java Cryptography Architecture (JCA) APIs for cryptographic functionality. The most common pitfalls with JCA APIs are that some default to insecure configurations, while others completely rely on developers' choice of secure parameters. Together with a lack of clarity in the official documentation, this has led to developers making serious mistakes in protecting their application's data. Since the exact ramifications of the particular misuse classes have already been thoroughly discussed in previous publications [5, 6, 11], we only provide a very condensed overview of the relevant points here.

For SSL/TLS (provided by the `SSLSocket` class), applications often implement `TrustManagers` that fail to properly verify the certificate chain presented by the server or `HostnameVerifiers` that don't confirm correspondence between a server's hostname and its presented TLS certificate. Both issues leave applications vulnerable to Man-In-The-Middle (MITM) attacks that can compromise all transmitted data. In Android 7.0, Google introduced the Network Security Configuration (NSC) system that allows developers to conveniently set up certificate pinning and trusted self-signed certificates. Although this possibility eliminated most needs for custom `TrustManagers` or `HostnameVerifiers`, many applications remained vulnerable in some form [13]. Additionally, a considerable portion of Android applications still use the unprotected HTTP protocol in their communication with servers [17].

Applications that utilize the Cipher API for data encryption and decryption were found to commonly use hardcoded or predictable initialization vectors, which breaks the indistinguishability against chosen plaintext attacks (IND-CPA) property of symmetric ciphers. Two ciphertexts can reveal similarities between their original plain texts. Electronic Code Book (ECB) mode for symmetric block ciphers exhibits the same vulnerability to chosen plaintext attacks. Since ECB is the JCA's default mode of operation, it can still be found in a considerable amount of applications.

Password-Based Encryption (PBE) in the JCA can be accessed through the SecretKeyFactory API. If it is supplied with hardcoded passwords or reused or predictable salt values, derived keys can either be compromised immediately or with considerably less effort than in bruteforce attacks against properly parameterized PBE.

Hardcoded passwords pose a problem to the KeyStore API for generating and storing cryptographic keys as well. They may lead to compromitation of all stored keys and thus any related encrypted data.

Lastly, Java offers a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG) through the SecureRandom API. Android's default implementation of the API has had a history of severe flaws caused by the reuse of seed material that led to the generation of a predictable sequence of numbers. Although these flaws have been mitigated at the framework level in recent versions of the Android OS, vulnerable legacy devices remain in use.

2.2. Android Application Package Format

Android applications are deployed in a special file format called Android Package (APK). On the outmost layer, it consists of a ZIP container signed by the application developer to ensure that only updates from the same author can replace an original installation. The content of the container follows a particular structure that is unique to the APK format. One or more Dalvik Executable (DEX) files store the class structure and program byte code of the application. The `AndroidManifest.xml` file encodes the contract between the OS and the application, declaring the supported functionality and required permissions or device capabilities. Native libraries are organized in dedicated folders within the APK archive. When obtained from Google Play, an app commonly is installed as a set of multiple Split APK files. The set comprises a base APK file storing DEX files, and several additional splits for device-specific resources or native code. As a key observation for the functionality of our solution, we point out that APK files reside in public storage on the Android device after their installation. PackageManager APIs can be used for locating all APK files for a given package. On Android versions up to 10, even completely unprivileged applications can access installed APK files, i.e. no permission is needed *at all*. Starting from Android 11, a suitable `<queries>`

element in the application manifest or the QUERY_ALL_PACKAGES permission is required.

2.3. Android Runtime

Android applications are typically written in Java or Kotlin and compiled into architecture-independent Dalvik bytecode. In current Android versions, the Android Runtime (ART) takes over the responsibility of executing this bytecode on the device. Applications may additionally implement parts of their functionality in native shared libraries that are dynamically linked into the runtime process. It is worth noting that shared libraries are in a position that allows them to manipulate the internal data structures of the ART runtime.

3. Problem Statement, Challenges, and Threat Model

In this section, we state the problem that needs to be addressed, derive a set of core goals, highlight the challenges involved in finding a solution and show how CryptoShield was designed to overcome them.

3.1. Problem Statement

We seek to provide a mitigation for crypto API misuse that enables users to protect themselves until the research community has identified an effective long-term solution for this widespread problem. Our goal is to cover as many classes of critical crypto misuse as feasible without breaking the functionality of target applications. We envision deployment by security-conscious individuals, as well as corporate organizations that wish to ensure data confidentiality on their complete fleet of Android devices no matter what applications employees utilize.

The core design goals of our solution are thus:

3.1.1. Practicality

We strive to propose a design that seamlessly integrates into the everyday workflows of real-world Android users.

3.1.2. User-Centricity

Our solution is intended to put users in control of application security. They no longer have to rely on developers properly securing their applications.

3.1.3. Compatibility

We consider compatibility with as many third-party applications as possible a crucial feature of a workable solution.

3.2. Attacker Model & Crypto Rules

Our attacker model assumes a malicious party that tries to gain access to a vulnerable application's stored or transmitted data. To this end, the attacker attempts to take advantage of vulnerabilities caused by the target application's misuse of cryptographic APIs. The modeled mode of attack depends on the specific crypto API misuse that is being exploited. For vulnerabilities in TLS host authentication or unprotected HTTP connections, the attacker is assumed to hold a Man-In-The-Middle (MITM) position somewhere between the client and server that enables the interception and modification of network traffic. Attacks against improper parametrization of cryptographic primitives are assumed to be carried out through a malicious application on the same device. It is worth noting that vulnerabilities in the *implementations* of the cryptographic APIs that can be exploited irrespective of their proper employment are considered out of scope. Additionally, we assume vulnerable applications to be benign in general, i.e. not actively trying to evade the mitigations put in place by our solution.

3.2.1. Crypto Rules

We define the vulnerabilities addressed in this paper as violations of well-established rules that consumers have to respect for ensuring the security of specific cryptographic APIs. As the basis for our rules, we use the comprehensive and recent set collected by Rahaman et al. [16], which we further refine for the scenario of vulnerability mitigation. We start this refinement process by ensuring the high level of precision needed for mitigating detected vulnerabilities while keeping negative side effects as rare as possible. Specifically, we modify our rules to only disallow non-cryptographic random number generators when their output is used in cryptographic operations. Similarly, we believe there are legitimate use cases for specifying custom TLS TrustManagers and HostnameVerifiers, so we strive to only mitigate cases where the custom implementations introduce vulnerabilities. Lastly, our experiments indicated that insecure cryptographic hash functions and hardcoded keys are so frequently used in communication with servers that mitigating them breaks most affected apps. Consequentially, we remove these two rules from our set. In their place, we introduce new rules concerned with reuse of nonce values, which has the same security repercussions as hardcoded values. The rules addressed by our mitigations are thus:

R01: Don't use unprotected HTTP connections.
R02: Don't verify TLS connections in insecure ways.
R03: Don't use predictable passwords for key stores.
R04: Don't use predictable passwords for key derivations.
R05: Don't use ECB mode for symmetric ciphers.
R06: Don't use predictable IVs for symmetric ciphers.
R07: Don't reuse IVs for symmetric ciphers.
R08: Don't use predictable salt values for key derivations.
R09: Don't reuse salt values for key derivations.
R10: Don't use predictable CSRNG seeds.

We use the term *predictable* to refer to values that were hardcoded or derived from an improper source of randomness. *Reused* values may have been securely generated but are used as a nonce multiple times, violating their uniqueness requirement.

3.2.2. Severity of Vulnerabilities

Our mitigations only address crypto misuse vulnerabilities of high or medium severity. For vulnerabilities that are hard to exploit or only offer limited gain for attackers, we argue that the potential of side effects introduced by mitigations (cf. Subsection 4.1) voids the small gain in security.

Severity ratings follow the reasoning by Rahaman et al. [16] that is guided by the difficulty and gain of exploitation. HTTP connections (**R01**) lack any form of protection for transferred data, so are trivially rated highly severe. Vulnerabilities in TLS connections (**R02**) immediately void all security properties of the protocol, so must be considered highly severe as well. Similarly, using predictable passwords (**R03**, **R04**) voids the confidentiality property of all operations the derived or stored keys are used for. ECB mode (**R05**) and predictable (**R06**, **R08**) or reused nonces (**R07**, **R09**) considerably weaken the confidentiality properties of cipher operations, so are rated medium severity. Lastly, predictable CSPRNG seeds (**R10**) have a history of causing critical vulnerabilities in applications running on legacy Android systems [9]. Because recent Android versions are not affected, we assign medium severity.

3.3. Challenges

We identify the following key challenges to accomplishing the goals stated above.

3.3.1. Precision

Before a certain API misuse can be mitigated, it has to be detected in the first place. The precision of this detection is of utmost importance to the mitigation, since we do not want to modify proper use of crypto APIs by forcefully applying our mitigation, nor can we tolerate missing legitimate misuse and leaving it unmitigated. Most established methods for crypto API detection rely on static analysis of the control flow graph of an application. However, this method has proven to be imprecise in real-world scenarios [8, 14, 15], suffering from a high degree of false positives due to unreachable code and false negatives due to dynamically loaded code, complex data dependencies or obfuscation. To satisfy the requirements on

precision, our solution employs dynamic runtime instrumentation. This approach enables us to guarantee that every crypto API call executed at runtime is detected and can be analyzed based on the concrete parameters passed from application code. In case of a detected misuse, parameters can be corrected at runtime, with no noticeable performance penalty. Additionally, relying on runtime instrumentation allows our solution to operate on-device in a self-contained manner, improving availability and privacy by not relying on any external server component.

3.3.2. Transparency to application code

All mitigations have to operate largely *transparently* to application code so that side effects on the functionality of affected programs can be avoided as much as possible. This is a particularly difficult task for cases where an additional parameter has to be transported between two separate API calls from code that has not provisioned for this possibility originally. CryptoShield solves this problem by taking advantage of a series of general observations about the JCA and its typical use in Android applications.

3.3.3. Practicality

We introduce our design as an emergency aid against the security vulnerabilities resulting from crypto API misuse. As such, we want it to be accessible and deployable to as large an audience as possible. Since maintaining a custom OS for the fragmented Android device landscape is not realistic and average Android users cannot be expected to be willing to flash a custom operating system, our concept needs to operate on stock firmware. Additionally, we consider rooting a device for deploying crypto API mitigation detrimental to the overall security. To surmount this challenge, our implementation employs a novel combination of performant package-level instrumentation and the Android DevicePolicyManager API.

4. Mitigating Crypto API Misuse

Conceptually, crypto API misuse can be described as a misparametrization of crypto APIs. For the crypto-savvy developer of a new application, avoiding crypto API misuse is usually as simple as choosing appropriate

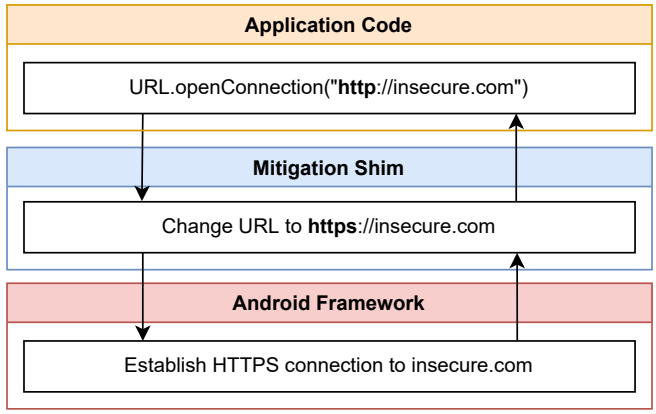


Figure 8.1: Our mitigations operate as shims between application code and the Android framework

parameters for all employed crypto APIs. However, once such a misparametrization has found its way into a compiled application, its mitigation is a much more complex task. Retrofitted mitigations have to operate transparently to existing application code, so that the offending application's functionality remains intact. To this end, all our mitigations operate as transparent shims between application code and the Android framework, as illustrated in Figure 8.1. It is also worth noting that modern applications are usually not solitary pieces of software, but part of a much larger ecosystem consisting of web services and client applications that operate together. Our mitigations must fit into this bigger picture as seamlessly as possible, i.e. fix crypto API misuse in all parts of code executed by the client locally, while retaining web service interaction functionality.

4.1. General Observations and Strategies

Before we describe our mitigations for specific crypto API misuse cases, we highlight a series of general observations and strategies that guide the design of our mitigation procedures.

4.1.1. Correspondence of Operations

First, we highlight the basic observation that many cryptographic operations are carried out in ordered pairs, where the second operation either

reverses, replicates or validates the first operation. This correspondence of two operations within the distributed ecosystem of a modern application yields three possible scenarios. It is worth pointing out that neither dynamic nor static analysis methods are capable of *reliably* foreseeing which of these scenarios will occur during further program execution.

In scenario Ⓘ, both of the corresponding operations are carried out locally in the application that is subject to the mitigations. In this case, we can trivially ensure that the mitigations applied during the first operation are correctly reversed or replicated in the second. An example would be changing a cipher mode of operation for both encryption and decryption of a local file. In scenario Ⓘ, the first of the corresponding operations is carried out in an external part of the ecosystem, while the second is carried out locally in the application that is subject to our mitigations. For covering this scenario, the second operation needs a way to know the origin of the data that is to be processed, so that the mitigation is only applied if local data is processed. This can be accomplished e.g. by introducing an additional communication channel between local corresponding operations. Alternatively, for operations that have clearly defined failure cases, we can try applying our mitigations and fall back to an unmitigated operation if a failure is detected. In scenario Ⓘ, the first operation is carried out locally, while the second is not. This case is problematic, since the external processing entity will not have any knowledge about the mitigations applied in the first, local operation. Still, due to the inability to foresee the later destination of the output from the local operation, mitigations have to be carried out unconditionally in the first operation. In Subsection 6.2, we demonstrate that this scenario is reasonably rare in popular real-world applications for us to consider it an acceptable failure case of our solution in line with our goal of establishing an *emergency* aid.

4.1.2. Operation Context & Context Transfer

Even for purely locally corresponding operations, a communication channel between two corresponding operations is required. It has to supply the second operation with the context needed for reversing or replicating the mitigations applied in the first operation. For some mitigations, this context is specific to a particular execution, so it cannot be stored e.g. in a global variable. As an example, consider initialization vector reuse in a symmetric cipher encryption operation. As part of the mitigation, a fresh IV has to be generated to replace the insecure value supplied

from application code. The same generated value later also has to be made available to the decryption operation of the resulting ciphertext. For our mitigations, we solve this problem by leveraging the output of the first operation as a communication channel to the corresponding second operation. In the IV reuse case described above, the IV generated by our mitigations is prepended to the resulting ciphertext and extracted at the corresponding decryption mitigation. Although this procedure changes the output size of the first operation, it still is transparent to most applications. This is because the Java Cryptography Architecture exposes functions for application code to query the output size of cryptographic operations. Due to the general opaqueness of cryptographic operations to the average application developer, the vast majority of Android applications uses these functions instead of hardcoding buffer sizes.

4.1.3. Introducing Fresh Entropy

Mitigations for misuse scenarios that involve an insecure key or password require the introduction of fresh entropy. To this end, our mitigations take advantage of the Android Keystore for generating and storing a fresh key we call key derivation key (KDK). Because the Android Keystore is limited to asymmetric cryptography on some devices, we use the KDK and the insecure key for deriving a fresh key that can be used for any cipher. This solution ensures that different installations of the same application always use unique keys.

4.1.4. Reuse Across Launches

Multiple covered vulnerabilities are concerned with reuse of nonces. For efficiently tracking used nonces, our mitigations maintain a simple nonce database.

4.1.5. Hardcoded Keys or Passwords

Our solution extracts hardcoded strings, character arrays and byte arrays from the target application package during instrumentation. This procedure can be implemented as a lightweight extraction algorithm that simply parses DEX data structures. It does not need any control flow information. For further reducing the runtime and memory overheads, the

procedure is restricted to values that follow the allowed or typical format of passwords or keys. For DEX files dynamically loaded during execution, data extraction is performed at runtime.

4.1.6. Predictability

In addition to values that are hardcoded, predictability can also be caused by employing randomness sources other than those that are cryptographically secure (`SecureRandom`, `/dev/(u)random`) for cryptographic operations. For identifying this problem, our mitigations maintain a cache of the most recent output of the Java `Random` API, which is particularly convenient to use, but not cryptographically secure.

4.2. Specific mitigations

In the following, we describe the detailed procedures for mitigating misuse of specific cryptographic APIs. For all mitigations, we assume the ability to instrument (or intercept) Java methods of our choice.

4.2.1. SSL/TLS

For mitigating flaws in TLS use (**R02**), we focus on the `SSLSocket` interface. While only a minority of applications directly access this low-level API, it forms the backbone of the vast majority of HTTPS networking libraries, which means they can all be covered through this single point of interception. Our mitigation intercepts the creation of `SSLSocket` instances and returns a wrapper around the originally created object. When the application code signals the beginning of application-level data transfer through calls to the `getInputStream()` or `getOutputStream()` methods, our mitigation queries a configurable certificate trust policy for determining the trustworthiness of the connection. In the event of a negative assessment, we immediately abort the connection, without ever having sent any application-level data over the compromised channel.

By performing the certificate check immediately before the target program starts sending application data, we know that the former considers the connection secure. Combined with the information about the legitimacy of the connection, we can take this knowledge as a basis for deducing whether an application uses insecure certificate validation logic.

The default certificate trust policy of our prototype implementation is based on the Trust-On-First-Use (TOFU) principle. When the first connection to a new host is made, its TLS certificate is cached by our policy code. Subsequent connections are then only allowed if the certificate presented by the TLS host is identical to the one encountered previously. Our prototype additionally supports a more complex certificate policy that builds on a notary web service. This web service takes the position of a certificate oracle, serving the legitimate TLS certificate for any host it is queried for. The certificate trust policy contacts the notary web service for any new host or certificate mismatch to determine whether its own connection to the target host is subject to an ongoing MITM attack. It is worth pointing out that both currently implemented certificate trust policies are only prototypes that illustrate the flexible nature of supported policies. Development of a more sophisticated policy that fully considers privacy and reliability aspects is deemed out of scope for this work.

4.2.2. Ciphers

We intercept the creation of `Cipher` instances to return a wrapper object backed by a custom security provider. Once the wrapper has received all required parameters from application code, it enforces the rules established in Subsection 3.2. Implicit or explicit uses of ECB mode for symmetric block ciphers (**R05**) are automatically upgraded to CBC mode. Whenever a predictable (**R06**) or reused IV (**R07**) is identified, a fresh value is automatically generated and passed to the wrapped primitive instead. All parameters changed as part of our mitigations are encoded in a ciphertext prefix so they are later available in the corresponding decryption operation. Figure 8.4 in Appendix 10.1 displays a simplified mitigation flow for implicit ECB mode using a ciphertext prefix. Our mitigation also overrides the `Cipher.getOutputSize()` method so that applications that pre-allocate their buffers can correctly accommodate the prefix.

4.2.3. Password-Based Encryption

Mitigations for PBE are integrated into the custom security provider described above. Whenever a PBE key is passed to the Cipher API, its parametrization is checked for rule violations. For PBE keys that are passed to the Cipher API in serialized form, our mitigation code keeps a map between recently serialized PBE keys and the corresponding

parameters that were used in their creation. If a predictable password (**R04**) is detected, a replacement password is derived from a key generated in the Android Keystore. Salt reuses (**R09**) detected through the nonce database or predictable salt values (**R08**) are mitigated by generating a fresh replacement value. All modified PBE parameters are encoded in the prefix of any ciphertext generated using the affected PBE key. While replacement salt values are completely embedded in the prefix, mitigations on other parameters are only encoded as flags interpreted in the corresponding decryption operation.

4.2.4. Random Number Generation

Our mitigation code installs a custom CSPRNG provider when an instrumented application launches. In the event of a predictable seed value (**R10**) passed from application code, we replace the explicit seeding with randomness obtained from the default system-provided entropy source.

4.2.5. HTTP

Most applications establish HTTP connections using either the `HttpUrlConnection` API or the open-source OkHttp library. Our mitigation intercepts calls to both of these implementations to check if a target server is contacted via the plain HTTP protocol (**R01**). If this is the case, the connection is automatically upgraded to HTTPS, i.e. an attempt is made to perform a TLS handshake with port 443 of the target host. If this handshake fails, our mitigation code falls back to the original unprotected HTTP url. We maintain a cache of hosts that were reachable over HTTPS in the past to thwart downgrade attacks that mask the availability of HTTPS on a server. A more sophisticated implementation may employ a pre-compiled list such as the one offered by DuckDuckGo Smarter Encryption [3].

4.2.6. KeyStore

When a predictable password is used for key storage (**R03**), a replacement password is derived by adding entropy from a key generated in the Android

[3] DuckDuckGo Smarter Encryption: `https://help.duckduckgo.com/duckduckgo-help-pages/privacy/smarter-encryption/`

Keystore. For all load operations on key stores, our mitigation code tries the replacement key (derived in the same way as for store creation) and falls back to the key supplied from application code upon failure.

5. Prototype Implementation

The CryptoShield prototype demonstrates how our mitigations can augment the existing security architecture and user experience of the unmodified Android platform. The implementation consists of two core modules, described in the following and illustrated in Figure 8.2. In total, our prototype implementation consists of over 25.000 custom lines of code.

The first part of our prototype is comprised of an instrumentation module that implements the individual mitigations described in Subsection 4.2. Once injected into a target application, the instrumentation module sits as a mitigation shim between application code and framework calls. Parameters passed from application code are inspected and collected for monitoring purposes. If any of the rules established in Subsubsection 3.2.1 is violated, parametrizations are upgraded transparently to application code following the procedures put forth in Subsection 4.2.

The second part of our prototype is an Android application we call CryptoShield Agent (CSA). The CSA offers a convenient solution for automatically injecting our instrumentation module into all applications installed on an unrooted Android device. It consists of a daemon service that starts immediately after booting the OS and a user interface that allows managing instrumented applications, as well as inspecting detailed crypto API usage reports. By listening for installation events broadcast by the system, the background service can automatically generate an instrumented version of every application the user installs or updates on the device. In the following, we highlight key aspects of our prototype implementation.

5.1. Instrumenting Applications on Unrooted Android Devices

Instrumenting running application processes on Android requires root privileges, which counteracts the goals established in Subsection 3.1. Instead of directly manipulating *processes*, our prototype instruments application *packages*. It takes advantage of the observation that the APK files of any

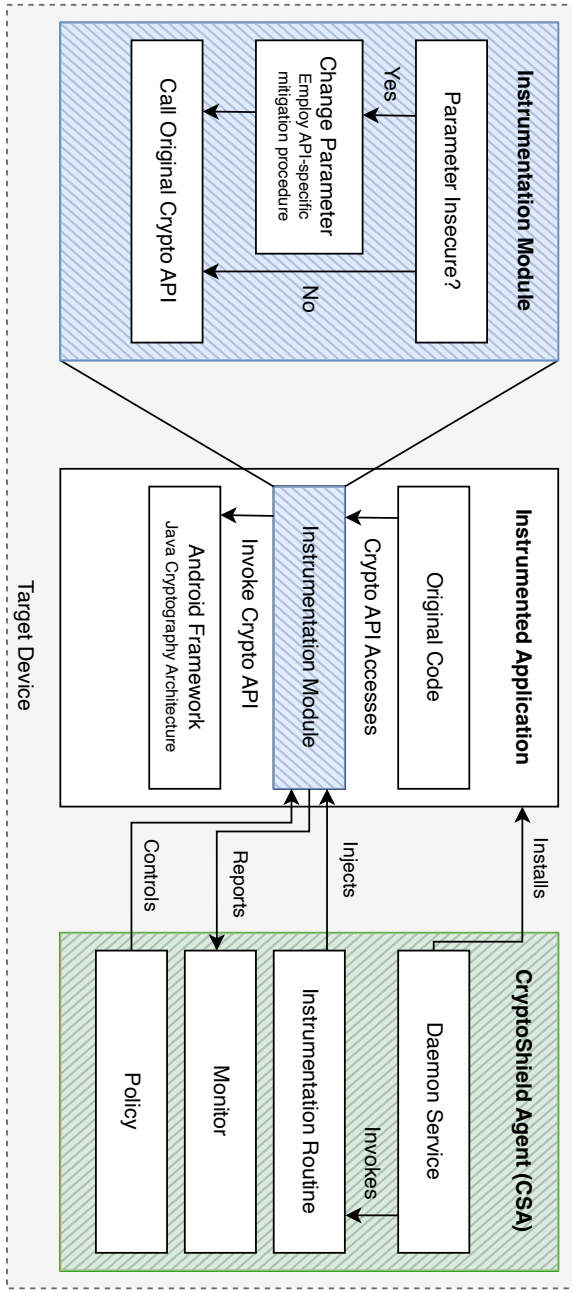


Figure 8.2: The two main components of our prototype implementation: *Instrumentation Module* and CryptoShield *Agent*

installed application remain stored on the device and can be accessed by other processes. This opens the possibility for the CSA to generate modified copies of arbitrary installed application packages. Our instrumentation routine unpacks the APK files to add a DEX file and a native library. The modified APK files are then compressed and signed using a freshly generated signing certificate. A cache ensures that applications originally signed with the same signing certificate will retain this property across instrumentation. Once the instrumented APK files are installed and the app is launched on a device, the injected native library manipulates data structures in the ART runtime to reroute crypto API calls to the injected DEX file. Our mitigation code in the injected DEX file can analyze and modify all passed parameters before it invokes the original implementation of the intercepted method. At a high level, this instrumentation approach was first presented by Styp-Rekowsky et al. [21]. Our prototype uses a custom implementation on top of low-level primitives from the open-source SandHook library [4] that lazily instruments methods on demand as their defining classes are loaded into runtime. It supports modern APK features such as Multi DEX or Split APKs, and all existing APK signature schemes.

5.2. Updating Instrumented Applications via Google Play

Modifying an APK package and resigning it with a different certificate usually breaks the common update route through Google Play. This is because all updates will still be signed with the original signing certificate and thus cannot be installed on top of the instrumented variant. As a remedy to this problem, our prototype changes the package name of every instrumented application, so that it can be installed alongside the original version. The original version is then disabled, so that it remains installed on the device but can no longer be launched. Whenever an update to an application is available, the user can install it from Google Play. The CSA will then automatically pick up the new version of the original application and produce a corresponding update for the instrumented variant.

5.3. Changing Package Names of Instrumented Applications

Manipulating the package name of a compiled Android application requires additional changes to the application package. CryptoShield intercepts

[4] SandHook Android ART Hook: `https://github.com/asLody/SandHook`

all framework methods where the package name is passed between application code and the outside world (e.g. in Intents or HTTP headers, even when accessed through Reflection APIs). Beyond the package name itself, all authorities of `Content- Providers`, custom permissions and account type identifiers are changed as well, so that they do not interfere with the respective components of the original version. Although our implementation goes to great lengths to cover as many scenarios as possible, a few issues remain in cases where it is unclear whether the original or modified package name yields correct behavior. For example, the original package name is sometimes used for fetching data from a proprietary configuration table in app resources. However, as detailed in Subsection 6.2, only a small number of real-world applications are affected by these issues. It is also worth pointing out that any issues caused by this tradeoff between practicality and application compatibility only affect our prototype implementation and are unrelated to the crypto API misuse mitigations themselves.

5.4. Installing and Disabling Applications without User Interaction

To install instrumented applications without user interaction, the CSA takes advantage of Android's DevicePolicyManager API. By implementing a `DeviceAdminReceiver` that is configured as the device owner, the CSA is granted access to the PackageInstaller API. Via this interface, it can install and uninstall applications, both in the form of a single APK file or multiple Split APKs supplied to a single installation session. Additionally, the DevicePolicyManager API permits the disabling of installed applications. It is crucial to note that the device owner role grants CryptoShield precisely the privileges it needs for autonomous operation, while allowing it to seamlessly *integrate* into the security architecture of the Android OS. This represents a key advantage over solutions that require root access on the device, which effectively *circumvents* the system's security foundations, with potentially disastrous consequences for inexperienced users. In order to streamline the process of installing and configuring the CSA application as a device owner, our prototype includes an easy-to-use companion tool for desktop computers. Once the Android device is connected to the computer via a USB cable, our software asks the user for explicit consent to the start of the setup procedure. A single button click then automatically installs and configures the CryptoShield system.

5.5. Support for MDM Deployment

As the CryptoShield Agent is built upon Android's DevicePolicyManager APIs, it can easily be integrated into corporate Mobile Device Management (MDM) solutions. CryptoShield thus represents a novel device management policy that can be rolled out to all devices in a company's device fleet for offering full application choice to users without significantly compromising data security. We envision a system that can be fully configured by the MDM administrator. Configuration options could include the enforced crypto rules and additional rule-specific adjustments. For example, it would be possible to deploy custom certificate validation policies for TLS or configure a custom HTTPS upgrade host list.

5.6. Crypto API Call Monitoring

In addition to mitigating crypto API misuse, our prototype also implements functionality for monitoring crypto API invocations from third-party applications. This functionality covers the same crypto APIs as our mitigations, i.e. TLS, Cipher, PBE, CSPRNG, HTTP and KeyStore APIs. We elect to restrict monitoring to this set of APIs for consistency with our mitigations (which are the main focus of CryptoShield), but further expansion is feasible. Monitoring is designed to provide users with real-time feedback on the vulnerabilities that CryptoShield protects them from. The instrumentation module inside instrumented applications collects relevant calls and the respective parametrizations. It then communicates all findings to the CSA. Through a monitor UI, the user can inspect all reported crypto API use and misuse.

5.7. Fallback Mechanism

As described in 4.1, although our mitigations are designed to retain functionality as much as possible, we anticipate that they might lead to malfunction in a small number of instrumented apps. To accommodate for this possibility, our prototype implements a fallback mechanism: Upon noticing malfunction, a user may manually toggle CryptoShield's mitigations for a specific application. We have deliberately made disabling mitigations an explicit user choice, because it means that users will be susceptible to attacks exploiting crypto misuse in affected applications. Still, CryptoShield's monitoring functionality allows users to track

crypto API invocations while mitigations are disabled, and issues alerts of concrete misuses that have been uncovered.

6. Evaluation

The evaluation in this section is comprised of five parts. In an automated large-scale analysis, we first demonstrate our prototype's efficacy on 1604 applications. Secondly, through a manual examination on the 99 most popular free applications from Google Play, we validate that applications are still functional and usable after installing CryptoShield's mitigations. Moreover, we provide per-API call runtime overhead measurements and complement the analyses on real-world applications with results from a synthetic benchmark that enables comparison to static detection-only tools. Lastly, we present a detailed case study on how our mitigations fix critical security vulnerabilities in two widely-used applications.

6.1. Automated analysis

To obtain a quantitative measure for the effectiveness of our mitigations, we ran an automated security analysis on 1604 applications before and after injecting our instrumentation module. We based our automated analysis testbed on the CryLogger crypto API misuse detector by Piccolboni et al. [14], which we adapted to our crypto rules and augmented with concepts from Sounthiraraj et al. [19] for improved detection of TLS issues. CryLogger runs applications in an Android emulator driven by random user input (here: 10k input events $\approx$ 23 % line, 33 % class coverage). Its custom emulator image logs calls to various crypto APIs. In our testbed, network connections were additionally subjected to a MITM attack by rerouting the emulator's network traffic through a MITM proxy server that logs all successful TLS connections [5]. All collected logs were then scanned for violations of the rules stated in Subsection 3.2. If a particular misuse was logged while running an original application, but not in the instrumented version, we consider this a successful mitigation. Building our automated evaluation on the CryLogger tool by Piccolboni et al. enables comparison to an independent baseline for the vulnerable applications in our sample set and their respective violations of our crypto API rules.

[5] CryptoShield's notary web service certificate trust policy (see Section 4.2.1) was used in the automated analysis

For executing this analysis, we used a bare-metal-virtualized Ubuntu 20.04 LTS system with 128 CPU cores and 128 GB of RAM, backed by a Dual AMD EPYC 7502 CPU with 1 TB of RAM. This setup allowed us to run our analyses on 6 virtual devices in parallel. Since the SandHook library utilized in our prototype implementation does not support the x86 instruction set, we ran the automated analysis on a slightly modified version that intercepts method calls by rerouting invocations inside the DEX code ahead of execution. The mitigation procedures inside the instrumentation module were not changed, so that the effectiveness measured here directly translates to that of the implementation described in Section 5. The set of 1604 tested applications consists of the free top-100 for 33 categories on Google Play as of January 2022, cleared from entries incompatible with either the x86 instruction set or the Android emulator. All 1604 applications in our set use some cryptographic API, while 388 applications violate some of our crypto rules.

In total, CryptoShield was able to mitigate all vulnerabilities induced by rule violations (as identified by our testbed) in 348 applications, which corresponds to 89.7 % of the vulnerable subset. Some vulnerabilities were mitigated in 372 applications (95.9 % of the vulnerable subset). The most commonly mitigated vulnerabilities concerned violations of **R02** (164 apps), **R01** (75 apps) and **R06** (66 apps). Violations against multiple rules were detected and mitigated in 94 apps. Although CryptoShield was able to produce installable packages for all tested samples, 105 of all instrumented applications (6.5 %) crashed at some point during execution.

40 applications were still vulnerable after instrumentation due to plain HTTP communication or insecure TLS connections from native code. As discussed in Subsection 7.5, our prototype does not cover native code, although obfuscated code or invocations of Java code through Reflection APIs or JNI are supported.

Detailed statistics for individual application categories can be obtained from Figure 8.3.

6.2. Manual analysis

Automated analysis is incapable of ensuring that applications retain their original functionality across our instrumentations and mitigations. Therefore, we manually collected additional compatibility and performance statistics for a smaller set of 99 applications on a physical Google Pixel

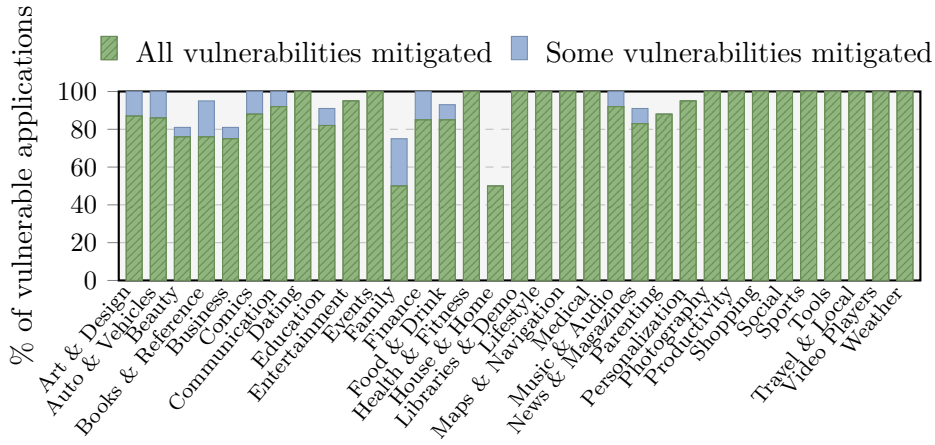


Figure 8.3: Automated analysis: Portions of vulnerable apps per category that could be partly or fully fixed

3 device running Android 11 (Build RQ3A.210605.005). As part of this analysis, every application was executed for two runs of 3 minutes, once in the original version and once in the instrumented version. In each run, a human operator navigated the application UI, simulating a typical end user exploring the software's functionality. The test set of 99 applications was obtained from Google Play in April 2022. It consists of the three most popular free packages for 33 different categories. For applications that required a paid membership or particular hardware peripheral, we picked a replacement application that immediately followed in the ranking. If a free user account was required, registration was completed before the timer was started.

6.2.1. Deployment Speed

Our prototype completed instrumentation of an application package in 10.7 seconds on average. Over all data collected, we can observe a strong correlation between the original APK size and the instrumentation duration ($\rho = 0.70$). This observation aligns well with the fact that the most computationally demanding task for our instrumentation routine is the compression and signing of the APK file. The distribution of instrumentation time across our test set is documented in Appendix 10.2.

6.2.2. Package Size

The relative package size change caused by CryptoShield's instrumentation averaged at 5 %. It is worth pointing out that the size of the instrumentation module injected into all APK files was identical. The variance in instrumented APK size overhead results from our instrumentation routine's ZIP compression implementation that differs from the one found in official Android build tools. For several applications, the overall APK size even decreased during instrumentation. The distribution of absolute APK size changes across the test set is detailed in Appendix 10.2.

6.2.3. Compatibility

From manually exploring the user interfaces of instrumented applications, we can report that 91.9 % of applications in our test set retained their original functionality. Although the instrumentation routine produced valid APK files for all tested applications, some showed signs of malfunction during execution. Through additional reverse engineering, we were able to trace back most of these cases to issues with package name modifications (3 applications), or signature checks that were intended to prevent malicious manipulation of the APK (3 applications). Two instrumented programs were no longer functional due to incompatibilities with CryptoShield's mitigation procedures (cipher mitigations broke server communication).

Given the non-exhaustive nature of manual UI exploration, the compatibility rate discussed here represents an upper bound. Still, we believe that manual exploration is the most accurate method for obtaining a representative compatibility measure in the absence of validation models that could be used for automated exploration.

6.2.4. Launch Overhead

CryptoShield's instrumentation module and mitigations are initialised during application launch. As the single most significant addition of instructions, this procedure has the potential to harm the user experience of instrumented applications. For our manual analysis, we therefore timed application launch time before and after instrumentation. We found the average launch time overhead of instrumented applications to be 571 ms,

which is hardly noticeable to end-users. Appendix 10.2 documents the distribution of absolute launch time overhead across the test set.

6.3. Per-API Runtime Overhead

Intercepting crypto API methods, inspecting arguments and applying mitigations inevitably has some impact on their runtime performance. To obtain a measure for the concrete overhead for each affected crypto API method, we timed their execution (1) in an uninstrumented app, (2) while only monitoring is enabled and (3) while only mitigations are enabled. For each API, measurements were collected for the most expensive use case and mitigation. For each measurement, we calculated the average from at least 1000 samples.

Our measurements indicate that the worst-case runtime overhead of our mitigations is about 101 ms (Cipher API calls where a key has to be derived via the hardware-backed AndroidKeyStore). Measurement details can be found in Appendix 10.3.

Please note that an average per-app runtime overhead cannot sensibly be provided for CryptoShield. Any such value calculated over the entire app lifecycle is entirely dependant on user behavior (how often intercepted methods are triggered).

6.4. Synthetic Benchmark and Comparison to Static Detectors

Although our manual and automated analyses use large datasets of popular real-world applications, they do not cover all our addressed crypto misuses. To provide a more complete picture of CryptoShield's mitigation robustness and to facilitate comparing its detection precision with state-of-the-art detection-only tools, we ran CryptoShield against a comprehensive synthetic benchmark. To the best of our knowledge, all existing crypto API benchmarks are designed for traditional *static* misuse *detectors*. In their original form, they are thus unsuitable for evaluating CryptoShield's *dynamic mitigation* capabilities. We therefore selected the CryptoAPI-Bench [1] benchmark as the best match for our misuse rules, and adapted it to our setting. Adaptations ensure executability of all test cases, and introduce checks for retained functionality. For example, one of these checks asserts that encryption followed by decryption yields

Category	Cases TP	Cases TN	C.Shield FP	C.Shield FN	C.Guard FP	C.Guard FN
InsecureTrustManager (**R02**)	8	0	0	0	0	8
InsecureHostnameVerifier (**R02**)	4	1	0	0	1	1
EcbCrypto (**R05**)	7	2	0	0	1	1
Http (**R01**)	7	3	0	0	0	1
NoHostnameVerifier (**R02**)	2	0	0	0	0	1
PredictableIV (**R06**)	10	2	0	0	1	1
PredictableKeyStorePW (**R03**)	9	3	0	0	1	1
PredictablePBEPW (**R04**)	11	2	0	0	1	2
PredictableSeed (**R10**)	16	3	0	7	2	6
ReusedIV (**R07**)	9	2	0	0	0	9
ReusedSalt (**R09**)	8	2	0	0	0	8
PredictableSalt (**R08**)	11	3	0	6	1	1
Total	102	23	0	13	8	40

Table 8.1: Synthetic Benchmark test cases and results per misuse category. **T**rue **P**ositive and **T**rue **N**egative indicate test case ground truth. **F**alse **P**ositive and **F**alse **N**egative indicate errors in CryptoShield's and CryptoGuard's detections. Corresponding crypto rules in braces.

the original plaintext even if CryptoShield replaces the key to mitigate its predictability. Furthermore, we added 29 test cases that closely follow the structure of existing cases, but fill CryptoAPI-Bench's gaps in coverage of misuse rules **R07** and **R09**. Lastly, we introduced new test cases for covering Crypto API invocations through Java Reflection APIs (13) and complex TLS certificate validation issues (3). Test cases not related to CryptoShield's rules were removed. In total, our benchmark comprises 125 test cases, of which 102 contain misuse and 23 were designed to catch false positives. For quantitatively evaluating CryptoShield's detection capabilities against a state-of-the-art static detection-only tool, we also analysed the same benchmark with CryptoGuard [16].

Benchmark details and results for both CryptoShield and CryptoGuard can be found in Table 8.1. CryptoShield's detection correctly identified 89 of the misuse cases (false negative rate: 13 %) and did not produce any false positives. All missed misuses were caused by hardcoded CSPRNG seeds (7) or PBE salts (6). Most issues can be traced back to a limitation in our prototype, which for efficiency reasons only considers hardcoded

byte arrays of at least 4 bytes. This may be addressed in future work. For all test cases where CryptoShield detected a misuse, the consistency checks indicated that mitigation procedures retained existing functionality.

While CryptoGuard performed slightly better than CryptoShield on hardcoded values, it had higher false positive (9 cases ≈ 36 %) and false negative (41 cases ≈ 37 %) rates overall. The static detector was unable to identify insecure TLS certificate validation or nonce reuse at all. It is worth noting that CryptoAPI-Bench was designed for evaluating static misuse detectors, so some of its test case variations specifically challenge static detectors, but look identical to CryptoShield. A benchmark for objectively comparing static and dynamic crypto API misuse detectors has not been proposed yet. Dynamic *detectors* usually suffer from poor code coverage. However, for dynamic *mitigation*, code coverage is irrelevant, as long as we can ensure that all *executed* misuse is *mitigated* (see Section 6.1). CryptoShield is designed for preventing exploitation of crypto API misuse, not for generating an extensive list of misuse hidden in code bases.

Appendix 11 shows additional benchmark implementation details.

6.5. Case Studies

To demonstrate how CryptoShield prevents leakage of sensitive user data in real-world scenarios, we conduct case studies on two popular applications that contain critical vulnerabilities caused by crypto API misuse. Case study samples were chosen based on *1)* number of installations, hence real-world representativeness, *2)* illustrative quality, i.e. suitability for demonstrating causes and consequences of crypto API misuse, and *3)* sensitivity of processed data. Displayed vulnerabilities were responsibly disclosed to the respective vendors.

6.5.1. Banggood

Banggood is a Chinese online retailer whose Android app has been downloaded more than 10 million times from Google Play as of August 2022. The application uses insecure `TrustManager` and `HostnameVerifier` implementations, allowing a MITM attacker to intercept the data exchanged between the Banggood client program and server. Although many of the application's requests are protected by a proprietary RSA scheme on top

of TLS, the customer registration endpoint is not, meaning that complete user accounts can be compromised.

When installing the application on a system protected by CryptoShield, the data disclosure is successfully prevented. From the CSA monitor interface, we can confirm that the network request for registration fails during an ongoing MITM attack because our system aborts the connection after it detects the MITM attack.

6.5.2. Amaze File Manager

Amaze File Manager is an open-source file management application for the Android platform. Together with its derivates, the program has reached an audience of more than 10 million users according to Google Play. Beyond the basic feature set common in this software category, Amaze also supports encryption and decryption of sensitive files.

CryptoShield detects that the same IV is used for AES-GCM encryptions of multiple different files. Experiments with an unmodified version of the file manager confirm that from two ciphertexts C_1 and C_2 encrypted using the same key and one of the plaintexts P_1, we can calculate the other plaintext P_2 as $P_2 = C_1 \oplus C_2 \oplus P_1$. Since encrypted files are saved in public storage, this means that any other application with appropriate read access may decrypt files just by guessing one original plain text.

The flaw can also be confirmed from the source code of the Amaze File Manager application, where the corresponding class holds a comment about this exact problem. The example underlines that developers are often overwhelmed by the security requirements of modern applications, lacking the experience for ensuring proper data protection

When the same experiment is conducted after applying CryptoShield to the vulnerable app, fresh IVs are injected for every encryption, so that no information about P_2 is leaked.

7. Discussion & Future Work

This section elaborates on the issues raised in Section 5 and highlights additional limitations. It also explores how our approach can be extended to related research fields.

7.1. Robustness

Although our mitigations were designed to keep side-effects at a minimum, some of them have the potential to lead to issues in an app's existing functionality. Example scenarios are applications that use insecure ECB mode in server communication, or that pass prefixed ciphertexts to fixed-size buffers in native code. However, evaluation on real-world apps (Section 6.2) shows that these issues combined only affect a very small number of applications (2 %), so that CryptoShield's security benefits far outweight its potential for introducting side-effects. For affected applications, users can take advantage of CryptoShield's fallback mechanism (Section 5.7). It is worth noting that our instrumentation may be incompatible with advanced app packers that hook Android Runtime APIs. However, to the best of our knowledge, these techniques are only used in malicious apps that seek to evade analysis. As discussed in Section 3.2, malicious applications are out of scope for CryptoShield.

7.2. Implementation Security

By utilizing the DevicePolicyManager APIs, CryptoShield assumes a security-sensitive role on the Android device. If an attacker was to find and exploit a vulnerability in CryptoShield, they could carry out operations on the device that might harm the user. For our prototype, we took great care to follow security best practises for the Android platform, and we publish our source code for examination by fellow security researchers[6]. Before deploying our solution in a real-world scenario, an independent code audit is advisable for ascertaining the implementation's security and trustworthiness.

7.3. Signature Checks

For instrumenting third-party applications on unrooted devices, our prototype implementation modifies APK files, re-signing them with a new certificate in the process. Like any other solution that takes advantage of this approach, it is affected by signature checks that some developers integrate into their programs as protection against maliciously modified

[6]Source code is available at `https://extgit.iaik.tugraz.at/fdraschbacher/cryptoshield`

redistributions. While it would be possible to work around many implementations of signature checks using our instrumentation technology, we respect the intention behind these barriers and refrain from pursuing ideas for circumventions. Only 3 % of applications in our manual evaluation test set were affected by this problem. We argue that developers who are versed enough to implement package signature checks are more likely to be aware of and follow best practices for cryptography in general. It is worth pointing out that this is a limitation of the prototype implementation and not our mitigation procedures. Another implementation may choose a different trade-off between usability and compatibility or take advantage of an entirely new instrumentation technique.

7.4. Bookkeeping Costs

CryptoShield maintains runtime databases for detecting and mitigating nonce reuses. Inevitably, this bookkeeping incurs some runtime overhead, e.g. when compared to the one-time cost of a static approach that analyses the control flow graph of an application. However, we argue that the bookkeeping runtime cost is not critical here - it is not noticable to the user in our prototype (see Section 6.3). Much more importantly, CryptoShield's runtime instrumentation can take advantage of data that static analysis never has access to. It enables mitigation of misuse cases that are not noticable to static analysis, such as reuse of dynamically generated nonces or non-trivial issues in TLS certificate validation.

7.5. API Coverage

Our prototype implementation only detects and mitigates misuse of cryptographic APIs provided by the system frameworks or by known third-party libraries. Applications that integrate from-scratch implementations of primitives or native third-party crypto libraries are not covered by our protection measures. Still, it is worth noting that our mitigation procedures can in principle be extended to any implementation of the covered APIs. Obfuscated code and invocations of Java code through Java Reflection APIs or the Java Native Interface are already supported in our prototype implementation.

7.6. Device Coverage

We implemented our prototype for the popular Google Pixel series of devices and the three most popular iterations of the Android OS (versions 9-11)[7] as of August 2022. The system was extensively tested on a Google Pixel 3 running Android 11. Given the slight variations of ART behavior between different devices and Android versions, we anticipate minor changes before support for other devices can be guaranteed. We also have to point out that our current implementation only works on the ARM instruction set, which covers the vast majority of available Android devices. Extending support to additional CPU platforms is feasible and only a matter of resources.

8. Related Work

In this section, we highlight previous publications in the domain of crypto API misuse on the Android platform.

8.1. Mitigating Crypto API Misuse

The only previous works capable of mitigating crypto API misuse in compiled Android applications are *Pin It!* by Buhov et al. [3] and *CDRep* by Ma et al. [11]. The first requires root permissions and only addresses issues with TLS misuse. CDRep follows similar high-level principles as our solution, but differs significantly in its approach and practicality. In contrast to CDRep, CryptoShield (1) can be directly deployed to protect a user's device in a self-contained manner, (2) addresses many more misuse cases, among them particularly wide-spread and critical TLS issues, (3) emphasizes retaining functionality, and (4) does not suffer from the resource-intensiveness and imprecision of static CFG analysis (CDRep only mitigates the first misuse per app method and takes 19 seconds per misuse for patch generation). Unfortunately, Ma et al. were unable to share their source code or data set with us, so we cannot provide quantitative comparisons.

[7] As indicated by Google's official Android version distribution chart inside the Android Studio IDE

Beyond these immediately related works, the closest publication to CryptoShield in terms of the addressed security vulnerabilities is FireBugs by Singleton et al. [18], which employs static analysis and pattern-based patching for retrofitting security best practices into applications that contain crypto API misuse. However, their solution only produces source-level patches that can serve as a guideline for application maintainers. It is not capable of protecting end users. Newbury et al. [12] combine static analysis and the desktop JVM's instrumentation API for hotpatching crypto API misuse in enterprise Java applications. However, their trivial mitigation strategies completely neglect the potential for side effects. Conceptually, our solution bears similarities to the work by Bates et al. [2], which proposes a solution for fixing SSL certificate validation on Ubuntu by dynamically linking a shim between third-party applications and SSL libraries. Some Android-specific publications suggest source-level mitigations for crypto API misuse, e.g. by facilitating the configuration of certificate pinning through XML files [7, 20].

8.2. Detecting Crypto API Misuse

Most previous works on crypto API misuse focus on detection in compiled application packages. The most recent major publications in this field of research have been provided in Section 1. The current state-of-the-art purpose-built tools ready for practice are generally considered to be CryptoGuard by Rahaman et al. [16] and CogniCrypt/CrySL by Krueger et al. [10].

9. Conclusion

Official countermeasures and previous efforts by the research community were ineffective in eliminating the common security vulnerabilities in popular Android applications that are induced by misuse of cryptographic APIs. In this paper, we devised a set of generic mitigation procedures for the most severe crypto API misuse scenarios. Based on prototype implementations of our mitigation procedures, we provide a tool that can protect users against crypto API misuse vulnerabilities until a long-term solution has been found. Our system's on-device daemon service is capable of automatically injecting an instrumentation module into all application packages installed on a device. This module subsequently monitors crypto

API calls inside a target application and mitigates identified misuse transparently to application code. We showed that CryptoShield effectively mitigates vulnerabilities in real-world applications, is compatible with the vast majority of Android applications, and introduces minimal overhead.

References

[1] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. In: IEEE Cybersecurity Development (SecDev). IEEE, 2019, pp. 49–61 (p. 232).

[2] Adam Bates, Joe Pletcher, Tyler Nichols, Braden Hollembaek, Dave Tian, Kevin R. B. Butler, and Abdulrahman Alkhelaifi. Securing SSL Certificate Verification through Dynamic Linking. In: ACM SIGSAC Conference on Computer and Communications Security (CCS). 2014 (p. 239).

[3] Damjan Buhov, Markus Huber, Georg Merzdovnik, and Edgar R. Weippl. Pin it! Improving Android network security at runtime. In: IFIP Networking Conference, Networking and Workshops. 2016 (pp. 208, 238).

[4] Florian Draschbacher and Johannes Feichtner. CryptoShield: Automatic On-Device Mitigation for Crypto API Misuse in Android Applications. In: AsiaCCS. 2023 (p. 205).

[5] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In: ACM SIGSAC Conference on Computer and Communications Security (CCS). 2013 (p. 210).

[6] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. Why eve and mallory love android: an analysis of android SSL (in)security. In: ACM Conference on Computer and Communications Security (CCS). 2012 (p. 210).

[7] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking SSL development in an appified world. In: ACM SIGSAC Conference on Computer and Communications Security (CCS). 2013 (p. 239).

[8] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In: 35th International Conference on Software Engineering (ICSE). 2013 (p. 215).

[9] Alex Klyubin. Some SecureRandom Thoughts. Aug. 2013. URL: https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html (p. 215).

[10] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. CogniCrypt: supporting developers in using cryptography. In: IEEE/ACM International Conference on Automated Software Engineering (ASE). 2017 (p. 239).

[11] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In: 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS). 2016 (pp. 208, 210, 238).

[12] Kristen Newbury, Karim Ali, and Andrew Craik. Hotfixing misuses of crypto APIs in Java programs. In: 31st Annual International Conference on Computer Science and Software Engineering (CASCON). 2021 (p. 239).

[13] Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications. In: 30th USENIX Security Symposium. 2021 (pp. 208, 210).

[14] Luca Piccolboni, Giuseppe Di Guglielmo, Luca P. Carloni, and Simha Sethumadhavan. CRYLOGGER: Detecting Crypto Misuses Dynamically. In: 42nd IEEE Symposium on Security and Privacy (SP). 2021 (pp. 215, 228).

[15] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In: 21st Annual Network and Distributed System Security Symposium (NDSS). 2014 (p. 215).

[16] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In: ACM SIGSAC

Conference on Computer and Communications Security (CCS). 2019 (pp. 208, 214, 215, 233, 239).

[17] Jingjing Ren, Martina Lindorfer, Daniel J. Dubois, Ashwin Rao, David R. Choffnes, and Narseo Vallina-Rodriguez. Bug Fixes, Improvements, ... and Privacy Leaks - A Longitudinal Study of PII Leaks Across Android App Versions. In: 25th Annual Network and Distributed System Security Symposium (NDSS). 2018 (p. 210).

[18] Larry Singleton, Rui Zhao, Harvey P. Siy, and Myoungkyu Song. FireBugs: Finding and Repairing Cryptography API Misuses in Mobile Applications. In: IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC). 2021 (p. 239).

[19] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In: 21st Annual Network and Distributed System Security Symposium (NDSS). 2014 (p. 228).

[20] Vasant Tendulkar and William Enck. An Application Package Configuration Approach to Mitigating Android SSL Vulnerabilities. In: CoRR abs/1410.7745 (2014) (p. 239).

[21] Philipp Von Styp-Rekowsky, Sebastian Gerling, Michael Backes, and Christian Hammer. Idea: Callee-Site Rewriting of Sealed System Libraries. In: Engineering Secure Software and Systems - 5th International Symposium (ESSoS). 2013 (p. 225).

10. Appendix

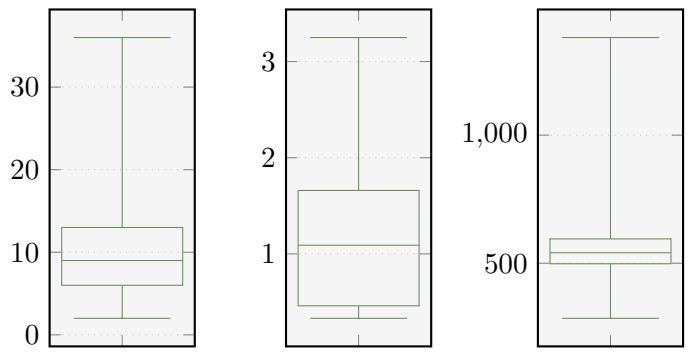
10.1. Mitigation Procedures

Figure 8.4 illustrates the simplified mitigation procedure for implicit ECB mode in the Cipher API. ① Cipher creation is intercepted to return a wrapper object. ② In encryption mode, ECB mode is upgraded to CBC upon initialization and a fresh salt is generated. ③ The salt and an upgrade flag are prepended to the actual ciphertext ④ The prepended ciphertext is returned to application code, which later passes it to decryption. ⑤ Cipher creation is intercepted to return a wrapper object. ⑥ In decryption mode, the wrapped cipher is initialized with parameters from application code. ⑦ If a ciphertext prefix is found, the wrapped cipher is re-initialized

with the parameters extracted from the prefix. Extracting the prefix from data streams fed through the cipher adds significant complexity to the complete implementation.

10.2. Manual Analysis

Figure 8.5 shows the distribution of instrumentation time, APK size overhead and launch time overhead across our manual analysis test set.



(a) Instrumentation Time (Seconds) (b) Size Overhead (MB) (c) Launch Time Overhead (Milliseconds)

Figure 8.5: Manual analysis: Distribution of performance characteristics across the test set

10.3. Per-API Runtime Overhead

This section provides details on our per-API runtime overhead measurements. For each API, measurements were collected for the most expensive use case. For example, `Cipher.update()` mitigation measurements are for the first decryption buffer in ECB mode and use a PBE key with hardcoded password and salt, so that it involves CryptoShield extracting the ciphertext prefix and deriving a fixed PBE key. For each measurement, we calculated the average from at least 1000 samples. Table 8.2 contains the exact per-API runtime overheads of CryptoShield's monitoring and mitigation. It is worth noting that monitoring was disabled while mitigation was measured and vice versa. The most significant overheads were observed for mitigations that involve deriving a key via

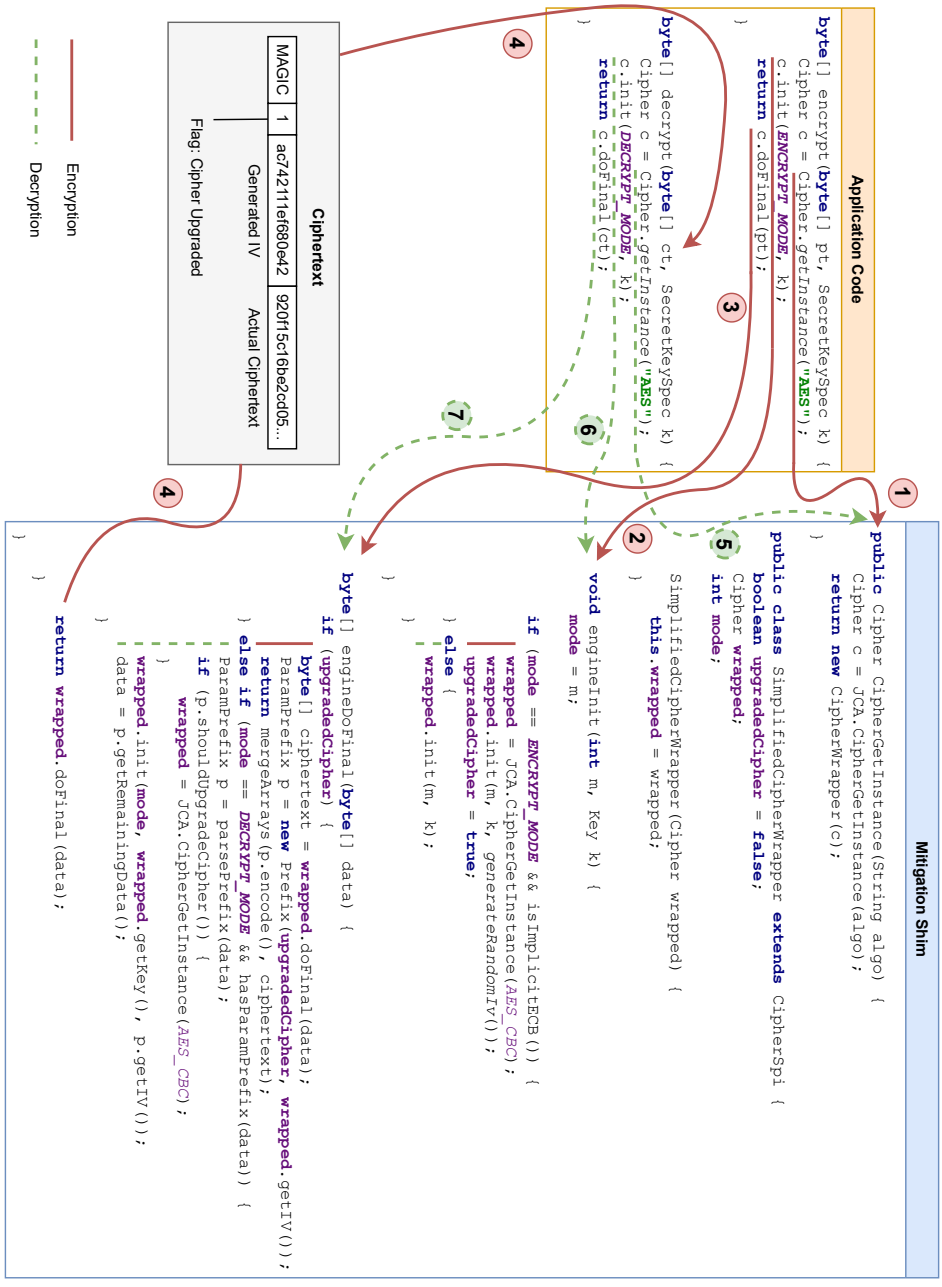


Figure 8.4: Simplified mitigation procedure for implicit ECB mode.

the hardware-backed AndroidKeyStore. We attribute the speedup of `SecretKeyFactory.generateSecret()` in the instrumented app to measurement variations. CryptoShield's implementation of this API is only a wrapper of the original method. PBE key issues are mitigated once they are used in cipher operations.

API	Orig.	ℹ	✔
Cipher.getInstance()	0.01	0.02	0.02
Cipher.init()	0.01	1.54	101.00
Cipher.update()	0.03	1.20	95.12
Cipher.final()	0.00	1.19	95.42
Cipher.updateAAD()	0.00	0.00	0.00
KeyStore.store()	140.94	5.20	87.07
KeyStore.load()	140.60	3.85	86.99
SecretKeyFactory.getInstance()	0.00	0.01	0.01
SecretKeyFactory.generateSecret()	13.66	-0.08	-0.16
SecureRandom.getInstance()	0.00	0.02	0.02
SecureRandom.setSeed()	0.00	0.57	0.02
SSLSocketFactory.createSocket()	42.90	3.49	1.96
SSLSocket.getInputStream()	9.34	8.03	7.00
SSLSocket.getOutputStream()	10.09	9.04	5.60
URL.openConnection()	69.85	34.19	59.08
okhttp3.Request.Builder.url()	0.03	0.63	0.07

Table 8.2: Per-API runtime overheads in milliseconds. Orig. displays the API runtime in an unmodified app, while ℹ and ✔ contain overheads (relative to Orig.) for monitoring and mitigations, respectively.

11. Synthetic Benchmark Details

In this section, we provide additional implementation details for the synthetic benchmark employed in our evaluation (Section 6.4).

11.1. Functionality Checks

Where possible, we implemented functionality checks by extending CryptoAPI-Bench's existing testcases to run a full sequence of corresponding operations, so that we can compare the final output to the original input.

Listing 11.2 displays a simplified example of this principle for using ECB mode with a symmetric cipher. For cases involving TLS communication, we check for retained functionality by establishing a connection with and without an ongoing MITM attack, ensuring that the latter succeeds and the former is prevented by CryptoShield's mitigations. Test cases involving CSPRNG seeding are assumed to always stay functional, since CryptoShield never influences the structure of generated values.

11.2. Reflection Test Cases

We extended our benchmark to cover invocations of cryptographic APIs through Java Reflection APIs. For cases that involve interfaces implemented by application-level code, we utilize Java's Dynamic Proxy infrastructure to construct a proxy implementation that forwards all calls to a handler. A simplified example of this approach can be found in Listing 11.1.

```
public class Case1 implements InvocationHandler {
    public static void main(String[] args) {
        HostnameVerifier hv = (HostnameVerifier)
        ↪  Proxy.newProxyInstance(...,
                new Class[] {HostnameVerifier.class},
                new Case1());
        // We're simulating a MITM attack here!
        if (hv.verify("mitm.com", session)) {
            ssl.getOutputStream().write(data);
        }
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[]
    ↪  args) throws Throwable {
        return true; // Insecure!
    }
}
```

Listing 11.1: Example test case using `java.lang.reflect.Proxy`

```
public class EcbCase1 {
    public static void main(String[] a) throws ... {
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        SecretKey key = keyGen.generateKey();
        Cipher c = Cipher.getInstance("AES/ECB/PKCS5Padding");
        c.init(Cipher.ENCRYPT_MODE, key);

        byte[] data = new byte[128];
        byte[] cipherText = c.doFinal(data);
        cipher.init(Cipher.DECRYPT_MODE, key);
        byte[] plainText = c.doFinal(cipherText);
        if (Arrays.equals(plainText, data))
            System.out.println("Still functional");
    }
}
```

Listing 11.2: We adapted CryptoAPI-Bench to check retained functionality (adaptations highlighted).

9

A2P2: An Android Application Patching Pipeline Based On Generic Changesets

Publication Data

Florian Draschbacher. A2P2 - An Android Application Patching Pipeline Based On Generic Changesets. In: ARES. 2023

Contributions

The author of this thesis is the sole author of the paper, and originator of the idea, concept and implementation for A2P2.

A2P2: An Android Application Patching Pipeline Based On Generic Changesets

Florian Draschbacher

Graz University of Technology

Abstract

Inspecting and manipulating runtime behavior of Android applications is a common need in mobile security research. However, existing tools lack a holistic application-agnostic approach. They either require changes to be manually adapted to each target application, or they focus exclusively on executable code parts, neglecting the key role the application manifest and resources play in the Android ecosystem. This limits their use for research purposes, where a specific series of modifications on various app components frequently has to be applied to a whole body of applications.

In this paper, we present A2P2, a flexible patching pipeline for compiled Android applications. Our system encompasses a custom declarative patch format for specifying complex manipulations on all parts of an application package. Patch projects are developed inside the Android Studio IDE and compiled into patch packages. These may then be applied to an arbitrary number of application package (APK) files through our flexible patching pipeline implementation. Existing pipeline stages may be freely arranged and augmented with user-supplied custom stages so that entirely new sophisticated transformations may be implemented from a range of core primitives. For manipulating Dalvik bytecode, we provide two different rewriting backends and an abstraction that enables addition of new rewriting technologies transparently to patch projects.

We demonstrate A2P2's efficiency and efficacy by providing estimates for deployment speed and effects on compatibility, application size, and runtime performance for typical use cases. Lastly, we implement A2P2 patches that reproduce previous research and facilitate common security analysis tasks.

1. Introduction

As part of their work, mobile security researchers commonly face the need to inspect and manipulate the runtime behavior of closed-source Android applications. This may be necessary for analyzing a suspectedly malicious program, reverse-engineering a proprietary protocol, assessing the security of an unknown piece of software, or detecting and mitigating a particular class of vulnerabilities.

Over the past decade, a number of solutions for inspecting and manipulating application runtime behavior have been proposed. From these, three different technical approaches can be derived, each with its own advantages and disadvantages.

1. *System Modifications*. Root privileges can be utilized on the Android platform for either replacing parts of the system components that form the application runtime [8], or for attaching to an application process during execution [5]. However, gaining root privileges requires modifying the OS installation, which impacts the entire software stack on the device and not just individual applications. Additionally, application patching tools proposed as part of research are often designed for aiding inexperienced developers or advanced users in retrofitting security improvements into compiled apps. In these scenarios, the resulting apps are installed on real-world devices, where making root permissions available to the user must be considered a security risk.

2. *Virtual Containers*. Some solutions execute target applications within the context of a container application, e.g. [3]. From the perspective of the OS, the target application lives in the process of the container application, which grants it arbitrary capabilities for runtime manipulation without requiring root permissions. However, this setup also considerably limits the OS integration of the target application. Additionally, it requires the container application to statically pre-request all permissions any contained application may need later, which qualifies as an extreme case of over-permissioning.

3. *Repackaging*. Applying changes directly onto the application package (APK) file [14, 16] allows modifying any aspect of the contained files, such as the executable code or the application manifest. All changes only affect the target application and can be conveniently redistributed for installation on unmodified Android systems. However, modifying the APK file requires resigning it, which may lead to issues. Still, in research

scenarios where root privileges are not available, repackaging is the most reliable way for inspecting and manipulating Android application runtime behavior. We therefore concentrate on this technology in the following.

Although multiple repackaging tools are readily available for applying user-specified changes to a given APK file, their use for research purposes is limited. Many research use cases involve applying a given change to a whole body of diverse applications. However, available tools either require changes to be manually adapted to each individual target APK file or only allow modifying the executable parts of the APK, entirely neglecting the central role of the Android manifest as the contract between the application and OS. Due to this lack of existing tooling, researchers commonly had to implement their own purpose-built tools [4, 9, 12, 17], wasting resources that they may instead have invested into their core contributions.

We introduce A2P2 to address this evident gap in research tooling. To the best of our knowledge, our patching pipeline is the first solution for developing and deploying holistic application-agnostic patches to compiled Android applications.

Our key contributions are:

- We propose an application-agnostic patch format that allows declaratively specifying complex changesets to Dalvik bytecode, Android manifest and application resources, as well as addition of arbitrary files into a compiled Android application package. The format entails a developer-facing source variant (*A2P2 patch project*) as well as a compiled binary variant (*A2P2 patch package*) for feeding into the deployment pipeline.

- We present a suite of tools for developing and compiling changesets, as well as for applying them to concrete compiled Android application packages (APK files). The deployment components are organised in a fully customizable patch deployment pipeline. We provide full implementations of all pipeline stages needed for deploying our patch format. Additionally, our solution allows users to supply custom pipeline stages that may specify complementary changes procedurally, utilizing our low-level manipulation primitives for various Android-specific file formats.

- We provide performance and compatibility metrics for evaluating A2P2's efficacy and showcase a set of example patches that demonstrate how our patching pipeline facilitates mobile security research.
- We make our entire toolchain (including development tools and deployment pipeline) available to the research community under an open-source license. [1]

The remainder of this paper is organized as follows: Section 2 lays out the necessary background knowledge. Section 3 then introduces the overall architecture of the A2P2 system. Subsequent Sections 4 and 5 describe our patch format design and pipeline implementation, which are then evaluated in Section 6. We discuss limitations of our concept and plans for future work in Section 7, highlight related publications in Section 8 and conclude this paper in Section 9.

2. Background

In this section, we summarize background information on Android application development, the APK format, and the ART runtime.

2.1. Android Application Development Process

Google provides an official Android Studio IDE for developing Android applications. Key components of an application project are:

- *Android manifest*: An XML file declaring the permissions required for operation and the functionality (Activities, Services, ContentProviders) exposed to the rest of system. If any such externally exposed characteristic is to be added to or removed from the application, the change needs to be reflected in the manifest for the system to pick it up.
- *Program code*: Google recommends implementing program logic in Java or Kotlin, although integrating native code is possible for performance-critical program parts or incorporation of existing components. The Java Native Interface (JNI) enables interaction between native (C/C++) and managed (Java/Kotlin) code.

[1] Source code is available at `https://extgit.iaik.tugraz.at/fdraschbacher/a2p2`

- *Resources*: These include files that define UI layout structures, strings for localisations, images and many more.
- *Other assets*: Files that are copied into the application package in verbatim during build, i.e. in an unaltered form.

During build, various human-readable parts of an application project are transformed into representations more suitable for execution and packed into an Android Application Package (APK) file. The Gradle build system and an Android-specific plugin are used for coordinating and configuring the tasks involved in the build process. Java and Kotlin code is compiled into Java bytecode using the javac compiler, then further transformed into Android-specific Dalvik bytecode by the d8 build tool. The Android manifest and XML resources are compressed into an Android-specific binary XML representation. A resource index is assembled that assigns each resource an identifier that can be used for cross-referencing.

2.2. Android Package Format

The APK file format is used for deploying a compiled Android application to a device for installation. At the outermost layer, an APK file is a ZIP archive whose contents follow a well-defined structure. The core components of an APK file are stored in the top level of the ZIP container. These comprise the Android manifest in the AXML binary XML format and one or multiple Dalvik Executable (DEX) files that hold the Dalvik bytecode for the Java and Kotlin classes that implement the program logic. The ARSC file plays a key role in organizing the resources within an APK file. It contains a detailed resource index that not only allows unique identification of resources but also is structured in a way that enables dynamically selecting resource values at runtime depending on device characteristics such as screen size or configured locale.

Before an APK file can be installed onto an Android device, it needs to be signed with a self-signed developer certificate. This ensures that application updates can only be installed if they stem from the same developer as the original installation.

For efficiency reasons, modern Android applications may be composed of multiple APK files that carry the same package name and are signed with the same developer certificate. All individual APK files need to be installed onto the device for the application to be operational.

2.3. Android Runtime

The Dalvik bytecode stored in DEX files encodes program functionality in the instruction set of a register-based virtual machine. This ensures a compiled application is compatible with any Android device, no matter its native CPU architecture. The Android Runtime (ART) is the system component responsible for implementing the virtual machine instruction set. It executes Dalvik bytecode through a combination of interpretation, ahead-of-time (AOT) compilation, and just-in-time (JIT) compilation. Instead of initializing a fresh runtime instance at every application launch, application processes are forked from a special process called Zygote that already pre-initialized the most frequently used framework classes. A list of these classes may be obtained from the publicly-readable file at `/system/etc/preloaded-classes` on a device.

3. A2P2 Overview

In this section, we describe the core goals and concepts of the A2P2 system from a high-level perspective.

Most fundamentally, A2P2 seeks to keep the entry barrier for implementing common patch use cases low while still offering the maximum degree of flexibility for advanced patching needs. For maintaining a low entry barrier, we propose a custom declarative patch format and all tooling required for developing and deploying patches to compiled APK files. We accomplish flexibility by structuring the deployment functionality as a fully extensible patching pipeline that offers the framework and primitives for efficiently implementing arbitrarily complex APK transformations. Figure 9.1 illustrates how a patch in the A2P2 format can be deployed by configuring and executing a suitable A2P2 pipeline instance.

3.1. Declarative Patch Format

Our declarative patch format is designed for specifying changes in a generic form once and deploying them to an arbitrary number of application packages later. We accomplish this by focusing supported changes on the interface between the platform and application. Fundamentally, this entails the Android manifest, where an app registers its key components to the system and declares the permissions it requires for its operation. Our

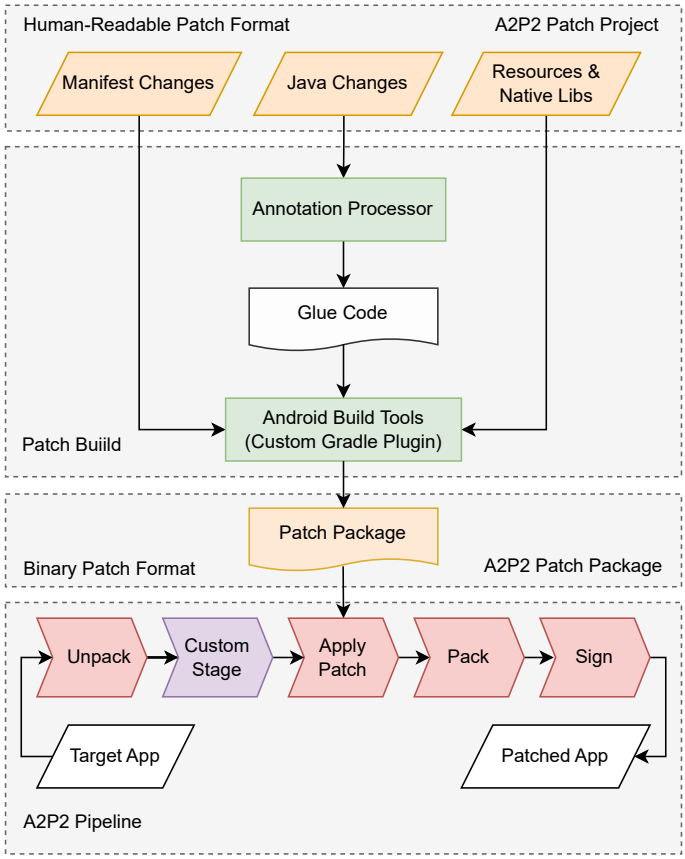


Figure 9.1: Core components of the A2P2 system: Patches in the A2P2 patch format can be deployed by configuring and executing a pipeline instance that uses the Apply Patch stage.

custom XML patch format supports specifying arbitrary modifications to manifest contents, including addition, deletion, and modification of element or attribute nodes. Second, the A2P2 patch format also allows addressing the APIs that form the functional interface between the OS and an application. An annotation-based domain-specific language enables precisely intercepting method invocations to system framework or third-party library APIs. Replacement methods are implemented in Java, may manipulate or inspect passed parameters and invoke the original method implementation. Furthermore, patches may specify resources for addition into target APK files' resource index in a way that does not collide with existing resource identifiers while still allowing to reference them in the

patch project's code or application manifest. Lastly, our patch format also allows provisioning assets or native libraries for addition into target APKs. Before a patch project can be used in the deployment pipeline, it is transformed into a patch package, which contains machine-readable representations of all changes that already resemble the respective target structures inside APK files.

3.2. Development Tooling & Patching Pipeline

A2P2 includes all tooling required for building patch packages, as well as for deploying them into application package (APK) files.

For lowering the entry barrier to A2P2, our *development tooling* integrates into the Android Studio IDE so that patch projects can be constructed in an environment Android researchers and practitioners are already familiar with. To this end, A2P2 provides a custom Gradle plugin that can be used for organizing the structure and build of a patch project. Transforming a patch project into a patch package takes advantage of the official Android build tools normally used for app development. A custom annotation processor integrates into the build process to generate glue code that later facilitates patch deployment.

Deployment tooling consists of a flexible pipeline design. As part of this design, we specify a chainable pipeline stage interface. Every stage implements a transformation on the currently processed target application package, optionally incorporating additional configuration or input files. An arbitrary number of pipeline stages and respective configuration or input files can be arranged into a sequence, forming a pipeline instance. A body of application packages (APK files) can then be filled into the pipeline and pass through the stages one by one. After undergoing the sequence of transformations implemented by the configured pipeline stages, each input package exits the last stage as an APK file again.

Our pipeline design and interface were kept generic so that users can implement and supply custom stages for integration into pipeline instances together with readily made stages shipped with our A2P2 reference implementation. Among these pre-built stages, we provide essential functionality for unpacking, packing, and signing application packages, as well as an Apply Patch stage for applying patch packages in our declarative patch format.

The Apply Patch pipeline stage takes advantage of primitives for parsing and manipulating AXML, ARSC, and DEX data structures, as well as our respective patch formats. For deploying code changes, A2P2 supports two rewriting backends that differ in compatibility characteristics.

4. Patch Format

This section describes our declarative patch format, which comprises the human-readable source variant we term *A2P2 Patch Project*, as well as *A2P2 Patch Package*, a binary variant that facilitates deployment.

The A2P2 patch format was designed to be application-agnostic yet easy to use. To this end, our patch project and patch package structures as closely as possible follow those of application projects and application packages, respectively. Changes to a given application component (e.g., the application manifest, code, or resources) are specified in the same general file format as their respective target. For example, changes to the application manifest XML file are themselves specified inside an XML file. In the following, we describe how patch projects can specify changes to the different components of an Android application package (APK file).

4.1. Application Manifest Changes

Changes to the application manifest are specified in the `AndroidManifest.xml` file of the patch project, which we term manifest patch file. While the manifest patch file would pass validation as an application manifest, it contains custom XML elements that specify a sequence of transformations to be applied to a target APK's application manifest file during patch deployment. The available XML elements are loosely based on RFC5261 [15] but adapted for the Android-specific AXML format and enhanced by XPath placeholders in added or replaced values.

Every transformation element encodes a combination of a selector and an operation. The selector is an XPath expression that specifies the precise subject or subjects (one or multiple element, attribute or text nodes) of the modification. Available operations are addition, replacement, or insertion of elements or attributes. Listing 4.1 showcases an example manifest patch file taking advantage of all supported transformation elements.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
↪  xmlns:android="http://schemas.android.com/apk/res/android"
↪  xmlns:patch="http://schemas.android.com/apk/res-auto"
   package="com.a2p2.sample.patch">

  <patch:add sel="manifest">
    <metadata android:name="patched" android:value="true" />
  </patch:add>

  <patch:replace sel="manifest/application/activity/@name">
    $${globalize(xpath("/manifest/@package"), xpath("."))}
  </patch:replace>

  <patch:remove sel="manifest/application/@testOnly"/>
</manifest>
```

Listing 4.1: Example manifest patch file

4.1.1. Add Element

Addition of new elements or attributes can be accomplished using the `<add sel="..." pos="..."/>` element in the manifest patch file. The `sel` attribute contains the selector expression, while the optional `pos` attribute encodes where exactly in relation to the selected node or its child nodes the addition should be applied. Possible values are `prepend`, `before`, and `after`. All child elements of the `add` node in the patch manifest file are added as children to target nodes. If no child elements are provided, all attributes of the `add` node are copied to selected target nodes.

4.1.2. Replace Element

The `<replace sel="..."/>` element allows replacing elements or attribute values. If its selector in the `sel` attribute targets an element, the latter will be replaced with the first child node of the `replace` element, including all attributes and children. If the XPath selector targets attributes, the

replacement value is to be provided as the text content of the `replace` element.

4.1.3. Remove Element

Lastly, elements or attributes may be removed by utilizing the `<remove sel="..."/>` element.

4.1.4. Placeholder Expressions

Added values may include placeholder expressions. These are string tokens prefixed with `$$` and surrounded by curly braces. The `$${xpath(...)}` placeholder allows including results of XPath expressions evaluated in relation to the replaced element. The `$${globalize(...,...)}` placeholder expression combines a relative class name (first parameter) and its package name (second parameter) into a fully-qualified class name. `$${appendeach(...,...)}` appends the string passed as its second parameter to every element in the semicolon-delimited list of strings in the first parameter. Placeholder expressions may be nested, enabling very complex combined expressions.

4.2. Java Execution Flow Changes

Fundamentally, A2P2's execution flow manipulation operates by intercepting method calls. Since this scheme is particularly suitable for manipulating the interaction between an application and Android framework APIs, it caters to our philosophy of application-agnostic patching. Only static methods, instance methods, and constructors that are publicly visible may be intercepted.

For every target method intercepted, patch developers implement an interception handler. Inside a patched application, this interception handler will be called in place of the target method. In a patch project, interception handlers are implemented as static Java methods that accept the same arguments, return the same type and throw the same exceptions as the target method. Every interception handler needs to be marked with an annotation that encodes the target class and method it wishes to intercept. For convenience, multiple interception handlers for methods of the same target class may be combined into one Java class, in which case the

target class annotation may be used at class level and is inherited by all contained interception handlers. Listing 4.2 takes advantage of this feature and demonstrates all possible interception annotations.

In addition to interception handler implementations, patches may contain arbitrary Java or Kotlin code. This is necessary for use cases where patches need to augment existing app functionality, e.g. by adding core components such as activities or services. These can be injected into the application manifest through the manifest patch file and backed by an implementation in the patch's code. All code is compiled into DEX files during patch build, which are to be copied into target application packages during patch deployment.

Whenever an application wishes to call the original implementation of an intercepted method, it can take advantage of the `OriginalMethods` class generated by A2P2's build tools. This class exposes a function for every intercepted method. How exactly these exposed functions implement the invocation of original method implementations depends on the employed rewriting backend (see Section 5.3). This abstraction enables adding arbitrary rewriting backends without modifying patch projects.

4.2.1. Intercepting Static Methods

For intercepting a static method, the corresponding interception handler needs to be marked with the `@PatchStaticMethod` annotation. The name of the target method is either assumed to be identical of the interception handler or passed explicitly as an argument to the annotation. Interception handlers for static methods may call the original implementation of the target method through the `OriginalMethods` abstraction.

4.2.2. Intercepting Constructors

The `@PatchConstructor` annotation encodes interception handlers for constructor invocations. Since Java identifies constructors solely by their signature, no additional name parameter or naming convention is necessary. Due to specifics of the Dalvik bytecode format and ART runtime which are further discussed in Section 5.3.1, constructor interception handlers do not have a means to explicitly call the original constructor implementation.

4.2.3. Intercepting Instance Methods

Interception handlers for instance methods may be marked with the `@PatchInstanceMethod` annotation. The same name rules and original method invocation mechanism apply as for static method interception.

A particularity of instance methods in the Dalvik bytecode format is that they carry the instance object as an implicit first argument. In the Java and Kotlin programming languages, this first argument is accessible by using the `this` keyword. Since that keyword may only be used in instance methods, static interception handlers for instance methods take the instance object as their explicit first argument.Code inside interception handlers may use this instance object for accessing member variables, or pass it to the `OriginalMethods` abstraction for invoking original method implementations.

4.2.4. Patch Inference

An important detail about the semantics of instance method interception requires further discussion. In many Android framework APIs, only the abstract type of objects passed from the system to the application is known. This affects arguments that the framework passes when invoking an application's lifecycle or callback methods, as well as results returned from framework methods.

For example, consider the `Activity.getContext()` method. Its return value is only guaranteed to extend from the abstract `Context` class. The exact concrete type of the returned object is not known before runtime and may in fact differ between applications. This potentially leads to problems if a patch wishes to intercept an instance method of the `Context` class. In most cases where the exact instantiation type of abstract framework types and interfaces is not known, it is desirable to intercept calls to all corresponding implementations in subclasses. Consider a patch that wishes to intercept calls to the abstract `Context.getSystemService()` method. In the absence of knowledge about the possible concrete implementations of this class (in the system framework or in the code of target applications), the most sensible approach for the patch is to intercept all implementations in all subclasses of the `Context` class.

This consideration plays an important role in the semantics of our execution flow patches. For maintaining application-agnosticity, we require

```
@PatchClass("java.net.Socket")
public class SocketSamplePatch {
  // Intercept Socket creation
  @PatchConstructor
  public static void init(Socket thiz, InetAddress address, int
  ↪  port) throws IOException {
    try {
      thiz.setSendBufferSize(4096);
    } catch (SocketException ignored) {}
  }
  // Intercept calls to Socket.setSocketImplFactory
  @PatchStaticMethod("setSocketImplFactory")
  public static void interceptionHandler(SocketImplFactory fac)
  ↪  throws IOException {
  }
  // Intercept calls to instance method of same name
  @PatchInstanceMethod
  public static boolean isConnected(Socket thiz) {
    return OriginalMethods.java_net_Socket.isConnected(thiz) ||
    ↪  true;
  }
}
```

Listing 4.2: Example execution flow patch

all instance method patches to target those classes in the inheritance tree that first define the target method. Our build and deployment tools take care of enforcing this requirement and inferring patches to all possible subclasses both in framework and application code. These semantics do not have any practical consequence on the expressiveness of our Java patch format. Interception handlers that wish to only target specific subclass implementations may simply check the runtime type of the instance object and use the `OriginalMethods` interface for invoking the original implementation for all types they are not interested in.

4.3. Adding Resources

Resources in the patch project will be copied into any target application package the compiled patch package is applied to. Different resource

types and configurations are fully supported. Cross-references between resources, as well as resource references in patch code (interception handler or otherwise) or the manifest patch file, may be used in patch projects and remain functional through patch deployment. x

4.3.1. Avoiding resource identifier conflicts

As described in Section 2, an index of all resources in an Android application project is created during compilation and stored in the ARSC file inside the APK package. As part of the indexing process, the aapt2 build tool assigns a unique integer identifier to each resource. Resources of the same type receive consecutive identifiers. For cross-referencing, Android's build tools automatically generate a Java class that maps between a human-readable resource file name and its integer identifier. The mapping is implemented by exposing a final static integer field for each resource, named after its file name and hardcoded to its integer identifier. Developers may thus conveniently reference resource identifiers through their corresponding fields in the generated Java class. During compilation, all final static integer references are inlined, which means that the convenience added by this approach does not have any runtime costs. Since inlined resource identifiers inside Dalvik bytecode can not reliably be distinguished from arbitrary hardcoded integer values, this scheme impedes changing resource identifiers in compiled application packages. However, given the predictable nature of resource identifiers and the fact that the ARSC index for patch packages is assembled by the standard aapt2 tool, identifier conflicts between resources in the application package and the patch package are to be expected.

To avoid this problem, the A2P2 patch package format takes advantage of a namespacing mechanism Android normally uses to distinguish between application and system resources . We alter the ARSC data structure inside patch packages to use a package identifier (`0x8f`) that differs from that used in applications (`0x7f`). Since the package identifier constitutes the higher-most byte of every resource identifier, this effectively prevents collisions.

4.4. Additional assets and native libraries

Patch packages may contain additional asset files or native libraries. During patch deployment, these are copied to locations in the target application

package that correspond to their locations in the patch package. It is worth noting that files in the patch package overwrite equally named files in the application package. If replacement is not desired, patch developers must take care to choose names and locations that are unlikely to occur in application packages.

5. A2P2 Development and Deployment Tools

In this section, we discuss the tools we implemented to support our A2P2 patch design. The A2P2 distribution comprises all software required for developing patch projects and compiling them into patch packages, as well as for applying patch packages to application packages.

5.1. Development Tools

A2P2 patch projects are developed inside the Android Studio IDE. The software components required for adapting the IDE to our purposes are implemented in a custom plugin for the Gradle build system and an annotation processor.

5.1.1. A2P2 Gradle Plugin

Since the A2P2 patch project and patch package formats closely resemble the respective Android application formats, our Gradle plugin extends the standard Android Gradle plugin. It integrates our annotation processor into the build process, configures the non-standard aapt2 resource namespace, and exposes new build tasks for the patch package. Furthermore, it automatically adds dependencies to our patch base libraries, which contain definitions for patch annotations and parts of the rewriting backend logic.

5.1.2. Annotation Processor

The A2P2 annotation processor (AP) holds a key role in the build process of an A2P2 patch project. It validates execution flow patches, implements parts of the inference logic for execution flow patches, and generates glue code for different rewriting backends.

For validating execution flow patches, the AP parses the classpath dependencies of the patch project, in particular the `android.jar` file included in the Android SDK distribution. This file contains stub implementations of all classes and interfaces in public Android APIs. The AP looks up the target methods of interception handlers in the classpath and compares the declared parameters, return types, and thrown exceptions. If a specified target method could not be found or a mismatch in the signature is detected, the AP aborts the build. It thus helps prevent bugs in patches that would otherwise later manifest in obscure runtime issues after patch deployment.

Inference of execution flow patches in the AP works by constructing a class inheritance hierarchy from the patch project's classpath dependencies. Using this information, the AP generates additional annotations for consumption during patch deployment.

The glue code generated by the AP is needed for abstracting away the implementation details of different rewriting backends, as well as for facilitating later patch deployment. Generated code parts constitute wrappers around interception handlers, as well as implementations of the `OriginalMethods` class for invoking original method implementations.

5.2. Deployment Tools

A2P2's deployment functionality is organized in a flexible transformation pipeline. Our implementation in the Java programming language may be used as a standalone command-line tool or integrated into other projects as a Java library. The logic for applying our declarative patch format is implemented in a pipeline stage. Other provided stages implement functionality for unpacking, packing, and signing application packages, or adding asset files. All stages follow a generic pipeline interface that facilitates the development of custom stages as well as extending existing stages.

The general procedure including pipeline instantiation and execution is depicted in Figure 9.2. We call the entirety of pre-installed and custom pipeline stages the stage repository. A pipeline instance is assembled as a parameterized sequence of stages from this stage repository. The pipeline instance is then executed on a concrete set of APK files. Every stage operates on a pipeline context, which represents the output of the previous

```
$java -jar a2p2.jar file1.apk ./folder_of_APKs ! unpack ! apply
↪  patch_static.zip static ! custom_stage ! pack ! sign !
↪  ./output
```

Listing 5.1: Example A2P2 pipeline invocation

stage in the execution. The result of a successful execution of the pipeline instance is a set of transformed output APK files.

5.2.1. Pipeline Implementation

The command line interface to our pipeline implementation allows configuring pipeline instances and executing them on a specific body of applications in one invocation. An example invocation is shown in Listing 5.1. Individual stages are specified by their name and separated using an exclamation mark character. Additional space-delimited arguments may be passed to each stage. A list of all known stages (the stage repository) and their supported arguments may be obtained by invoking the command line interface without any parameters. Custom stages may be added to the stage repository as jar files installed to a specific subdirectory of the A2P2 pipeline distribution.

5.2.2. Apply Patch Stage

The logic required for applying a patch package to an application package is implemented in the Apply Patch stage. Besides the mandatory path to a patch package, the user may optionally specify the rewriting backend. By default, static rewriting is employed.

5.2.3. Custom Stages

Pipeline instances may include custom stages implemented by third parties. Every stage must extend from our abstract `Stage` class, which provisions functionality for querying stage metadata (such as its name), configuring a concrete stage instance, and executing its transformation on a pipeline context. Custom stages may take advantage of A2P2's utility classes

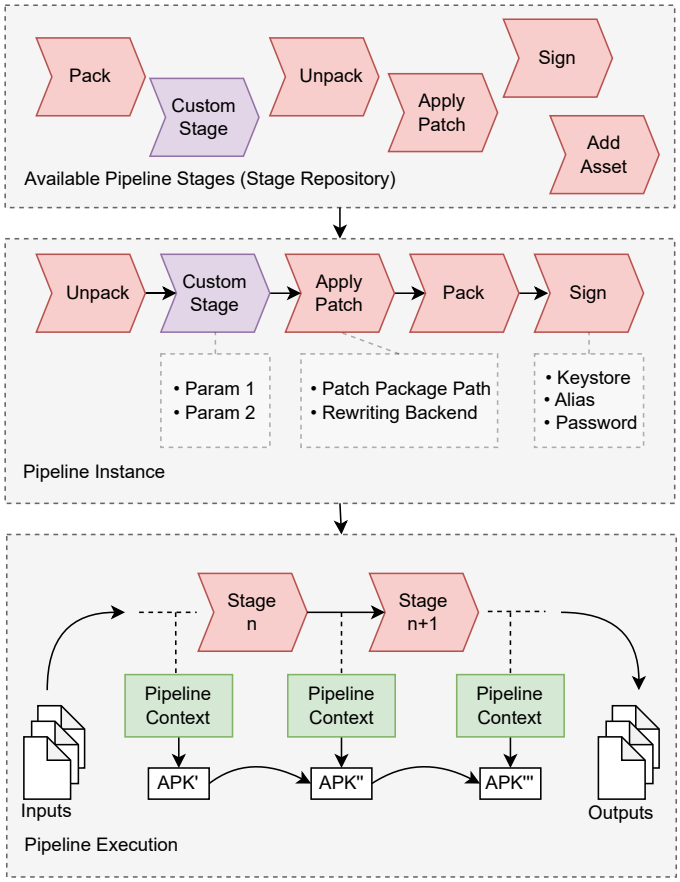


Figure 9.2: Pipeline instantiation and execution

for replacing or inserting instructions in method implementations inside Dalvik bytecode, manipulating binary AXML files, or altering ARSC files.

5.3. Rewriting Backends

Our build and deployment infrastructure supports two different rewriting backends for applying execution flow patches to DEX files. Static rewriting bakes all changes directly into DEX files, which means that the runtime overhead of interception is minimal, and the resulting patched application package is fully compatible with any device that supported the original APK. Dynamic rewriting injects a native library into application packages that applies code manipulations at runtime. In contrast to static rewriting,

this approach supports intercepting invocations through Reflection or JNI but relies on implementation details of the ART runtime that differ between devices, CPU architectures and ART versions.

A2P2 patch projects may be developed independently of the rewriting backend that is later used for deployment. Still, compiled patch packages in our current implementation are specific to a rewriting backend. During patch deployment, the user has to ensure the supplied patch package is coherent with the specified rewriting backend. The A2P2 build tools and deployment pipeline were designed so that new rewriting backends may be incorporated without changing any code in patch projects.

5.3.1. Static Rewriting

The static rewriting implementation scans Dalvik bytecode for invocation instructions, replacing references to methods targeted by execution flow patches with references to the corresponding interception handler.

For constructor interception, the call to the interception handler is injected as an additional instruction that does not replace the call to the constructor but immediately follows it. This is necessary because the Dalvik bytecode specification does not permit passing an allocated object to another method before its constructor has been called. Since the added interception handler invocation changes the addresses of subsequent instructions, our rewriting backend updates all offsets and ranges in affected try/catch or switch/case blocks, goto or jump instructions, and debug items.

Patch inference for classes implemented in application code works by scanning the app's DEX files for subclasses of all classes targeted by interception handlers. It is worth pointing out that super calls (invoking the super class's implementation of the currently executed method) are ignored in static rewriting. This is because the corresponding `invoke-super` instruction may only be used inside of class context, which means that the originally called method would not have been accessible to interception handlers. In practice, since our patch format is designed to focus on the interface between application and framework code, this limitation doesn't have any negative consequences on patch developers. Our static rewriting implementation builds upon low-level DEX manipulation primitives of the open-source dexlib2[2] library.

[2] dexlib2: `https://github.com/JesusFreke/smali/tree/master/dexlib2`

5.3.2. Dynamic Rewriting

Dynamic rewriting applies execution flow changes at runtime. Through a high-priority `ContentProvider` injected into target application packages, we can manipulate the ART runtime before any third-party code is executed. The `ContentProvider` parses annotation data from the patch DEX file to assemble a list of methods that need to be intercepted. It then iterates over all classes preloaded by the Zygote process to intercept all targeted methods they implement. Lastly, we instrument the ART runtime's class loading mechanism to intercept targeted methods for all classes as they are loaded. This also provides an opportunity for inferring patches onto subclasses.

Method interception is implemented by manipulating the JIT-compiled in-memory representation of target methods. Their preamble is replaced by a trampoline to the interception handler. The original preamble is copied to a freshly allocated executable memory region, followed by a jump to the end of its original location, and executed whenever the original method implementation is invoked through the `OriginalMethods` interface.

Because of patch inference, an interception handler for an instance method may be invoked for various different concrete overridden implementations of its target method. This leads to potential issues in the `OriginalMethods` implementations for dynamic rewriting, where the exact originally referenced method needs to be known for being able to call it. Due to potentially involved super calls (which cannot be distinguished from normal method calls), the object type alone does not suffice for identifying the originally called method. As a solution to this problem, our dynamic rewriting backend takes advantage of internal mechanisms of the `java.lang.reflect.Proxy` infrastructure for dynamically generating a distinct trampoline method for every infered method interception. These trampoline methods act as breadcrumbs in the call stack, which we combine with a lookup table for determining the originally called method in `OriginalMethods` implementations.

Our dynamic rewriting implementation integrates modified low-level primitives of the open-source SandHook[3] library.

[3]SandHook Android ART Hook: `https://github.com/asLody/SandHook`

6. Evaluation

In this section, we provide patch case studies that demonstrate the efficacy of A2P2 for facilitating mobile security research, as well as performance and overhead metrics for evaluating the efficiency of our solution.

6.1. Patch Case Studies

The patches we showcase here were chosen to demonstrate the capabilities of our patch and pipeline design and reflect typical use cases in mobile security research.

6.1.1. Grab'n'Run

We used A2P2 to reimplement the purpose-built application-rewriting tool proposed by Falsina et al. [9] for injecting their novel verification protocol for dynamic code loading into compiled applications. Falsina et al.'s implementation involves almost 1000 lines of Python code for manually manipulating the SMALI IR representation of decompiled Dalvik bytecode and adding permissions to the application manifest. In contrast, the A2P2 patch can be fully implemented using our declarative patch format, which means that all changes are specified in high-level Java and XML. The key part of the Java execution flow patch can be found in Listing 10.1 in the Appendix. It intercepts class loading through the `DexClassLoader` class to verify the code that is about to be loaded. Note how the patch intercepts the constructor to create shadow objects that are later looked up in the interception handler for the instance method. This pattern effectively allows replacing object types. The `loadClass()` interception handler also displays how changes to the execution flow may be applied selectively based on the concrete instance type. While our patch implementation uses the default A2P2 pipeline command line interface for deployment, a more convenient custom Java program could easily be written that integrates the pipeline as a library.

6.1.2. Cloning applications

In some scenarios, an app needs to be installed on a device twice, e.g. when it comes preinstalled to the device and thus cannot simply be uninstalled

for replacement with a patched version. The Android OS requires the application package name and certain app components to be unique on the system, so that the same APK file cannot be installed twice. While for simple applications, changing the package name in the application manifest is enough to create an APK file that can be installed alongside the original version, more modifications are needed in general.

We implemented an A2P2 patch project that changes the package name, custom permissions, and `ContentProvider` authorities in the application manifest. A part of the patch manifest of this project can be found in Listing 10.2 in the Appendix. It shows how A2P2's XML patch format and XPath expressions may be used for changing the package name and various related values inside an app's manifest. Note how the actual package name change is the last entry in the patch manifest. Since patch manifest entries are applied sequentially, this allows earlier entries to reference the original package name. This is e.g. used for globalising class name references that were constructed relative to the package name in the original manifest. The Java part of the patch project intercepts various framework API calls so that the original package name is spoofed to all application-facing code while system- and world-facing code is aware of the manipulated package name. This even entails package names sent to backend servers in HTTP request headers. We complemented our patch project with a custom pipeline stage that also replaces the account type for app-specific authenticators, which are required to be system-unique as well.

6.1.3. Injecting Flipper debugger

Flipper is an open-source mobile application debugging platform maintained by Facebook. Developers may integrate plugins into their Android application project that communicate with the Flipper desktop companion program. Flipper offers plugins for inspecting files in app-private storage, databases, UI layouts, or network communication. While originally intended for debugging during application development, the platform may also serve as a security analysis tool.

We designed an A2P2 patch that injects the base Flipper library and several plugins into target applications. A `ContentProvider` added through the manifest patch initializes plugins at runtime. Several execution flow patches are used for integrating the networking plugin into the OkHttp stack within target applications.

6.2. Performance and Overheads

Quantifiable performance and overhead metrics include the patch deployment duration, as well as the size and runtime overhead incurred by applying a patch to an application package. It is worth noting that all metrics shown reflect a pipeline configuration for applying a patch package, utilizing the built-in Unpack, Apply Patch, Pack and Sign stages.

6.2.1. Patch Deployment Speed

Patch deployment speed largely depends on the sizes of patch and application packages. As a realistic estimate, we provide measurements for applying our relatively large Flipper patch (see 6.1.3) to the APK file of the Wikipedia Android app (version 2.7.50431) on a 2.3 GHz Intel i7 quad-core CPU.

For dynamic rewriting, patch deployment consists of unzipping the APK file, adding a `ContentProvider` to the application manifest, merging patch resources, extracting the rest of the patch package (including native libraries and DEX files) into the target application package, zipping the resulting APK file and signing it again. For our test setup, this process takes about 12 seconds on average. Static rewriting additionally involves parsing interception handler targets from the DEX file in the patch package and iterating over all instructions in the target application's DEX files. In our test setup, these additional steps resulted in an overall patch deployment time of 21 seconds for static rewriting.

6.2.2. Application Size Overhead

The file size overhead of application packages caused by our patching very closely follows the size of the applied patch package. All patch packages contain support files that amount to about 950 KB for dynamic rewriting and 10 KB for static rewriting. Since the size of patch packages otherwise depends on the contents of the corresponding patch projects, we cannot provide any more specific metrics here.

	Constructor	Static M.	Instance M.
Static Rewriting	77 ns	23 ns	20 ns
Dynamic Rewriting	25733 ns	1779 ns	43499 ns

Table 9.1: Per-call runtime overhead for different rewriting backends and method types.

	Constructor	Static M.	Instance M.
Dynamic Rewriting	1.46 ms	0.46 ms	0.86 ms

Table 9.2: Per-method runtime overhead for dynamic rewriting at app launch (no launch overhead for static rewriting).

6.2.3. Application Runtime Overhead

As an indicator for the runtime impact of execution flow patches, we timed the overhead per call to an intercepted method. Additionally, we timed the ART manipulations carried out for every intercepted method at app launch when dynamic rewriting is used.

For all measurements, we used a simple patch that intercepts a constructor, static, and instance method. The interception handlers simply invoke the corresponding original method implementations. We then applied the patch on an application that times the execution of intercepted methods over 100000 runs. Per-call overheads were taken as the average over these runs. Per-method (launch time) overheads were taken as the average time it took to carry out the runtime manipulations required for intercepting one method (or constructor) over 10 app launches.

The per-call overhead induced by static rewriting is almost non-existent (77 ns at maximum). Dynamic rewriting generates a higher overhead (we measured 43499 ns in the worst case) due to the involved Java Native Interface calls and original implementation lookup. Still, at less than a tenth of a millisecond, this overhead can still be considered negligible in most practical scenarios. Complete measurement data can be found in Table 9.1.

Static rewriting does not have any per-method overhead at app launch time. For dynamic rewriting, we measured a worst-case overhead (for constructors) of 1.45 milliseconds. Exact data can be found in Table 9.2.

6.2.4. Application Compatibility

As a measure for the general compatibility of our patching process, we applied the Cloning patch described in Section 6.1.2 to the 132 most popular free applications (4 per category) from Google Play and confirmed that the patched APK could still be successfully installed and launched on a Pixel 3 running Android 11 (Build RQ3A.211001.001). Our results show that for static rewriting, 92 % of applications still launched, while the success rate of dynamic rewriting was 91 %. Patching generated an installable APK file for all applications. Dynamic rewriting caused one app to hang during launch. Other incompatibilities with individual applications that lead to runtime issues across rewriting backends could be attributed to signature checks (3 apps) and side effects of the package name manipulations of the specific patch implementation (9 apps).

7. Discussion & Future Work

This section discusses known limitations in A2P2's design, as well as possible improvements to the current implementation.

7.1. Resigning

Some applications integrate signature checks designed for preventing malicious repackaging attacks. Unfortunately, these checks also impact application patching for security research purposes. Still, Ibrahim et al. [11] have recently shown that less than 0.1 % of apps employ out-of-process app attestation. All other in-process signature check schemes may be bypassed through application patching, and we argue this is ethically viable for security research. We note that A2P2's execution flow patching may be used for bypassing the most common signature check implementations.

7.2. Malicious Patching

It is worth noting that although we designed A2P2 as a security tool, it may be misused for applying malicious modifications to compiled application packages, i.e. for mounting repackaging attacks. These attacks may e.g. use A2P2 for creating a version of a paid application that bypasses

license checks and/or adds code for collecting sensitive user information. However, these types of attacks have been possible before, so we argue that A2P2's publication does not lead to any additional harm on end users. We hope that the availability of Android application patching tools in general encourages more developers of sensitive applications to adopt out-of-process app attestation, which is the only effective mitigation for repackaging.

7.3. Obfuscation

In principle, A2P2 is capable of not only intercepting framework methods but also those exposed by third-party libraries. To this end, our annotation processor can look up method definitions from any class in the classpath of the patch project. However, implementations of these libraries are embedded in application code and may thus be subjected to obfuscation, i.e. replacing class and method names with undescriptive short strings during build. This technique is commonly used by developers to thwart reverse engineering and optimize APK size. Since our execution flow patches rely on method and class names for identifying target methods, obfuscation may render them ineffective. Future work could investigate how target methods can be identified via obfuscation-resistant selectors.

7.4. Native Code

Our current patch design does not support manipulating the execution flow of native code. However, dynamic rewriting does cover calls from native code to Java through the Java Native Interface.

7.5. Device and OS compatibility

As mentioned in Section 5.3.2, dynamic rewriting relies on implementation details of the ART runtime that differ between devices, OS, and ART versions. We developed and tested our dynamic rewriting backend on a Google Pixel 3 running Android 9, 10, and 11. We anticipate slight adjustments for expanding compatibility to additional device configurations. For devices not yet supported by dynamic rewriting, patch developers may use static rewriting.

7.6. Performance

It is worth noting that our implementation can be further optimized, particularly for patch deployment performance. Possibilities for enhancements would be parallelizing the processing of individual DEX files inside the Apply Patch stage or implementing DEX, ARSC, and AXML manipulation primitives in native code.

8. Related Work

In this section, we highlight previous publications that are concerned with modifying some aspects of compiled Android application packages.

8.1. Modifying APK Files

While some solutions exist for manipulating arbitrary parts of a compiled APK file, none of them share A2P2's application-agnosticity. The most established tool for manipulating APK files both in academia and in industry is Apktool[4]. It is capable of decompiling resources and application manifests back into human-readable XML format and disassemling Dalvik bytecode into the SMALI intermediate representation. Although Apktool supports recompiling APK files after modifications to any of its parts, changes have to be manually applied and adapted to each target application.

8.2. Manipulating Dalvik bytecode

A number of publications have proposed generic solutions for modifying *the execution flow* of compiled Android applications. Some of them seek to offer a similar application-agnostic approach as A2P2's patch format.

The existing work that shares the most similarities with A2P2 with respect to execution flow patching is RetroSkeleton by Davis and Chen [6]. It allows intercepting arbitrary methods using transformation policies written in Closure. RetroSkeleton also follows a similar notion of patch inference but only supports bytecode instrumentation, which operates similarly to

[4]Apktool - A tool for reverse engineering Android apk files: `https://ibotpeaches.github.io/Apktool/`

our static rewriting. The implementation was never made available to the public. Reptor by Ki et al. [13] focuses on API virtualization, for which they go to great lengths to accomplish method implementation replacement instead of call replacement. However, their solution creates boilerplate class hierarchies that lead to considerable APK size overhead and was never made available to the public.

Some tools decompile Dalvik bytecode to Java so they can take advantage of established manipulation tools and techniques for the desktop Java platform. However, this approach is prone to introduce issues since the transformation from Java to Dalvik bytecode during app compilation is not fully reversible in general. SIF by Hao et al. [10] takes advantage of this approach for implementing their instrumentation framework that allows procedural configuration of interception targets. While their system supports complex analysis of the control flow graph, it does not have any notion of patch inference. Arzt et al. [2] and Ali-Gombe et al. [1] apply aspect-oriented programming principles from the desktop Java world to Android applications.

Beside solutions proposed by academia, Frida[5] is a popular choice for instrumenting Android applications both on rooted or unrooted devices. It injects the V8 engine into target applications, so that runtime manipulations can be expressed in JavaScript code. However, Frida operates by manipulating runtime structures. Similar to our dynamic rewriting backend, it is prone to break whenever ART implementation details change.

9. Conclusion

Inspecting and manipulating runtime behavior of Android applications is a common need in mobile security research. However, existing solutions for application patching either lack a holistic view of the application package or do not support application-agnostic operation. In this paper, we introduced the A2P2 patch format and deployment pipeline for remediating these limitations in tooling. Our patch format allows declaratively specifying changes to the application manifest and Dalvik bytecode, as well as addition of resources, native libraries, and assets. We discussed the patch semantics for different APK components, the functionality we provide for building application-agnostic patch projects, and our extensible

[5] Frida - Dynamic instrumentation toolkit: `https://frida.re`

tooling for applying them to compiled Android application packages. For demonstrating the efficiency of our design and implementation, we showed various performance characteristics of our solution. Lastly, we evaluated the efficacy of A2P2 through compatibility tests and by showcasing patches that reproduce previous research and facilitate security analysis.

References

[1] Aisha I. Ali-Gombe, Irfan Ahmed, I. I. I. Golden G. Richard, and Vassil Roussev. AspectDroid: Android App Analysis System. In: Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy (CODASPY). 2016 (p. 279).

[2] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Instrumenting Android and Java Applications as Easy as abc. In: Runtime Verification - 4th International Conference (RV). 2013 (p. 279).

[3] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged App Sandboxing for Stock Android. In: 24th USENIX Security Symposium. 2015 (p. 252).

[4] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. AppGuard - Enforcing User Requirements on Android Apps. In: Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference (TACAS). 2013 (p. 253).

[5] Valerio Costamagna and Cong Zheng. ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime. In: Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security (IMPS). 2016 (p. 252).

[6] Benjamin Davis and Hao Chen. RetroSkeleton: retrofitting android apps. In: The 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys). 2013 (p. 278).

[7] Florian Draschbacher. A2P2 - An Android Application Patching Pipeline Based On Generic Changesets. In: ARES. 2023 (p. 249).

[8] Lukas Dresel, Mykolai Protsenko, and Tilo Müller. ARTIST: The Android Runtime Instrumentation Toolkit. In: 11th International Conference on Availability, Reliability and Security, ARES 2016. 2016 (p. 252).

[9] Luca Falsina, Yanick Fratantonio, Stefano Zanero, Christopher Kruegel, Giovanni Vigna, and Federico Maggi. Grab 'n Run: Secure and Practical Dynamic Code Loading for Android Applications. In: Proceedings of the 31st Annual Computer Security Applications Conference. 2015 (pp. 253, 272).

[10] Hao Hao, Vicky Singh, and Wenliang Du. On the effectiveness of API-level access control using bytecode rewriting in Android. In: 8th ACM Symposium on Information, Computer and Communications Security (ASIA CCS). 2013 (p. 279).

[11] Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. SafetyNOT: on the usage of the SafetyNet attestation API in Android. In: MobiSys '21: The 19th Annual International Conference on Mobile Systems, Applications, and Services. 2021 (p. 276).

[12] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd D. Millstein. Dr. Android and Mr. Hide: fine-grained permissions in android applications. In: Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM). 2012 (p. 253).

[13] Taeyeon Ki, Alexander Simeonov, Bhavika Pravin Jain, Chang Min Park, Keshav Sharma, Karthik Dantu, Steven Y. Ko, and Lukasz Ziarek. Reptor: Enabling API Virtualization on Android for Platform Openness. In: Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys). 2017 (p. 279).

[14] Jierui Liu, Tianyong Wu, Xi Deng, Jun Yan, and Jian Zhang. InsDal: A safe and extensible instrumentation tool on Dalvik byte-code for Android applications. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2017 (p. 252).

[15] Jari Urpalainen. An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors. RFC 5261. Sept. 2008. URL: https://rfc-editor.org/rfc/rfc5261.txt (p. 259).

[16] Philipp Von Styp-Rekowsky, Sebastian Gerling, Michael Backes, and Christian Hammer. Idea: Callee-Site Rewriting of Sealed System Libraries. In: Engineering Secure Software and Systems - 5th International Symposium (ESSoS). 2013 (p. 252).

[17] Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In: 22nd Annual Network and Distributed System Security Symposium (NDSS). 2015 (p. 253).

10. Appendix

10.1. Patch Case Study Snippets

```
public class DexClassLoaderPatch {
  static Map<String, SecureDexClassLoader> loaders = new HashMap<>();

  @PatchClass("dalvik.system.DexClassLoader")
  @PatchConstructor
  public static void init(DexClassLoader thiz, String dexPath, String dir,
  ↪  String path, ClassLoader parent) {
    loaders.put(thiz.toString(),
    ↪  RepackHandler.generateSecureDexClassLoader(dexPath, dir, path,
    ↪  parent));
  }

  @PatchClass("java.lang.ClassLoader")
  @PatchInstanceMethod
  public static Class<?> loadClass(ClassLoader thiz, String name) throws
  ↪  ClassNotFoundException {
    if (thiz instanceof DexClassLoader) return
    ↪  OriginalMethods.java_lang_ClassLoader.loadClass(thiz, name);
    SecureDexClassLoader loader = loaders.get(thiz.toString());
    if (loader == null) return
    ↪  OriginalMethods.java_lang_ClassLoader.loadClass(thiz, name);
    Class result = OriginalMethods.java_lang_ClassLoader.loadClass(loader,
    ↪  name);
    if (result == null) RepackHandler.raiseSecurityException();
    return result;
  }
}
```

Listing 10.1: Grab'n'Run Patch: Securing class loading by intercepting `DexClassLoader` methods

```
<!-- Permission names must be unique -->
<patch:replace sel="manifest/permission/@name">
  $${xpath(".")}.patched</patch:replace>

<!-- Also adjust permission uses -->
<patch:replace sel="manifest/uses-permission[not(starts-with(@name,
↪  'android.permission')) and not(starts-with(@name,
↪  'android.gms.permission')) and not(starts-with(@name,
↪  'com.google.android.calendar'))]/@name">$${xpath(".")}.patched
</patch:replace>

<!-- Change class reference to use full original package name -->
<patch:replace sel="manifest/application/@name">
  $${globalize(xpath("/manifest/@package"), xpath("."))}
</patch:replace>

<!-- Make sure the patched app can query the original -->
<patch:add sel="manifest/queries">
  <package android:name="$${xpath(&quot;/manifest/@package&quot;)}"/>
</patch:add>

<!-- Provider authorities must be unique -->
<patch:replace sel="manifest/application/provider/@authorities">
  $${appendeach(xpath("."), ".patched")}</patch:replace>

<!-- Change package name by appending .patched -->
<patch:replace sel="manifest/@package">
  $${xpath(".")}.patched</patch:replace>
```

Listing 10.2: App Cloning Patch: Simplified manifest patch for making a patched app installable alongside the original

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.