University of Brasília

Institute of Exact Sciences
Department of Computer Science

Assessing the Resilience of Popular Android Apps Against Repackaging Using Controlled Variants and Instrumentation

Leandro de Souza Oliveira

Dissertation submitted in partial fullfilment of
the requirements for the Master's Degree in Informatics

Advisor
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2025

University of Brasília

Institute of Exact Sciences
Department of Computer Science

Assessing the Resilience of Popular Android Apps Against Repackaging Using Controlled Variants and Instrumentation

Leandro de Souza Oliveira

Dissertation submitted in partial fullfilment of
the requirements for the Master's Degree in Informatics

Prof. Dr. Rodrigo Bonifácio de Almeida (Advisor)
University of Brasília (UnB)

Prof. Dr. Eduardo James Pereira Souto
Federal University of Amazonas (UFAM)

Prof. Dr. Rui Rua
New York University of Abu Dhabi (NYU)

Prof. Dr. Cláudia Nalon
Coordinator of the Postgraduate Program in Informatics

Brasília, Setembro 08, 2025

Dedication

I dedicate this thesis to Brazilian software engineering researchers who continue to pursue their work with determination despite the many challenges in obtaining adequate resources for their experiments.

Acknowledgements

I would like to express my sincere gratitude to my advisor, Dr. Rodrigo Bonifácio de Almeida, for his openness to my ideas, his guidance, and his constant support throughout this journey. His efforts in fostering connections with other researchers greatly enriched this work.

I am deeply thankful to my wife, Janaina Silva de Oliveira, for her unwavering support and for the valuable insights she offered during the development of this thesis.

I also acknowledge the Computer Forensics Section of the Criminalistics Institute of the Civil Police of the Federal District for their trust and support.

Finally, I extend my thanks to all who contributed, directly or indirectly, to the completion of this research.

Avaliando a Resiliência de Aplicativos Android Populares contra Reempacotamento por meio de Variantes Controladas e Instrumentação

Resumo

Contexto: O reempacotamento (ou *repackaging*) é uma técnica utilizada por atacantes que consiste em obter aplicativos legítimos, descompilá-los, fazer modificações (geralmente inserindo funcionalidades maliciosas) e redistribuí-los como se fossem os aplicativos originais. Esta prática representa uma ameaça significativa à segurança de aplicativos Android, com potencial para impactar a receita de desenvolvedores e expor usuários a softwares maliciosos. No entanto, o reempacotamento também é amplamente utilizado em técnicas de instrumentação que incorporam mecanismos de monitoramento aos aplicativos, viabilizando pesquisas de segurança sobre seu comportamento. A taxa de sucesso de técnicas de instrumentação varia consideravelmente e explicações genéricas para falhas deixam em aberto a seguinte questão: até que ponto aplicativos populares são suscetíveis ao reempacotamento?

Objetivo: O objetivo deste trabalho é identificar e analisar os fatores que levam ao fracasso em técnicas que utilizam reempacotamento como etapa do processo de instrumentação.

Estrutura da Pesquisa: O presente estudo propõe uma abordagem sistemática para a compreensão do problema baseada em uma infraestrutura específica capaz de automatizar e monitorar o processo de reempacotamento. Essa infraestrutura foi desenvolvida e aplicada a um conjunto curado de aplicativos da Google Play Store e, durante a execução dos experimentos, foram coletados e analisados dados sobre o comportamento desses aplicativos frente às diferentes etapas do processo de reempacotamento, oferecendo subsídios para a identificação de defesas e falhas de instrumentação.

Resultados: Entre os resultados obtidos, apresentamos o InstruMate, uma abordagem sistemática capaz de gerar aplicativos reempacotadas. Essa infraestrutura pode ser utilizada para avaliar a resiliência de aplicativos ao reempacotamento, uma vez que realiza modificações em artefatos sensíveis e utiliza um procedimento de verificação que compara detalhadamente o comportamento de cada variante com sua versão original. Em um conjunto curado de 156 aplicativos populares da Google Play Store, nosso estudo empírico revelou que 86% não são resilientes ao reempacotamento básico (por exemplo, alteração simples de assinatura), 81% permitem execução em modo de depuração, 83%

toleram modificações superficiais e 65% são confirmadamente suscetíveis à adulteração de código. De forma geral, nossas contribuições incluem a infraestrutura para reempacotamento e um procedimento aprimorado de verificação de variantes reempacotadas.

Conclusões: Os resultados obtidos evidenciam tanto a facilidade quanto os riscos associados ao reempacotamento de aplicativos Android, além de validarem o InstruMate como uma infraestrutura eficaz para avaliar a resiliência de apps frente a esse tipo de alteração. Analistas de segurança podem se beneficiar do teste de repackaging proposto, já que as técnicas de instrumentação utilizadas têm como objetivo final simplesmente introduzir um método de rastreio do código e proporcionar relatórios de cobertura. Caso um aplicativo se mostre suscetível a essas abordagens, é provável que técnicas mais avançadas também obtenham êxito.

Palavras-chave: Atestação de software, endurecimento (*hardening*) binário, engenharia reversa, Android, reempacotamento, proteção contra reempacotamento.

Resumo Expandido

Avaliando a Resiliência de Aplicativos Android Populares contra o Reempacotamento por meio de Variantes Controladas e Instrumentação

Introdução

O reempacotamento (ou *repackaging*) é uma técnica utilizada por atacantes que consiste em obter aplicativos legítimos, descompilá-los, fazer modificações (geralmente inserindo funcionalidades maliciosas) e redistribuí-los como se fossem os aplicativos originais. Esta prática representa uma ameaça significativa à segurança de aplicativos Android, com potencial para impactar a receita de desenvolvedores e expor usuários a softwares maliciosos. Embora as lojas de aplicativos implementem ativamente estratégias para detectar e remover aplicativos reempacotados, o problema persiste, com a disseminação de ferramentas automatizadas de reempacotamento como *App Cloner Premium* [1] e *ModHeaven* [2], que alcançam milhões de *downloads* na Google Play Store. Curiosamente, o reempacotamento também desempenha um papel legítimo na pesquisa em segurança, ao possibilitar a instrumentação de aplicativos para monitoramento do uso de APIs sensíveis, detecção de *bugs*, entre outras funcionalidades. Estudos anteriores relatam taxas de sucesso variáveis na geração de aplicativos reempacotados — com média em torno de 56% — e tendem a focar nos resultados obtidos, sem explicar por que o reempacotamento falha em certos casos. Essas falhas são geralmente atribuídas à complexidade do processo, porém, causas mais específicas — como mecanismos de defesa contra reempacotamento embutidos nos aplicativos — raramente são mencionadas. Essa ausência de uma análise detalhada sobre as falhas limita o escopo dos estudos e pode impactar negativamente a sua aplicação prática em casos reais.

A presente pesquisa tem como objetivo avaliar como os aplicativos Android populares reagem quando são reempacotados e explorar as razões por trás de falhas que ocorrem nesse processo. Ela é guiada por três perguntas de pesquisa: quão suscetíveis a *repackaging* são os aplicativos populares Android (RQ1), quais são os mecanismos de defesa comumente

utilizados pelos aplicativos que não podem ser reempacotados (RQ2) e quais são as causas fundamentais que levam a falhas de reempacotamento (RQ3).

Estrutura da Pesquisa

O presente estudo propõe uma abordagem sistemática para a compreensão do problema observado em estudos relacionados a instrumentação de aplicativos. Uma infraestrutura modular capaz de automatizar e monitorar o processo de reempacotamento é proposta. Essa infraestrutura foi aplicada a um conjunto curado de aplicativos populares da Google Play Store e, durante a execução dos experimentos, foram coletados e analisados dados sobre o comportamento desses aplicativos frente às diferentes etapas do reempacotamento. Os dados coletados ofereceram subsídios para a identificação de defesas e falhas de instrumentação. Um experimento prático foi utilizado para revisitar o problema teórico, permitindo identificar falhas que são decorrentes de mecanismos de defesa, falhas no processo de instrumentação que ocorrem durante a criação de variantes e falhas que ocorrem somente em tempo de execução.

Resultados e Discussões

Destaca-se o InstruMate, uma infraestrutura capaz de gerar e avaliar variantes controladas de aplicativos Android. Essa infraestrutura é capaz de modularizar o processo de reempacotamento e utiliza ferramentas especializadas para cada etapa. Para avaliar a resiliência de aplicativos Android ao reempacotamento, o estudo partiu dos 500 apps mais populares da Google Play Store, excluindo aplicativos pré-instalados. Os apps estáveis e livres de bugs serviram como base para gerar variantes controladas. Cada variante foi avaliada em múltiplas rodadas para detectar alterações comportamentais, classificando os apps originais como não resilientes ao reempacotamento, caso ao menos uma variante fosse obtida e não apresentasse diferenças comportamentais em relação ao original. Em geral, as chances de sucesso na produção de variantes foram maiores quando as modificações foram na assinatura, no manifesto ou em recursos (arquivos de configuração, imagens, entre outros). De outra forma, modificações baseadas em instrumentação apresentaram taxas de falha mais elevadas. Foi observado que APKs mesclados (*merged*) melhoraram levemente os resultados. O presente estudo também identificou mecanismos de defesa contra reempacotamento e causas raiz de falhas, destacando que nenhuma técnica de instrumentação se mostrou universalmente eficaz e evidenciando que diferentes técnicas de instrumentação devem ser utilizadas de forma complementar, o que pode maximizar as chances de sucesso. O presente trabalho revelou ainda que a maioria dos aplicativos analisados é altamente vulnerável ao reempacotamento. A proposta aprimora trabalhos anteriores ao impor critérios mais rigorosos para a avaliação de variantes.

Conclusões

Este trabalho apresenta uma abordagem sistemática para avaliar aplicativos Android frente a múltiplas técnicas de reempacotamento. O estudo também demonstra que mecanismos de defesa contra reempacotamento são fatores impeditivos para o sucesso de técnicas de instrumentação. Os resultados indicam que, quando o procedimento de reempacotamento interfere no código, os desafios se iniciam na construção do aplicativo reempacotado e permanecem durante a execução do aplicativo, que muitas vezes falha abruptamente. Argumenta-se que o InstruMate pode apoiar pesquisas futuras e ajudar desenvolvedores a implementar estratégias de proteção contra reempacotamento já nas fases iniciais do desenvolvimento de aplicativos Android.

Palavras-chave: Atestação de software, endurecimento (*hardening*) binário, engenharia reversa, Android, reempacotamento, proteção contra reempacotamento.

Abstract

Context: Application repackaging refers to the process by which attackers acquire legitimate mobile applications, reverse-engineer their code through decompilation, inject malicious payloads or modify existing functionalities, and subsequently repackage and distribute these tampered versions. The repackaging of applications poses a significant security threat, with the potential to impact developer revenue and expose users to malicious software. However, repackaging is also widely used in instrumentation techniques that embed monitoring mechanisms into apps, enabling security research on their behavior. The success rate of instrumentation techniques varies considerably, and generic explanations for failures leave the following question open: to what extent are popular apps susceptible to repackaging?

Objective: Identify and analyze the factors that lead to the failure of various instrumentation techniques based on repackaging.

Research Structure: This study proposes a systematic approach to understand the problem. To that end, a dedicated infrastructure was developed to automate and monitor the repackaging process. Then, this infrastructure was applied to a curated set of popular apps obtained from the Google Play Store. During the experiments, data was collected and analyzed on how these apps behaved when subjected to various stages of repackaging, providing insight for better understanding of the causes of repackaging failures.

Results: We present InstruMate, a systematic approach that generates controlled repackaged variants by modifying sensitive Android artifacts. This infrastructure is able to assess the resilience of the app to repackaging, using verification procedures that compare the behavior of each variant with its original version. In a curated dataset of 156 popular Google Play apps, our empirical study revealed that 86% are not resilient to basic repackaging (e.g., signature alteration), 81% allow execution in debug mode, 83% tolerate superficial modifications, and 65% are susceptible to advanced code tampering, while a small group actively deploys defenses against repackaging. In general, our contributions include a new repackaging infrastructure and an improved verification procedure for repackaged variants.

Conclusions: Overall, the findings highlight both the ease and the risks associated

with Android app repackaging, while also validating InstruMate as an effective infrastructure for assessing app resilience to such modifications. Security analysts can benefit from the proposed repackaging infrastructure. Since the instrumentation techniques used aim at code coverage, if an app proves to be not resilient to these approaches, it is likely that more advanced techniques will also succeed.

Keywords: Software attestation, binary hardening, reverse engineering, Android, Repackaging, Repackage-proofing

Contents

List of Figures

List of Tables

Abreviations and Acronyms

AAB Android App Bundle.

ADB Android Debug Bridge.

AOSP Android Open Source Project.

API Aplication Programing Interface.

APK Android Installation Package.

C&C Command and Controll.

DEX Dalvik Executable.

GUI Graphical User Interface.

MIME Multipurpose Internet Mail Extensions.

OEM Original Equipment Manufacturer.

Glossary

Activity A fundamental Android component that represents a single screen with a user interface.

AndroidManifest.xml The primary configuration file in Android applications that provides essential information about the app to the Android system. This XML file declares the app's components (Activities, Services, Broadcast Receivers, and Content Providers), required permissions, minimum API level, hardware features, and other metadata. The manifest file is required for every Android app and must be located at the root of the app's project directory.

AndroZoo A lresearch repository that continuously collects and archives Android applications from various sources for academic research purposes.

Apache Tika An open-source content analysis toolkit developed by the Apache Software Foundation that detects and extracts metadata and text from various file formats.

ApkEditor An Android application that allows editing of APK files..

ApkTool A tool for reverse engineering Android apps..

AspectJ A programming language extension for Java that enables aspect-oriented programming.

Bytecode An intermediate representation that is more abstract than machine code, designed for execution by virtual machines.

Debian Package (DEB) A file format used by Debian-based Linux distributions (including Ubuntu, Mint, and others) for software installation packages.

Dynamic Delivery The process used by the Google Play Store for generating device-specific APKs from Android App Bundles.

Dynamic Instrumentation A software analysis technique that modifies program behavior at runtime by injecting or altering code without permanently changing the original binary or source code .

Frida A dynamic instrumentation toolkit that allows injection of code into running processes for analysis and modification.

Intent An Android messaging object used to request an action from another app component.

Java Native Interface (JNI) A programming framework that enables Java code to call and be called by native applications and libraries.

Jimple An intermediate representation used by the Soot framework for analyzing and transforming bytecode.

Malware Malicious software designed to damage computer systems.

Manifest File A configuration file in Android apps that declares essential information about the app, including permissions and components.

Microsoft Installer (MSI) A file format used by Microsoft Windows for software installation packages.

MIME Types A standard way of classifying file types on the Internet by specifying the nature and format of documents, files, or assortments of bytes.

Notepad++ A free, open-source text editor and source code editor for Microsoft Windows..

Obfuscation The practice of making code difficult to understand to prevent reverse engineering.

Piggybacked Apps Repackaged applications that carry malicious payloads alongside the original functionality.

Protocol Buffers (Protobuf) A language-neutral, platform-neutral extensible mechanism developed by Google for serializing structured data.

Repackage-proofing Security mechanisms embedded in applications to detect and prevent execution of tampered versions.

Repackaging The process of modifying an existing application, typically by altering its code or resources, and redistributing it.

Reverse Engineering The process of analyzing software to understand its design, functionality, and implementation.

SHA-1/SHA-256 Cryptographic hash functions used to verify data integrity and generate unique identifiers.

Signature A cryptographic mechanism used to verify the authenticity and integrity of Android applications.

SMALI An assembler/disassembler for Android's DEX bytecode format.

SOCKS Proxy A **SOCKS** (*Socket Secure*) proxy is a network protocol that routes network packets between a client and server through a proxy server..

Soot A framework for analyzing and transforming Java bytecode.

Split APKs Multiple APK files generated from an Android App Bundle, each containing specific resources or code for different device configurations.

Static Analysis Examination of program code without executing it, used to understand structure.

Static Instrumentation A software analysis technique in which additional code, tracing instructions, or probes are inserted into a program's binary or source code before execution.

UI Automator Android's testing framework for automating user interface interactions.

View Views are rectangular areas on the screen that handle drawing and event handling.

Watermarking The practice of embedding identifiable markers within software to establish ownership and detect unauthorized copies.

Weaving In aspect-oriented programming, the process of integrating aspects (cross-cutting concerns) with the main program code.

Chapter 1

Introduction

Application repackaging refers to the process by which attackers acquire legitimate applications (or "app" for short), unpack them, reverse-engineer their code through decompilation, inject malicious payloads or modify existing functionalities, and subsequently repackage and distribute these tampered versions. The repackaging of Android apps poses a significant security threat, with the potential to impact developer revenue and expose users to malicious software.

Malicious actors exploit repackaging as a tool to spread malware, commit plagiarism, and exfiltrate sensitive information [6, 7, 8, 9, 10, 11, 12, 13, 14]. To mitigate this threat, software marketplace operators implement diverse strategies to identify and eliminate repackaged apps from their platforms [15, 16, 11, 17, 7].

Despite significant initiatives to combat the growing threat of repackaging, the problem continues to escalate. For instance, at the time of this research, repackaging Android apps remains easily accessible to general users. Popular automated repackaging tools, such as *App Cloner Premium* [1] and *ModHeaven* [2], are gaining traction on the Google Play Store, each surpassing five million downloads.

Repackaging is also widely used in instrumentation techniques [18, 19, 20, 21] that embed monitoring mechanisms into apps, enabling security research on their behavior. The success rate of instrumentation techniques varies considerably, and generic explanations for failures are often provided, limiting the scope and applicability of such techniques.

1.1 Research Problem

Previous research [18, 22, 20, 23] reports varying success rates in generating repackaged apps, with an average success rate of around 56% (as detailed in Chapter 2.3). Since these research studies treat repackaging as a preliminary step toward more advanced forms of analysis, they primarily emphasize the outcomes of the applied techniques, often overlook-

ing a detailed explanation of *why* the creation of certain repackaged apps fails. Among the scarce reasons cited, broad explanations are provided, typically attributed to factors such as the complexity and intrusiveness of the repackaging process [18]. Furthermore, potential failures caused by repackage-proofing [14] mechanisms possibly present in the apps are often overlooked.

Regardless of the perspective—whether to understand the threat posed by Android app repackaging or to evaluate the effectiveness of repackaging-based techniques—there **remains a gap in the literature regarding systematic evaluations of the challenges involved in repackaging apps**.

As detailed in Chapter 2, this study focuses on mobile apps, specifically those developed for the Android platform, with the primary objective of evaluating the extent to which they can be successfully repackaged and examining the feasibility of detecting repackaging failures. To guide this investigation, the following research questions are proposed.

(RQ1) How susceptible are popular Android apps to repackaging?

(RQ2) What are the common defenses Android apps leverage against repackaging?

(RQ3) What are the root causes that lead to repackaging failures?

This research examines a critical issue: instrumentation techniques that rely on repackaging often exhibit varying success rates and frequently fail, without providing clear diagnostic feedback. For the research community, this hampers the practical application of such techniques and limits their potential for broader adoption. For security analysts, it may obscure the true extent of the threat posed by repackaging.

1.2 Research Characterization

This study originates from the identification of a theoretical gap in the literature, namely the lack of explanations for why repackaging attempts fail. This gap is particularly relevant given that prior research consistently highlights the security threats posed by repackaging.

To address this issue, we propose the development of a repackaging infrastructure designed to decompose the repackaging procedure into discrete steps while generating fine-grained analytical reports. Framed under the Design Science Research (DSR) paradigm, this work implements a cycle in which the proposed infrastructure is evaluated through an empirical assessment involving a curated set of highly popular Android apps. The knowledge produced through this process contributes to explaining the underlying reasons

for repackaging failures, as detailed in Chapter 5. For example, our findings show that some failures are attributable to explicit repackaging defenses intentionally embedded in apps, while others arise from issues in the construction process of repackaged variants.

The empirical assessment focused on apps that have been installed more than 50 million times, including Facebook, Instagram, WhatsApp Messenger, and Facebook Messenger — each one exceeding 5 billion installations. This selection was motivated by two factors: (i) such apps represent more attractive targets for malicious actors seeking monetization through repackaging, and (ii) they are more likely to incorporate advanced security mechanisms. To operationalize the study, the research goal was translated into a set of research questions, which were subsequently refined into measurable metrics, an application of the GQM approach, as outlined in Section 4.1. The research structure is shown in Figure 1.1.

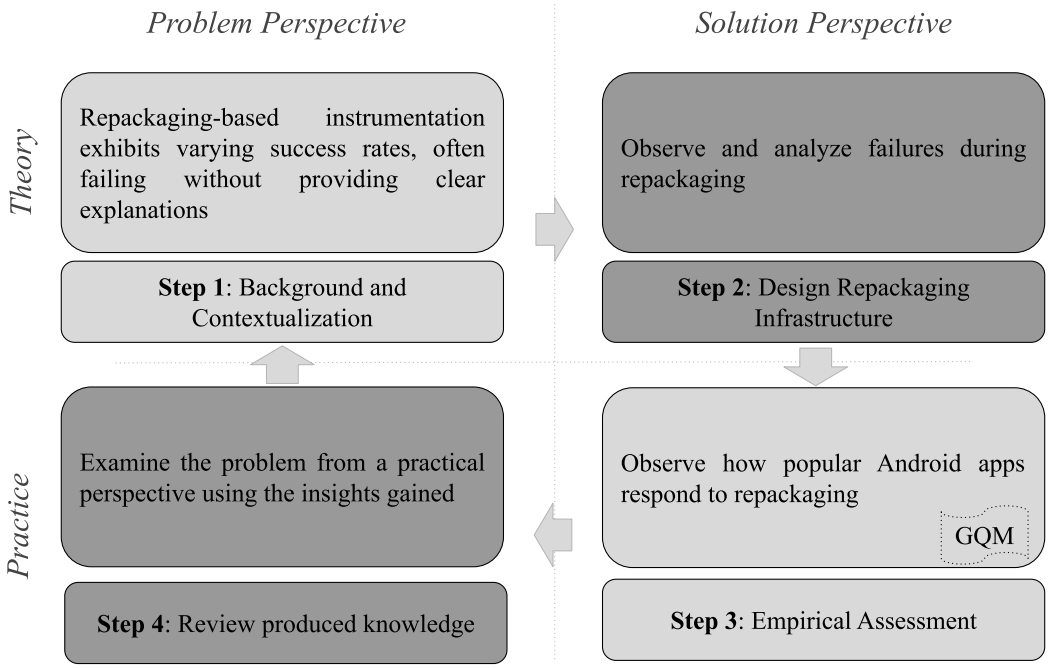


Figure 1.1: Research structure.

1.3 Overview of the Contributions

In summary, this research makes the following contributions:

- We introduce a new repackaging infrastructure designed to build controlled repackaged variants (Chapter 3)

- We propose novel health check procedure (Section 3.3) that classifies repackaged apps as *healthy* (fully compatible with the original); *healthy but incompatible* (functional, but with unintended differences); or *faulty* (non-functional). This represents an advancement over previous research in the field. Earlier studies [22], considered a repackaged variant to be healthy if it executed without crashing for a duration of only three seconds.
- We highlight that some apps implement defenses against repackaging by actively preventing execution. This observation is relevant for refining datasets used in scientific research on repackaging and instrumentation.
- We report the results of an empirical study showing that popular Android apps are susceptible to repackaging: 81% of the apps can be modified to run in debug mode, while 65% are vulnerable to code tampering using various techniques. Our study also reveals common practices employed by apps to prevent repackaging, as well as root causes of failures observed when using code instrumentation tools. The details of the empirical study are provided in Chapter 4 and Chapter 5.

1.4 Manuscript Organization

In the list below, we present the chapters and a summarized description.

- Chapter 2 — Background: introduces fundamental concepts of the Android platform relevant to this study, including the structure of installation packages and the app distribution process, repackaging, and code instrumentation.
- Chapter 3 — InstruMate: this chapter presents the design and implementation details of the proposed infrastructure. It outlines key aspects such as the pipeline used to create repackaged variants and the procedures employed to verify their integrity and functionality.
- Chapter 4 — Empirical Assessment: our systematic approach to evaluating the resilience of Android apps against repackaging is applied to a curated subset of the most popular apps from the Google Play Store. This selection ensures that the analysis targets modern, actively maintained apps, thereby offering a realistic assessment of their robustness to repackaging attempts. This chapter presents detailed information about the curated dataset and the methodology employed in the evaluation.
- Chapter 5 — Results: the selected set of original apps was subjected to a controlled repackaging process, resulting in the generation of modified variants. These variants

were subsequently verified for health and functionality. This chapter presents the success rates associated with variant creation, along with key findings that address the stated research questions.

- Chapter 6 — Final Remarks: this chapter discusses the implications of the findings, and highlight potential threats to the validity of the research.
- Appendix A — Additional Figures: presents supplementary figures that illustrate the repackaging process and provide visual documentation of the experiments conducted in this study.
- Appendix B — Apps: provides the complete listing of the apps selected to compose the dataset used in this study.

Chapter 2

Background

This chapter provides the background necessary to understand the technical concepts underlying this research.

Section 2.1 introduces the concept of repackaging, highlighting its relevance as a challenge that spans multiple platforms. Since the focus of this study is on the Android platform, Section 2.2 presents this architecture, and justifiy this choice. To contextualize repackaging within the Android ecosystem, Section 2.3 examines characteristics specific to the Android platform. Finally, Sections 2.4 and 2.5 present static and dynamic instrumentation.

2.1 Repackaging

To understand repackaging, it is necessary first to contextualize the notion of a software package. A package typically contains software along with metadata that specifies dependencies, shared libraries, versioning constraints, and installation requirements [24, 25]. Packaging systems facilitate software distribution and updating, ensuring that users receive bug fixes, security patches, and functional enhancements.

Examples of such systems include the `Microsoft Installer (MSI)` package format in Microsoft Windows [26] and the `Debian Package (DEB)` format used in Debian-based Linux distributions [27]. The success of platforms such as Debian is closely related to the robustness of their software packaging infrastructure [27], which allows efficient dependency management and large-scale software distribution.

Repackaging refers to the process of obtaining a legitimate software package, unpacking it, decoding, decompiling, and subsequently modifying and reassembling it into a redistributable form. Repackaging is often used for malicious purposes [28, 15, 11], and a common motivation is to redistribute commercial software without authorization, thus violating intellectual property rights. In addition, attackers can inject malicious compo-

nents that execute without user knowledge, enabling activities such as data exfiltration, privilege escalation, or persistence within enterprise environments.

High-profile incidents illustrate the severity of this threat. An example was the compromised redistribution of Notepad++ [29], where attackers delivered a trojanized variant of the widely used text editor. Given that Notepad++ is popular among system administrators and developers, the malicious version infiltrated enterprise networks and operated covertly as both a keylogger and a data exfiltration tool. Remaining undetected until actively exploited, it significantly amplified its impact across multiple organizations.

In the Notepad++ case, the attacker had to create a malicious variant and redistribute it to the end users. This approach inherently limited the attack surface to those who were persuaded to install the weaponized version. In contrast, a far more dangerous strategy is the compromise of the software supply chain, which allows malicious code to be embedded directly into trusted distribution channels. In the SolarWinds case [30], attackers interfered with the build process itself, injecting malicious instructions that became part of the final compiled software. By compromising the system responsible for compiling and signing updates, attackers ensured that malicious code was propagated through trusted distribution channels. The scale of the breach affected critical organizations worldwide, including several branches of the U.S. government, underscoring the systemic risks of supply-chain attacks.

Repackaging of mobile apps can impact a large number of users, particularly given the scale of modern app market repositories. Figure 2.1 illustrates the redistribution flow of repackaged Android apps. Initially, developers publish legitimate apps to official and third-party marketplaces, including the Google Play Store (Step 1). End users typically access these marketplaces to download authentic versions of the apps (Step 2). However, this standard distribution pathway can also be exploited by malicious actors, who obtain the official app packages (Step 3), apply repackaging techniques to modify them (Step 4), and subsequently redistribute the tampered versions through the same channels or unofficial channels (Step 5). The possibility that repackaged Android apps may reach official marketplaces is particularly noteworthy, given that a broad user base relies on these markets as trusted repositories for software distribution. The presence of malicious apps in such markets poses significant risks, potentially affecting a large number of users. This situation parallels supply chain compromises, such as the SolarWinds incident [30].

Beyond its malicious applications, repackaging also plays a constructive role in the software engineering process. Security professionals frequently employ repackaging to perform tests [22, 21, 18] and vulnerability assessments [19, 31, 23, 32]. By modifying apps in a controlled manner, they can uncover potential weaknesses and design flaws. For example, by instrumenting repackaged apps with monitoring code, it becomes possible

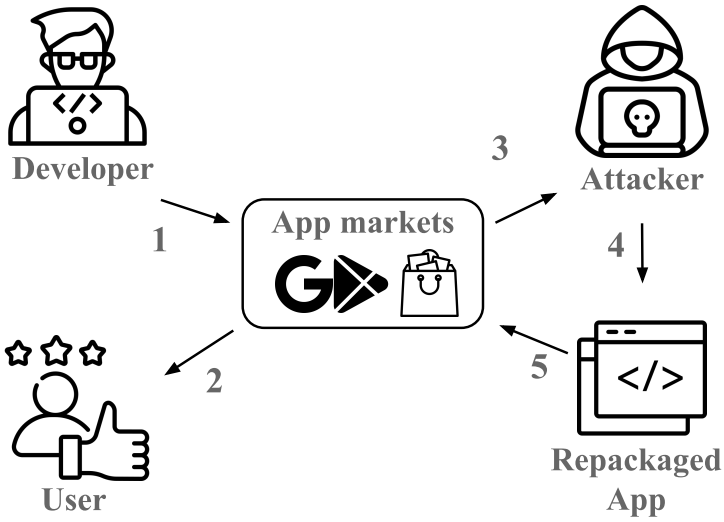


Figure 2.1: Repackaging threat model and redistribution.

to profile sensitive Aplication Programing Interface (API) calls, observe execution flows, and detect potential data leaks. Such modifications, though similar in mechanism to those performed by adversaries, are executed in ethical contexts to strengthen software resilience and guide secure system design.

There are multiple smartphone platforms, with Android maintaining a significantly larger market share than iOS—approximately 72% and 26%, respectively [33]. Beyond its market dominance, Android's open-source nature is particularly valuable for academic research, as it enables the development, customization, and dissemination of analysis tools, thereby fostering a collaborative environment for experimentation and reproducibility [34]. In contrast, iOS research often relies on physical devices due to the limited availability of robust emulator support, which significantly restricts both flexibility and reproducibility [35]. For these reasons, this study focuses on Android apps, even though iOS is also subject to repackaging threats, albeit to a lesser extent [36] due to built-in security mechanisms.

Repackaging has also become increasingly accessible to a wider range of users, including those without formal computer science training but with basic knowledge of computers and programming. Two apps available on the Google Play Store enable users to clone other apps: *App Cloner Premium* [1] and *ModHeaven* [2]. The use case scenario is straightforward: it involves selecting an official app as the target for repackaging and a set of features to be included. The premium features of the *App Cloner Premium* explicitly demonstrate its capability to modify specific components of apps subjected to the cloning process. In particular, the feature set includes the use of a SOCKS Proxy, which redirects traffic to potentially facilitate the exfiltration of sensitive data; audio playback capture, which can record all audio played within the cloned app; and the ability to automate tasks or

simulate button presses, which could potentially introduce malicious behaviors into the cloned app. More advanced users may employ free tools, such as APKTool-GUI [37], which presents a Graphical User Interface (GUI) for the ApkTool [38] repackaging toolkit and facilitates both the decoding and rebuilding of Android apps.

Currently, Google does not verify the identities of Android app developers. However, the scale of malicious app distribution has become so critical that Google has announced plans to introduce a comprehensive developer verification process in the future, as detailed in Subsection 2.3.4.

2.2 Architecture of Android Apps

Android apps are predominantly GUI components (named as *Activity*), that combine user interface elements (the *View*) to perform tasks. To transition to a different GUI, the a message mechanism (named as the *Intent* mechanism) takes place, stopping the current activity and launching the new one. The Intent message may be explicit, providing the target activity's class name, or implicit, which declares a general action to be performed, allowing the intent filters to take place. If a component's intent filter matches the implicit intent, it becomes eligible to handle it, enabling dynamic interaction between different components. Android apps may initiate broadcast receivers, start background, foreground, or bounded services, the last one bounded to the lifecycle of the initiator component. While broadcast receivers are components that respond to messages from other apps or from the system, content providers serve as a standardized interface to manage and share structured data, facilitating both intra-application and inter-application communication[39, 40]. The main components of Android's application model can be summarized as follows:

- **Activity**: Provides the foundation of the user interface, representing different screens of the app.
- **Service**: Offers background processing capabilities without a user interface.
- **Broadcast Receiver**: Responds asynchronously to system broadcast messages.
- **Content Provider**: Provides data repository capabilities to other components.

Android apps are commonly delivered to users as an Android Installation Package (APK) file that contains the installation-related components necessary for an app to function properly, including:

- Precompiled Dalvik Executable (DEX) bytecode stored in single or multidex files.

- Resources such as images, strings, layouts, etc.
- Precompiled code for different CPU architectures that can be loaded through Java Native Interface (JNI);
- Meta-data information such as the app's certificate for verifying the signature, versioning information, app permissions, entry points, and references to main components used by the app.

Figure 2.2, adapted from [3], illustrates the Android build process. As shown, Java source code is compiled to bytecode, and Java bytecode is then compiled into the DEX format, while resources and configuration files are also transformed to an encoded format. These components are subsequently packaged and signed to generate the final APK file, which is essentially a compressed archive that can be opened using standard compression tools available on the market. Figure 2.3 illustrates the assembled APK file of a simple experimental app that was compiled, assembled, and later inspected using the 7-Zip tool [4]. The first column shows the name of the archived item with its size in bytes on the right.

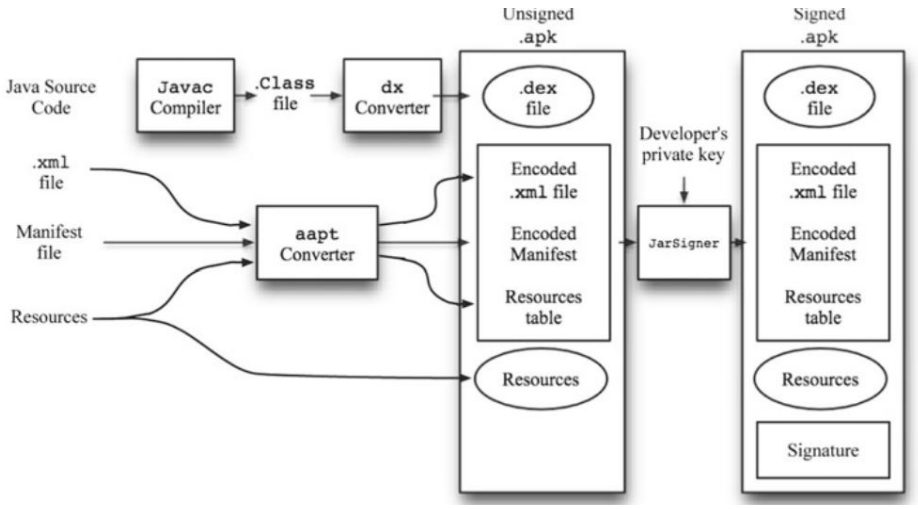


Figure 2.2: The Android build process, emphasizing artifacts and tools. Source: [3].

Although single APK files are still used for app distribution, they are no longer the standard file format used by the Google Play Store [41]. Since 2021, Android App Bundle (AAB) have been the standard format used by Google Play. Unlike traditional APKs, an App Bundle contains the code and resources for multiple device configurations but defers the APK generation to Google Play, a process known as **Dynamic Delivery**[42, 43]. This means that apps, especially those distributed on the Google Play Store, can exhibit significant variations on different devices due to the adoption of the AAB format. Google Play generates optimized APKs tailored to the specific configurations of individual

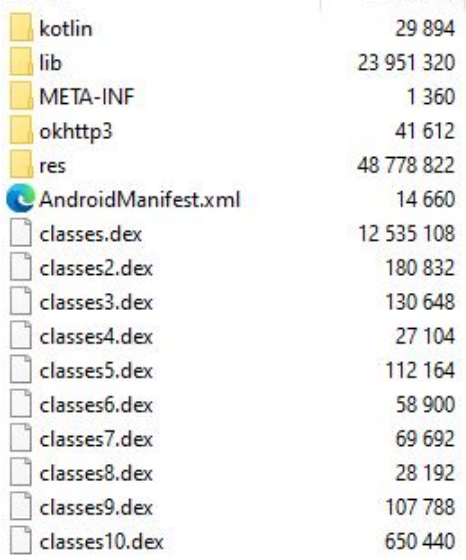


Figure 2.3: Shows an experimental app that was compiled, assembled, and later inspected using the 7-Zip tool [4].

devices. This optimization results in device-specific APKs that can be classified into the following types [42]:

- Base APK: contains core functionality.
- Feature APK: which contains code and resources related to a specific feature.
- Configuration APK: which contains specific device configurations, such as screen density, CPU architecture, and language resources.

Despite the fact that the Google Play Store uses Dynamic Delivery for app distribution, there are several single APK repositories, one of which is the AndroZoo project [10], a continuously growing dataset that aggregates Android apps from various sources. To collect apps from the Google Play Store, AndroZoo employs a custom crawler that utilizes a reverse-engineered client of the Protocol Buffers (Protobuf)[1] protocol and a valid Google account linked to a specific device. Allix et al. [10] and Alecci et al. [44] did not indicate whether Androzoo collects dynamically delivered apps (or split apps for short), and if so, to which device the account is linked and how split apps could be converted into a single APK file, raising questions about the completeness and representativeness of the dataset with respect to split apps.

Split apps can potentially influence the repackaging process in several ways. For example, all parts must share the same signature. Additionally, resources or code present in one split may be overridden by elements from another. During repackaging, adding

[1] https://protobuf.dev/

new components to one split can result in unintended duplications if the same additions are made to other splits. Finally, the repackaged apps, as a consequence, can also be device-specific. Few studies [45, 42] mention the repackaging of split apps. However, in the gray literature [46, 47, 48], there is a growing adoption of a process for merging parts, where multiple split APKs are combined into a single APK.

2.3 Android Repackaging

To gain a deeper understanding of the repackaging process within the Android ecosystem, Li et al. [11] dissected **piggybacked apps**, a term used to refer to repackaged apps that include malicious payloads [15]. Piggybacked apps include modifications to key components such as activities, broadcast receivers, services, permission settings, and content providers. Piggybacked apps act as a **carrier** for a malicious payload, referred to as the **rider**. The malicious behavior is activated through **hooks**, which can take the form of explicit method calls that connect the carrier to the rider or through a registered component that can be launched independently to initiate the rider.

2.3.1 Targeted Artifacts in Repackaging

When analyzing repackaging practices, several key artifacts within Android apps are commonly targeted for modification.

Signature Artifacts

This is the most fundamental change introduced during repackaging. Any alteration to the app package invalidates the original cryptographic signature, necessitating the generation of a new one. Signature replacement is therefore inherent to any repackaging attempt, as unsigned modifications would trigger verification errors during installation or execution [22]. If someone alters and redistributes an app with a different signature, future updates must be signed with the same certificate. This can negatively impact the original app's revenue [28] and potentially introduce malware in future updates. Furthermore, merely changing the signature is a practice known as *Lazy Cloning* [28], which may be done for fun or fame [14]

Manifest file

The manifest file is a frequent target due to its central role in defining critical app attributes. It specifies the app's components (e.g., activities, services, broadcast receivers), requested permissions, and configurations such as whether the app allows debugging or

permits cleartext traffic. Modifying this file can significantly alter the app's behavior and security. By altering this file, an attacker can add, remove, or modify advertising IDs or request additional permissions [28]. Furthermore, modifying the manifest can enable the *debuggable* flag, allowing the use of profiling tools for inspection [49, 50], while also slowing the app during execution. The manifest file can also be configured to export sensitive data during backups or, in some cases, weaken the app's network security by crafting a weakened network configuration profile [51].

Resource and Asset Files

Although these files do not contain executable code, they can influence the app's appearance and runtime behavior, particularly when configuration files store parameterized variables. Resource files include user interfaces, strings, and images, while asset files may contain auxiliary content loaded at runtime. According to [11], 91% of piggybacked apps have modified resources compared to the original implementation.

Executable Code Sections

Both security analysts and malware developers frequently target the app's compiled code. Analysts typically inject instrumentation probes to enable behavioral monitoring and security assessments [18, 19, 22, 45]. In contrast, attackers may tamper with the code to bypass security checks, insert malicious payloads, or activate dormant malware functionalities [11, 15]. Code tampering can significantly impact an app's functionality, potentially introducing new features, altering existing behaviors, disabling security mechanisms, or exfiltrating sensitive user data. Such changes compromise the app's integrity and may lead to malicious outcomes.

2.3.2 Repackaging Android System Apps

Repackaging firmware-provided apps—typically pre-installed system apps—requires special consideration due to their reliance on custom framework resources not present in the standard Android Open Source Project (AOSP). These dependencies can lead to failures during the repackaging process unless properly addressed. A critical first step involves identifying the specific framework components required by the target app, which varies across Original Equipment Manufacturer (OEM) such as Samsung, Xiaomi, and others. Once identified, these framework files must be incorporated during the decoding phase, as they contain essential resources referenced by the app. If modifications to the framework itself are necessary, successful deployment may require replacing the framework on

the target device—a task that typically demands root access and introduces additional complexity.

Beyond the static repackaging stage, runtime execution of the modified app presents additional challenges. For example, non-rooted environments prevent the overwriting of pre-installed system apps, particularly if the modified APK is signed with a different key. Additionally, the system does not permit multiple apps with the same package identifier. While altering the package ID may allow installation, it may also lead to reduced functionality, as system apps often rely on privileged permissions that are not granted to regular user-installed apps.

Finally, much of the practical knowledge surrounding the repackaging of firmware apps stems from gray literature, such as community forums, tutorials, and tool-specific documentation (e.g. [52] and [38]). These sources often lack the rigor and completeness required for reproducibility and fail to comprehensively address the limitations and pitfalls associated with this complex process.

2.3.3 Repackage Tools

Several studies [53, 22, 54, 45, 18, 55, 56, 14, 6] have relied on ApkTools and ApkEditors to perform essential operations involved in the repackaging process. ApkTool and ApkEditor support unpacking, modifying, and reassembling Android apps, each utilizing different underlying infrastructures: ApkTool relies on the Android Asset Packaging Tool (aapt/aapt2), a core component of the Android build system, while ApkEditor employs ARSCLib, a reverse-engineered library developed to parse and modify Android resource files without depending on aapt/aapt2. Resources and configurations are stored in a binary format, which must be decoded and converted to an editable format. The app's compiled code can also be decompiled into Smali, converted from DEX bytecode to Java bytecode, or to an intermediary representation such as Jimple, with the Soot framework (see Section 2.4, for more information on this topic). After modifications to the resources or code are made, the app is re-assembled into a repackaged version, signed, and ready to be redistributed. This signing mechanism ensures the integrity of the APK, as any modification to the package would render the signature invalid.

2.3.4 Defenses Against Repackaging

Several strategies have been proposed to detect repackaged apps, and representative state-of-the-art mechanisms are summarized in Table 2.1. In general, these detection techniques can be classified into four main categories [15]: **(i) static similarity-based pairwise comparison**, which examines whether two apps share sufficient similarities to be consid-

ered variants of the same original app; **(ii) machine learning-based techniques**, which are trained to extract relevant features from apps under investigation and infer whether they have been repackaged; **(iii) runtime monitoring approaches**, which observe app behavior during execution to identify anomalous activities indicative of repackaging; and **(iv) symptom discovery methods**, which operate under the premise that the repackaging process introduces specific, detectable anomalies or "symptoms" within the repackaged app.

Table 2.1: Android app repackaging detection mechanisms.

Tool Name	**Detection Mechanism**
Androguard [56]	Similarity Comparison
DNADroid [12]	Similarity Comparison
DroidSim [57]	Similarity Comparison
DroidMarking [58]	Runtime Monitoring
ResDroid [59]	Unsupervised Learning
DR-Droid2 [60]	Supervised Learning
AndroidSOO [61]	Symptom Discovery

Despite the availability of these techniques, accurately identifying repackaged apps remains a challenging task. Even when two highly similar apps are detected, determining which version represents the original and which constitutes the repackaged variant is inherently tricky, especially in the absence of a definitive repository of original apps for comparison. In addition, none of those above strategies scale effectively to the real-world scenario of multiple app stores, each hosting numerous versions of thousands to millions of apps [15].

Considering the current scale of the Google Play Store, which hosts approximately 1.6 million apps [62], even under highly optimistic conditions, the task of identifying repackaged apps remains a challenge. Assuming the existence of an algorithm capable of comparing two apps within one millisecond, and a computing environment capable of executing 100 parallel comparisons simultaneously, **it would still require approximately 148 days** of continuous computation to exhaustively compare each pair among the $\binom{1.6\times10^6}{2}$ possible combinations in search of repackaged variants. Moreover, the presence of multiple alternative app stores, each distributing numerous versions of the same app over time—including outdated versions that remain susceptible to repackaging—further exacerbates the challenge. These factors collectively illustrate the considerable complexity and difficulty inherent in accurately identifying repackaged apps. This line of reasoning is rigorously articulated by Li et al. [15], who describe the phenomenon as a **combinatorial explosion**. In their work, Li et al. advocate for a renewed research effort within the scientific community to address the challenges posed by repackaging.

Other techniques can also be used to protect against repackaging, including obfuscation and watermarking:

Obfuscation is a technique employed by legitimate developers to increase the difficulty of reverse engineering by making the code harder to interpret, forcing attackers to invest more time and effort to understand the app [63]. Obfuscation applies transformations that can be broadly categorized as **layout obfuscation**, which alters the layout and structure of compiled code, **control-flow obfuscation**, which alters the program's control-flow structure, or **data obfuscation**, which modifies data structures and variables [64]. More sophisticated techniques may also involve self-modifying code, in which the program alters its own instructions at runtime to hinder the analysis [63].

Watermarking serves to distinguish legitimate ownership by embedding identifiable markers within the app [65] that should survive the repackaging process. *Birfthmarking* refers to the technique of extracting unique characteristics from a software application to facilitate its identification. Birthmarks can be derived from various aspects of an app, including its opcode sequence, third-party library dependency graph, control-flow graph, or user interface components [17]. Although watermarking is generally not designed to prevent the execution of repackaged apps, it may serve this purpose if a trusted third party can detect and verify the presence of the watermark [45].

Although the defense techniques mentioned can discourage attackers, they do not provide a failsafe solution, as if these security mechanisms are bypassed, they cannot prevent repackaged apps from being distributed [14]. In contrast, **repackage-proofing** offers a more proactive approach by embedding defense mechanisms directly into the app [14]. This strategy detects repackaging when it occurs by introducing intentional code designed to trigger failures exclusively in repackaged variants, without affecting the original app's functionality.

Robust repackage-proofing techniques [14, 53] embed a network of integrity checkers (guards) that monitor integrity and trigger alerts, resulting in failures distant from the point of detection, leading to sporadic or buggy behaviors. This approach not only complicates the attackers' task of identifying and disabling the mechanisms but also creates an illusion of unpredictable behavior. An app that is embedded with these security schemes may also detect tampering and cooperate with a remote entity to deny service, simply identify the presence of the repackaged app, or immediately initiate code repair.

Repackage-proofing mechanisms may include specific routines, such as signature validation, code integrity verification, resource integrity checks, and installer verification,

that confirm if the app was not modified and was installed through any authorized app store, denying execution otherwise. Although these measures increase the difficulty for adversaries to compromise the app, it is important to note that software-based protections can ultimately be bypassed if an adversary is determined to invest the necessary time and resources [66]. Examples of repackage-proofing schemes include Droidmarking [58], SSN-Stochastic Stealthy Network [14], and AppWarder[66].

The Google and Apple Strategies for Preventing Repackaging

Google provides the Play Integrity API [67], which enables apps and their corresponding backend services to verify whether a request originates from an unmodified app binary that was legitimately installed via Google Play. However, the use of this API is optional and must be explicitly integrated by developers to protect against unauthorized access from compromised or repackaged apps.

Additionally, Google offers an automatic integrity protection mechanism that injects runtime verification code to detect signs of repackaging. This mechanism is available only when developers adopt the AAB. When enabled, the default behavior upon detecting a repackaged app is to redirect the user to the official app page on the Google Play Store. Importantly, this protection is opt-in and must be explicitly activated by developers during the app publishing process.

At the time of this research, an Android app's signature does not inherently indicate that a trusted third party issued the certificate. This design choice lowers the barrier for the distribution of repackaged apps, as any entity can sign an APK with a self-generated certificate. Consequently, malicious or unauthorized app modifications may be more easily introduced into the ecosystem. To address this limitation, Google has announced plans to introduce a verification process for both developers and their signing certificates in the future [68].

Although iOS is less susceptible to repackaging attacks compared to Android, it is not entirely immune to them. iOS apps must be signed using a valid developer or distribution certificate, and the integrity of this signature is verified by the operating system both at installation and at runtime [69]. Furthermore, the Apple App Store enforces a stringent app review process that all apps must undergo before distribution. This review extends beyond superficial code or interface analysis, encompassing evaluations of the app's business model, minimum functionality, and overall uniqueness [70].

In contrast to Android, signature verification on iOS devices is conducted within the Secure Enclave—a dedicated, tamper-resistant processor designed to ensure the security of cryptographic operations. As a result of these protections, repackaged iOS apps typically

require physical access to the target device for sideloading, which significantly restricts the potential attack surface.

Nonetheless, repackaging remains a significant threat within the iOS ecosystem. Decrypted apps can be obtained from websites [71] that facilitate the redistribution of repackaged iOS apps, underscoring the persistence of this issue despite Apple's more restrictive environment.

2.3.5 Android Repackaging Examples

To further illustrate Android repackaging, four popular unofficial WhatsApp variants are presented in Table 2.2. The table lists their names, package identifiers, versions, and the first ten digits of their respective SHA-1 hash values. In the initial stage, these variants display a logo and user interface elements identical to those of the official app. Additionally, these variants provide extended functionalities not available in the official app, including the ability to hide online status, conceal typing and recording indicators, selectively disable read receipts for groups or individual contacts, view deleted messages, and prevent other users from deleting previously sent messages. Despite the benefits, several sources have explicitly associated these variants with malicious or suspected activities, including account theft, exfiltration of sensitive data, and exposure of users to unwanted or harmful content [72, 73, 74, 75].

Table 2.2: Sample of unofficial WhatsApp variants.

App Name	App ID	Version	SHA1
FMWhatsApp [76]	com.wkfmwaapphfm.messenger	2.25.11.75	e70965bf40767aee
GBWhatsApp [77]	com.whatsapp.Main	2.25.9.78	832c7d5999d9e5c8
WhatsAppPlus [78]	app.executorsenderl.sko	2.25.9.78	52a63dc9ccf40bd2
YoWhatsApp [79]	com.exchanget.processorka	2.25.9.78	5ba234790dab729a

Kaspersky published a report [5] confirming the presence of spyware embedded within one modified variant of WhatsApp. The variant included suspicious additional components that, upon reverse engineering, was found to contain code explicitly designed to communicate with a Command and Controll (C&C) server.

Once activated, the spyware initiates periodic POST requests—transmitting information about the infected device and its contact list at five-minute intervals [5]. Additionally, it is configured to receive commands from the C&C server at pre-defined intervals, with a default polling frequency of one minute [5].

Further analysis confirmed that the spyware is capable of listing files on the device and exfiltrating sensitive data, including images, videos, and audio recordings—whether captured via the device's camera or stored in external memory.

Figure 2.4 illustrates the suspicious components embedded in the repackaged app added in the form of a Broadcast Recieiver, defined by the tag `receiver`. Figure 2.5 presents the code segment responsible for establishing communication with the C&C server. Note that the code is obfuscated and uses `Application_DM` constant in the malware code to select the C&C server. Figure 2.6 depicts the logic responsible for initializing threads that transmit contact information and receive remote commands.

```
<receiver android:name="com.google.android.search.validate.r0" android:enabled="true"
  <intent-filter>
    <action android:name="restartservice" />
    <action android:name="android.intent.action.BOOT_COMPLETED" />
    <action android:name="android.intent.action.ACTION_POWER_CONNECTED" />
    <action android:name="com.google.android.search.validate.restart-service" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</receiver>
<service android:name="com.google.android.search.validate.b0" android:enabled="true" a
```

Figure 2.4: Suspected additional components added to an WhatsApp variant. Source: [5].

```
n4.b = "/api/v1/AllRequest";
String s = c0.a;
String s1 = e0.Application_DM;
if(a.B(new byte[]{'1'}, s1)) {
    s2 = "aHR0cHM6Ly9hcHBsaWNhdGlvbi1tYXJrZXRpbmcuY29t";
}
else if(a.B(new byte[]{'2'}, s1)) {
    s2 = "aHR0cHM6Ly93aGF0c3VwZGF0ZXMuY29t";
}
else if(a.B(new byte[]{'3'}, s1)) {
    s2 = "aHR0cHM6Ly9hbmRyb2lkLXNvZnQtc3RvcmUuY29t";
}
else if(a.B(new byte[]{'4'}, s1)) {
    s2 = "aHR0cHM6Ly93aGF0cy1tZWRpYS5jb20=";
}
else if(a.B(new byte[]{'5'}, s1)) {
    s2 = "aHR0cHM6Ly93aGF0cy1tYXRlLmNvbQ==";
}
else if(a.B(new byte[]{'6'}, s1)) {
    s2 = "aHR0cHM6Ly93aGF0cy1tYXRlLm5ldA==";
}
else if(a.B(new byte[]{'7'}, s1)) {
    s2 = "aHR0cHM6Ly93aGF0cy1teWRucy5jb20=";
}
else if(a.B(new byte[]{'8'}, s1)) {
    s2 = "aHR0cHM6Ly93aGF0cy1teWRucy5uZXQ=";
}
else if(a.B(new byte[]{'9'}, s1)) {
    s2 = "aHR0cHM6Ly93aGF0cy12cG4uY29t";
}
else {
    s2 = a.B(new byte[]{'1', 0x30}, s1) ? "aHR0cHM6Ly93aGF0cy12cG4ubmV0" : "";
}
```

Figure 2.5: Code that establishes communication with the C&C. Source: [5].

2.4 Instrumentation

Instrumentation refers to the process of inserting additional code into a program for a specific purpose. A common objective is to embed instructions that trace code execution at runtime [80]. During dynamic analysis, these traces, together with a collection engine that records their execution, provide detailed statistics on which part of the code is being

```
if(x0.isDeviceInfoUploaded().booleanValue()) {
    b00.getClass();
    Thread thread1 = new Thread(new GetParams());
    thread1.setPriority(1);
    thread1.start();
    x0.o(10.0f);
    Thread thread2 = new Thread(new GetCommands(new long[]{0L}));
    thread2.setPriority(1);
    thread2.start();
    Thread thread3 = new Thread(new UploadAccountsAndContacts(b00));
    thread3.setPriority(1);
    thread3.start();
}
```

Figure 2.6: Setup of background threads that exfiltrate sensitive data. Source: [5].

executed. This information supports systematic testing procedures that exercise different parts of an app and capture potential errors. *Code coverage* is a widely used metric in the field of software testing that measures the extent to which a program's source code is dynamically executed and monitored. It can be assessed at different levels of granularity, such as class, method, or instruction. Static instrumentation is a common approach to modifying already compiled programs for the purpose of calculating this metric [18].

When source code is available, static code instrumentation is typically employed, allowing modifications directly in the program's source [80]. In cases where the source is unavailable, binary (or black-box) code instrumentation is used to alter compiled code and insert probes (trace statements). The ability to perform it in a black-box manner is crucial for analyzing apps in security-related scenarios, such as the two-phase Android Mining Sandbox approach proposed by Jamrozik et al. [31]. In the first phase, Android apps are monitored using a setup designed to systematically explore program behavior, recording access to sensitive APIs. The data collected are then used to build a sandbox that restricts access to APIs only seen in the exploration phase. One common way to implement this concept is through black-box instrumentation. To assess the applicability of the Mining Sandbox approach for malware classification, Bao et al. [23] conducted a large experiment using DroidFax [32] to embed monitors into repackaged apps that could trace calls to sensitive APIs. This form of instrumentation is similar to black-box code coverage, but adapted to the specific context of monitoring sensitive API calls.

These modifications can be classified as either static or dynamic. Static instrumentation permanently alters the app's code prior to execution, whereas dynamic instrumentation applies changes only at runtime, leaving the original codebase intact. Similar to static instrumentation, dynamic instrumentation also enables the transformation of specific instructions, arguments and return values, or even functions can be entirely modified, allowing the modifications to existing logic [81]. Instrumentation strategies vary. One common approach involves translating the target code into an **intermediate representation**, where transformations are applied before converting it back to the original

format. Alternatively, modifications can be made directly within the program's code sections, as illustrated in Figure 2.7.

Frida[2] is a widely used dynamic instrumentation framework, with repackaging commonly used to embed it directly in the app for redistribution. In this mode, Frida can either inject predefined instrumentation logic at startup or dynamically retrieve the instrumentation script from an external server [81, 82]. Frameworks such as Soot [83] implement instrumentation by converting Android application code to the Jimple [84] intermediate language, applying the desired changes, and then converting the modified code back to DEX format. Another approach transitions the code from DEX to Java bytecode [64], performs the required transformations, and subsequently converts it back to DEX. Additionally, instrumentation can be performed directly at the opcode level, enabling modifications without the need for an intermediate representation.

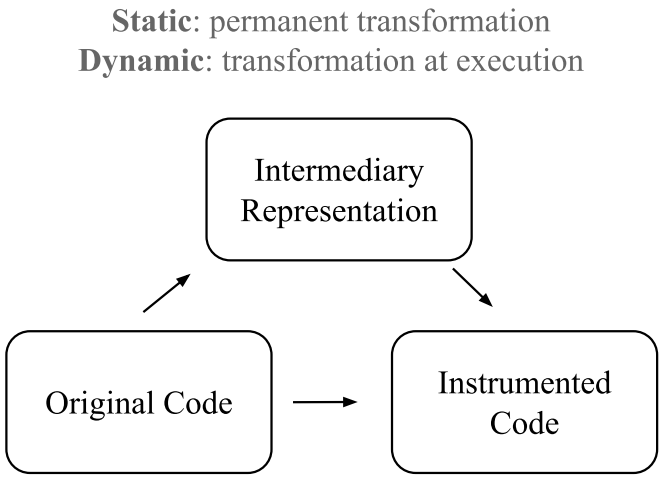


Figure 2.7: Binary instrumentation concepts.

2.5 Instrumentation for Security Assessment

Black-box app assessment techniques, such as Mining Sandbox [31], Joe Sandbox Mobile [85], and RV-Android[19], often rely on underlying instrumentation infrastructure such as DroidFax [32], COSMO [20], ACVTool [22], Androlog [18], or Frida [81]. COSMO Black-box approach[20] transforms Dalvik bytecode to Java bytecode and applies JaCoCo offline instrumentation which adds an array of probes and opcodes to the Java bytecode, capturing when code is executed. The Java bytecode is converted back to Dalvik and integrated into a repackaged app. ACVTool[22] instruments the Dalvik bytecode in its Smali representation. It uses instructions available on Android platforms to capture code

[2] https://frida.re/

coverage. ACVTool uses ApkTool to make changes to the `AndroidManifest.xml` file, transforming the app to require permission to write to the external storage, where coverage data are saved. DroidFax [32] and AndroLog [18] transform Dalvik bytecode into Jimple [84] intermediary representation using the Soot framework [83]. DroidFax injects coverage statements into sensitive APIs and AndroLog injects a new class named `LogCheckerClass` that prints only a log if it was not previously logged, to not slow down the app. Any other instrumentation consists of calling this class to produce coverage. Then, Jimple is used to generate the DEX instrumented code. RV-Android [19] converts the DEX file into a JAR and applies Java Aspect-Oriented Programming, similar to COSMO, but it also extends beyond that by enabling integration with a Monitor-Oriented Programming framework. Once the transformation and weaving are complete, the instrumented JAR is transformed to DEX, and the app is repackaged with the instrumented code.

Existing tools demonstrate a varying success rate in instrumenting apps [22], with average estimated to be around 56%. For instance, Samhi et al. [18] reported that ACV-Tool could instrument 48% of the selected apps, COSMO[20] 79%, Androlog 98%, and Bao et al. [23] conducted a large experiment using DroidFax [32], which showed that out of 2,750 apps, only 844 (31%) could be successfully instrumented. This information is summarized in Table 2.3.

Table 2.3: Summary of code instrumentation tools.

Tool	Representation	Success Rate	Additional Requirements
ACV-Tool [22]	direct/SMALI	48%	ApkTools
COSMO [20]	intermediary/Java bytecode	79%	Jar2Dex/Dex2Jar
Androlog [18]	intermediary/Jimple	98%	Soot
DroidFax [32]	intermediary/Jimple	31%	Soot

2.6 Chapter Summary

This chapter presented concepts underlying the study, with emphasis on repackaging and its implications in the Android ecosystem. It defined software packaging and repackaging, illustrated real-world risks, and justified Android as the focal platform due to its market share and open tooling. The chapter detailed Android's application model and packaging formats (APK and AAB), including the practical complications introduced by split APKs for analysis and repackaging workflows.

The chapter examined Android repackaging in depth: typical targets of modification (signatures, manifests, resources, and executable code), the particular challenges of

repackaging system apps, and the tooling commonly used to decode, modify, and rebuild apps. It reviewed detection and defense strategies—similarity analysis, machine learning-based approaches, runtime monitoring, symptom discovery, obfuscation, watermarking/birthmarking, and repackage-proofing—and highlighted scalability limits such as combinatorial explosion.

Finally, the chapter introduced instrumentation as both a research and security assessment technique, contrasting static vs. dynamic approaches and summarizing representative frameworks (e.g., Soot/Jimple, Frida, ACV-Tool, COSMO, DroidFax, Androlog) and their reported success rates.

Chapter 3

InstruMate

This chapter describes the repackaging infrastructure named InstruMate, a repackaging infrastructure that decomposes the repackaging process into smaller, monitorable steps, facilitating the extraction of detailed information about potential failures that may occur during app repackaging. The design rationale for InstruMate is that the literature underscores considerable variation in the success rates of instrumentation applied to Android applications.

InstruMate's approach separates unpacking and repacking steps from instrumentation, allowing the use of specialized tools optimized for each phase. The suite of strategies used to generate variants is centralized into a component, which we call *Variant Maker*. Different issues may arise during the construction of a variant, and even if the building process is completed without errors, the variant may encounter runtime problems that render it non-functional.

We first detail the InstruMate's variant creation pipeline, which is composed of a two-stage pipeline, as illustrated on Figure 3.1: the static analysis stage and the variant maker stage. As input, InstruMate accepts an app as input, along with a specification of the desired features to be included in the variant, and then InstruMate outputs either a repackaged app that meets the specifications provided or an error report. Section 3.1 explains the static analysis stage, and Section 3.2 explains the variant maker stage. Then we describe the procedures for verifying the health and functionality of the generated variants (Section 3.3), and Section 3.4 presents a usage scenario of InstruMate.

3.1 Static Analysis Stage

In the first stage of the variant creation pipeline, InstruMate executes a set of static analysis tasks of a given app, involving an *app analyzer*, a *content analyzer*, and a *binary analyzer*.

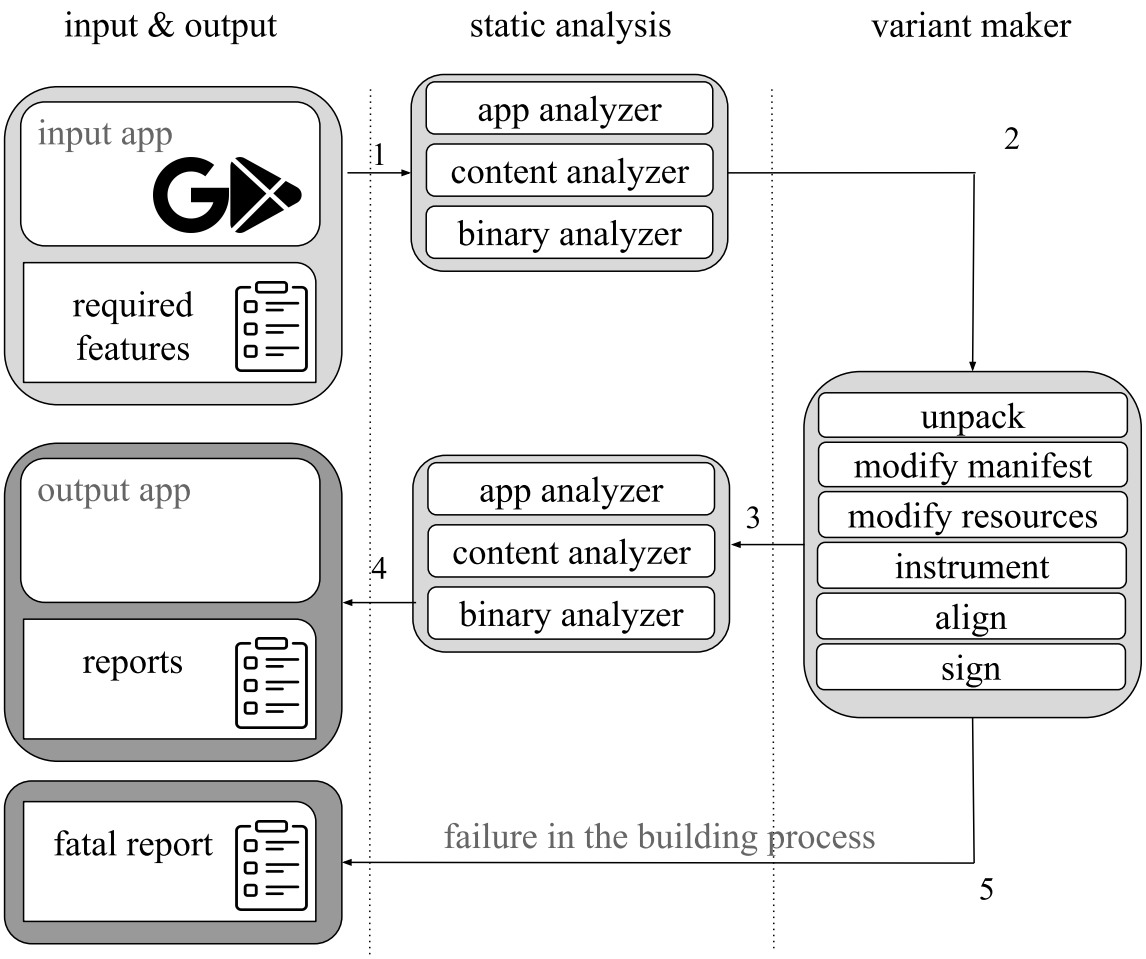


Figure 3.1: InstruMate's variant creation pipeline.

During the *app analyzer* task, InstruMate leverages Androguard [56] to collect key app characteristics including activity and service listings, permissions, package ID, version, signature information, and split count. Next, during the *content analyzer* task, InstruMate computes the SHA-256 hash of each individual file within the app. In addition, it uses Apache Tika[1] to determine the Multipurpose Internet Mail Extensions (MIME) type of each file within the app. This process uniquely identifies each file by its hash and provides essential metadata for *variant makers*.

Typically, Android apps store their compiled code within multiple DEX files included in the APK. However, they may also contain additional compiled code as assets or in the form of native code (files usually with the `.so` extension), which is accessed through the Java Native Interface (JNI). During the *binary-analyzer* task, InstruMate leverages Radare2[2] to analyze the binary files identified by Apache Tika as: `application/x-dex`, `application/x-executable`, or `application/ x-sharedlib`, and thus verify whether these files contain structures typically found in compiled code. This task provides additional information to the variant maker, including a detailed report on compiled code that covers symbols, imports, exports, library dependencies, function names, and strings.

The static analysis stage plays a crucial role in extracting useful information for variant makers, which leverage this data to optimize the variant creation process. For example, if a split APK contains confirmed DEX code identified during the static analysis phase, the variant maker can apply a specialized handling strategy for that particular split. Furthermore, a detailed analysis of the produced package contents enables a more thorough comparison between the original app and its variants, particularly aiding in the identification of repackage failures. *For this reason, the same static analysis procedures are also performed on the resulting variant (Step 3 in Figure 3.1).*

3.2 Variant Maker Stage

Variant makers are currently configured to generate repackaged variants with controlled modifications that preserve the original program logic. These modifications are designed to exercise a range of app transformations, beginning with trivial changes—such as modifying the app's digital signature, altering the manifest file, or changing resource elements like the app's name—changes for which any app should be prepared. The process then extends to more sophisticated transformations, including those introduced by static and dynamic instrumentation techniques. To facilitate analysis, the modifications were cate-

[1] https://tika.apache.org/
[2] https://rada.re/

gorized into four groups: Signature Modifications, Manifest File Modifications, Resource Modifications, and Code Modifications.

(G1) Signature Modification

The Android platform requires that any update to an app must be signed with the same key as the version already installed. This ensures that updates originate from the same developer [22]. If someone alters and redistributes an app with a different signature, future updates must be signed with the same certificate. This can negatively impact the original app's revenue [28] and potentially introduce malware in future updates. Furthermore, merely changing the signature is a practice known as *Lazy Cloning* [28], which may be done for fun or fame [14]. It also serves as a prerequisite for more sophisticated modifications, such as those expected in the G2, G3, and G4 groups (detailed below). The proposed implementation generates variants in this group by replacing the signature of the original apps. The procedure for generating **G1 variants** consists of unpacking the APK archive, removing the original signature metadata, generating new signature files, and reassembling the modified package into a valid APK. To achieve this, two main variant makers were configured: Me, which uses ApkEditor; and Mz, which relies solely on standard ZIP operations. These variant makers operate on a per-APK basis, generating split variants for apps originally delivered as split APKs. Additionally, the variant maker MeM was developed to create merged variants by utilizing InstruMate's merging capabilities. Since Mz lacks decoding capabilities, it was exclusively used to generate variants within the G1 group.

(G2) Manifest File Modifications

The manifest file is the primary configuration file for an Android app. By altering this file, an attacker can add, remove, or modify advertising IDs or request additional permissions [28]. Furthermore, modifying the manifest can enable the *debuggable* flag, allowing the use of profiling tools for inspection [49, 50], albeit at the cost of slightly reducing the app's performance. The manifest file can also be configured to export sensitive data during backups or, in some cases, weaken the app's network security by crafting a vulnerable network configuration profile [51]. The proposed implementation generates variants in this group by changing the manifest file of an original app to enable debugging capabilities. If the *variant configuration specification* required enabling the app's debuggable mode, Me and MeM are then configured to decode the `AndroidManifest.xml` in the base APK, activate the debuggable flag, and generate outputs with this modification in

addition to the signature change, thus producing split and merged variants within the G2 group.

(G3) Resource Modifications

Changes applied exclusively to the resource files can alter an app's name if it is stored in a resource string, modify the app's appearance, or adjust settings in configuration files, potentially impacting the app's behavior. According to [11], 91% of piggy-backed (repackaged) apps have modified resources compared to the original implementation. The generated variants in this group are configured to change the name of the app if its name is stored in a string resource item. Otherwise, it replaces a randomly selected string. To generate G3 group variants, InstruMate also leverages the variant makers Me and MeM, configured to change the app's name referenced in string resources.

(G4) Code Modifications

Code tampering can significantly impact an app's functionality, potentially introducing new features, altering existing behaviors, disabling security mechanisms, or exfiltrating sensitive user data. Such changes compromise the app's integrity and may lead to malicious outcomes. To simulate both static and dynamic code tampering, InstruMate was configured to modify the original app's code using state-of-the-art techniques that rely on static and dynamic instrumentation. Static instrumentation was performed using three black-box coverage engines: Androlog [18], Aspect-based [19], and ACVTool [22]. Each of these engines operates at different abstraction levels commonly found in the literature: the Jimple intermediate representation, the Java bytecode intermediate representation, or directly at the SMALI code level. Dynamic instrumentation is achieved by injecting the Frida engine in embedded mode [45] with predefined instrumentation that simply confirms the injection at run time. To produce G4 variants, variant makers Mand, Macv, MaspE, and simply applied, respectively, Androlog, ACVTool, and AspectJ-based instrumentation, all configured to produce method coverage. Variant maker MfriE injected Frida in embedded mode. These variant makers operate on per split basis, while variant makers MandM, MacvM, MaspEM, and MfriEM applied these techniques to the merged variants. The instrumentation was carefully configured to avoid altering any of the existing program's variables, serving only to capture coverage, in the case of static instrumentation, or, in Frida's case, to inject into the process and gain access to the app's internal memory.

Table 3.1 summarizes the modifications (features) that each variant is expected to have per group. Table 3.2 summarizes the current InstruMate's variant makers. In this

table, items with (*) indicate variant makers that leverage merging capability to produce a single merged APK.

Table 3.1: Features per group.

Group	Modifications (Features)
G1-Signature	Variants must have different signatures
G2-Manifest	Variants must have different signatures and also must have the debuggable flag turned on
G3-Resources	Variants must have different signatures and also must have changes in string resources
G4-Instrumentation	Variants must have different signatures and be modified to simulate code tampering

Table 3.2: Variant makers per group. Items with (*) merge split apps to produce single APK variants.

Group	Variant Makers
G1	Me, Mz, MeM*
G2	Me, MeM*
G3	Me, MeM*
G4	Mand, Macv, MaspE, MfriE, MandM*, MacvM*, MaspEM*, MfriEM*

3.3 Health Check Procedures

As shown in Figure 3.2, for each app or variant, the health check procedure consists of installing the app, granting the necessary permissions, and allowing it to complete its initial setup. The app is then launched and monitored over a time frame, during which **UI elements** and **exception-sites**—the exact points where exceptions are thrown—are recorded. This process can be repeated for an arbitrary number of iterations, with both the time window and the iteration count defined as configurable parameters. InstruMate uses UI automator to access the view representation and Logcat to monitor stack traces. When applied to the original apps, the user-defined number of pipeline executions establishes a baseline for comparison. When applied to the variants, it evaluates their health status by comparing the collected data with this baseline, subsequently determining their classification, which may result in one of the following outcomes.

- **Healthy and compatible with the baseline**: it means that no changes were observed between the original and variant versions of an app.

- **Healthy but incompatible with the baseline**: it means that a variant presents differences in terms of UI elements presented or exceptions thrown, but did not crash.
- **Faulty**: it means that the app crashed, failed to launch, or failed to function properly.

It is important to note that this procedure includes several checks not considered in previous research. Earlier studies considered a repackaged variant healthy if it could run without crashing for three seconds [22]. In fact, as described in Section 4.3, some Google Play Store apps require more than one minute to launch.

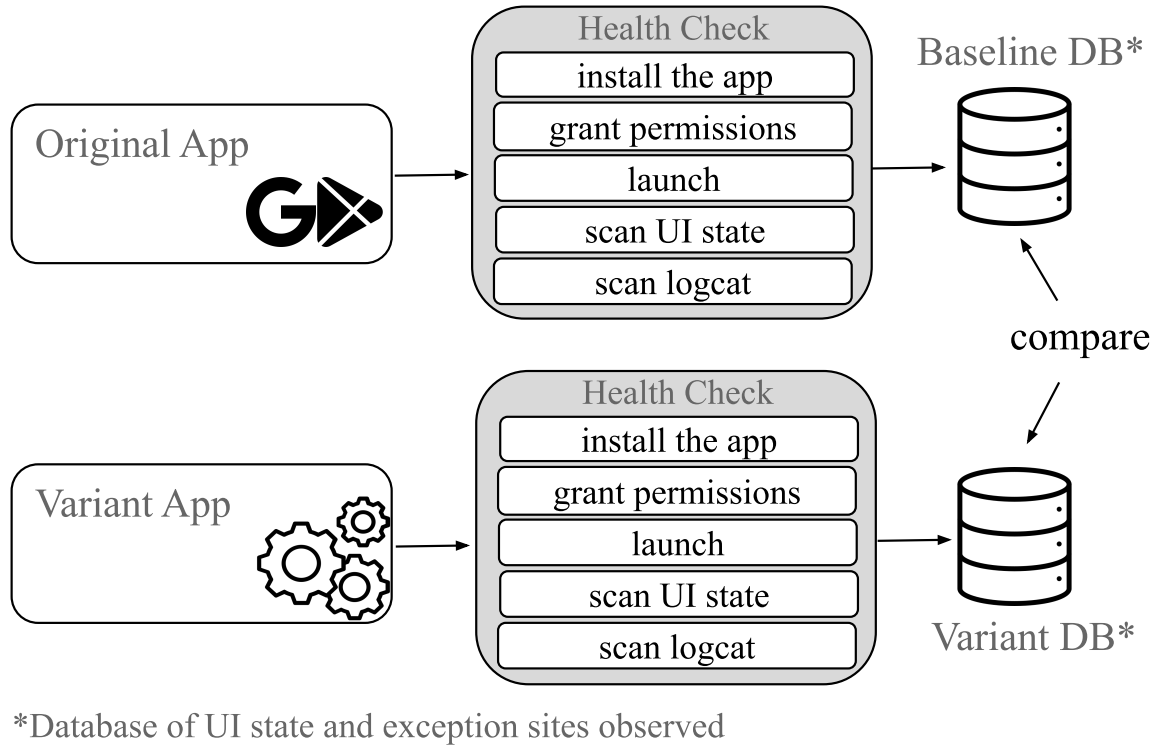


Figure 3.2: Baseline construction and health check pipeline.

3.4 Usage Example

InstruMate is a command-line framework that supports multiple operational modes, each designed to perform a specific task in the app analysis and instrumentation pipeline. The available modes are as follows:

- Install from Google Play (`install_from_gp`): This mode takes a list of app identifiers (IDs) and uses a properly configured, connected device with access to the

Google Play Store to obtain the apps. InstruMate automates the navigation and download of each specified app.

- Download from Device (`download_device_apps`): Given a list of app IDs, this mode retrieves all corresponding installation packages directly from the connected Android device.
- Analyze Mode (`analyze`): This mode performs static analysis on a provided set of directories, where each directory contains the installation package of an app. The output is a detailed static analysis report. No instrumentation is applied in this mode.
- Instrumate Mode (`instrumate`): In this mode, the framework first performs static analysis, then applies the instrumentation process, followed by a second round of static analysis—this time on the instrumented apps.
- Health Check Mode (`healthcheck`): This mode receives a set of applications as input, executes health check procedures, and generates corresponding diagnostic reports.
- Create Databases (`create_databases`): This mode processes previously generated reports to build a consolidated database, facilitating comparison and further analysis across multiple apps.

Figure 3.3 presents other options provided by the framework. Figure 3.4 presents the output of InstruMate's static analysis phase for a single app. For each app, the tool generates a structured folder that includes the installation files, a comprehensive app report (stored in `app.json`), and individual reports for each compiled binary file. The results of the content-type analysis are provided in a CSV file, listing each file along with its hash and the identified content type. Similarly, Figure 3.5 depicts how exception sites are reported by InstruMate. This figure also exemplifies a scenario of repackage-proofing, as discussed in Section 5.2. Figure 3.6 depicts InstruMate coordinating parallel health check operations on different apps, each executed within its own emulator instance.

3.5 Chapter Summary

This chapter introduces InstruMate, a repackaging infrastructure designed to generate and evaluate controlled variants of Android apps. InstruMate separates the unpacking/repacking stages from instrumentation tasks, enabling the integration of specialized tools for each phase. Its core component, the *Variant Maker*, orchestrates the creation of repackaged variants based on specified modification groups: signature changes (G1),

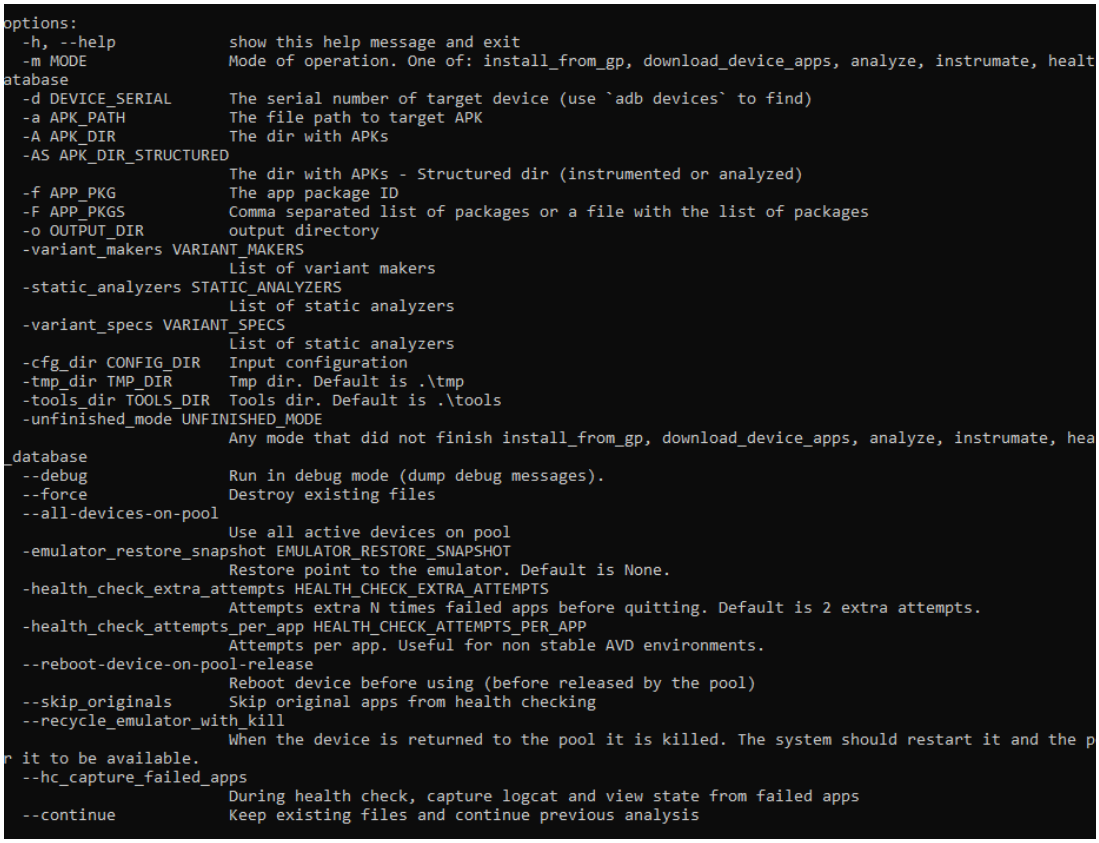


Figure 3.3: InstruMate options.

manifest file alterations (G2), resource modifications (G3), and simulated code tampering via instrumentation (G4). The infrastructure includes a comprehensive static analysis stage that extracts detailed metadata using tools such as Androguard, Apache Tika, and Radare2, facilitating informed and traceable modifications. To ensure robustness, multiple variant makers are employed—some operating at the APK level, while others merge split APKs to create unified packages. Code instrumentation in the G4 group leverages both static (e.g., Androlog, ACVTool, AspectJ) and dynamic (e.g., Frida) techniques. Following variant creation, InstruMate performs automated health checks to evaluate runtime behavior. This includes app installation, permission granting, execution monitoring, UI element inspection, and exception-site tracking, all benchmarked against a baseline constructed from the original app. Variants are then classified as healthy and compatible with the base line (no differences are observed), healthy but incompatible (behaviorally divergent), or faulty. This multi-stage process, incorporating both fine-grained analysis and execution monitoring, addresses limitations in previous work by adopting stricter health criteria.

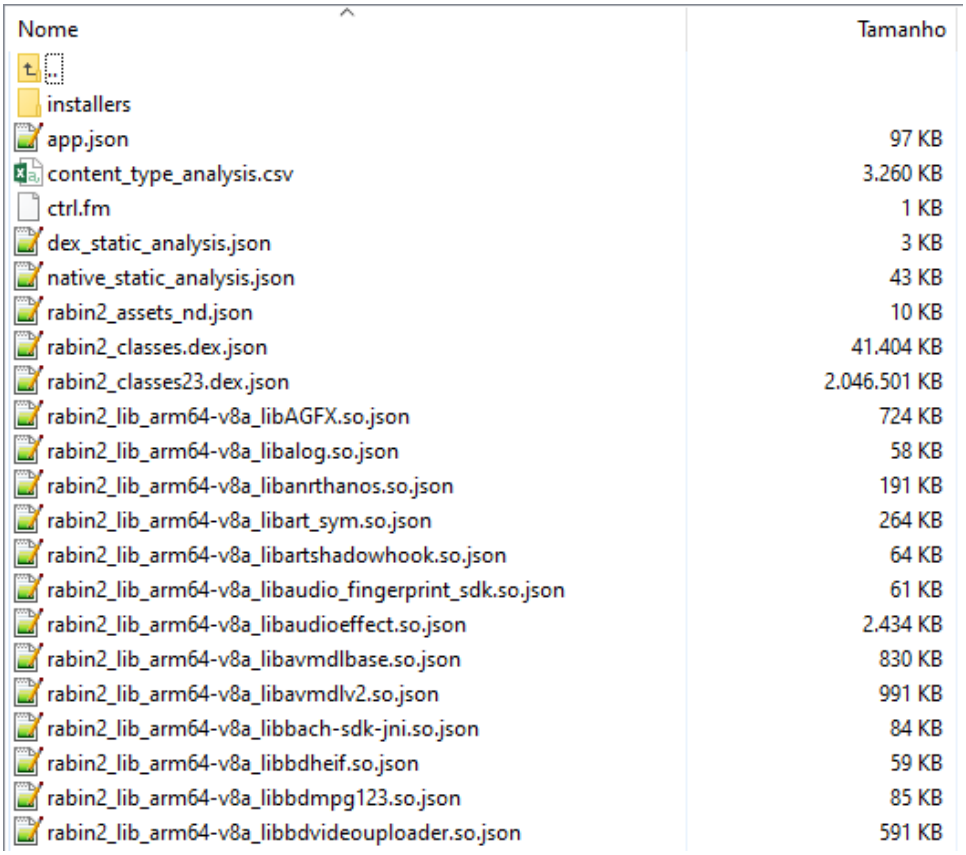


Figure 3.4: Presents InstruMate's output.

```
{
    "exception_name": "java.lang.RuntimeException",
    "method_signature": "android.os.BinderProxy.sendDeathNotice",
    "detail": "BinderProxy.java:726",
    "index": 3,
    "raw_msg": "12-28 23:57:11.258  5091  5162 E WM-WorkerWrapper: \tat android.os.BinderProx
},
{
    "exception_name": "java.lang.NullPointerException",
    "method_signature": "com.microsoft.office.plat.archiveextraction.c.b",
    "detail": "Unknown Source:6",
    "index": 0,
    "raw_msg": "12-28 23:58:07.793  5091  5091 W System.err: \tat com.microsoft.office.plat.a
},
{
    "exception_name": "java.lang.NullPointerException",
    "method_signature": "com.microsoft.office.plat.archiveextraction.a.d",
    "detail": "Unknown Source:2",
    "index": 1,
    "raw_msg": "12-28 23:58:07.793  5091  5091 W System.err: \tat com.microsoft.office.plat.a
},
{
    "exception_name": "java.lang.NullPointerException",
    "method_signature": "com.microsoft.office.plat.assets.OfficeAssetsManagerUtil.fAssetsFile
    "detail": "Unknown Source:4",
    "index": 2,
    "raw_msg": "12-28 23:58:07.793  5091  5091 W System.err: \tat com.microsoft.office.plat.a
},
{
    "exception_name": "java.lang.NullPointerException",
    "method_signature": "com.microsoft.office.plat.OfficeAssetManager.getAssetFileLoc",
    "detail": "Unknown Source:82",
    "index": 3,
    "raw_msg": "12-28 23:58:07.793  5091  5091 W System.err: \tat com.microsoft.office.plat.O
},
```


Figure 3.5: Presents a portion of the heatlh check procedures result containing a description of the exception-sites. This output is also mentioned at Section 5.2.

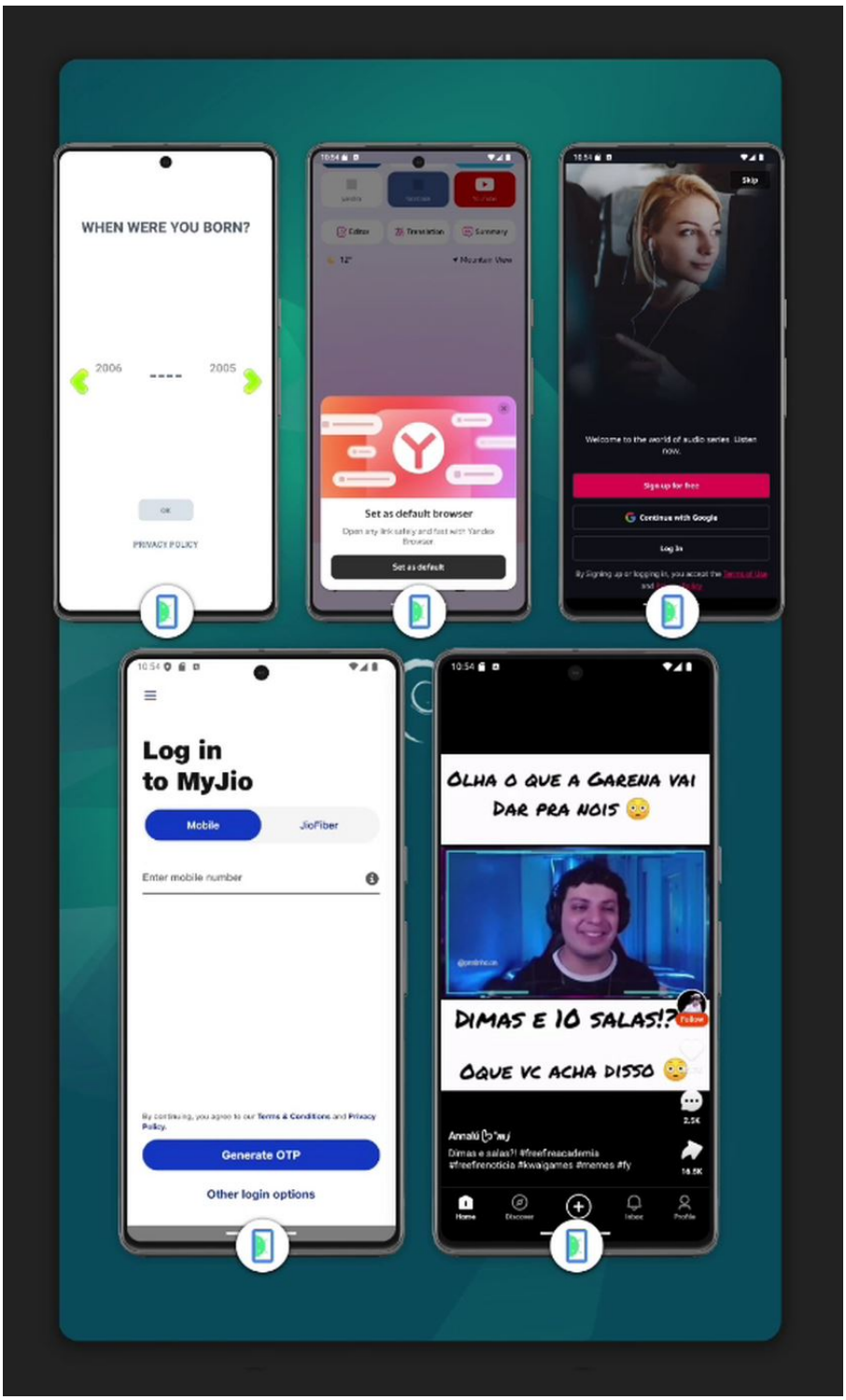


Figure 3.6: Presents InstruMate orchestrating five emulators during the health check procedures.

Chapter 4

Empirical Assessment

According to Khanmohammadi *et. al.*[8] and Li *et. al.*[11], popularity is the main criterion for an app being chosen for repackaging by malicious actors. Popular apps, by definition, have large user bases, significantly amplifying the potential impact of malware distribution. Additionally, users are more likely to install apps they already recognize and trust, making popular apps desirable targets for monetization through repackaged variants. At the same time, such apps might be more likely to incorporate advanced security mechanisms, as developers of widely used software have stronger incentives to implement protective measures.

This chapter presents the design of the empirical assessment. Sections 4.1 to 4.7 outline the objectives of the experiment and describe the curated dataset constructed to address these objectives. The specific procedures employed in the study are also explained in detail in these sections.

4.1 Goal, Questions, and Metrics

The primary goal of this empirical study is to employ InstruMate to evaluate the extent to which popular Android apps can be successfully repackaged, as well as to examine the feasibility of detecting repackaging failures. To guide our analysis, we formulate the following research questions.

(RQ1) How susceptible are popular Android apps to repackaging?

(RQ2) What are the common defenses Android apps leverage against repackaging?

(RQ3) What are the root causes that lead to repackaging failures?

The primary metric we use to address these questions is an indicator of whether a given variant maker successfully produced a repackaged variant. From this measure, we derive

the total number of apps that are not resilient to repackaging, addressing RQ1. We also derive (a) the number of successfully built variants per variant maker, (b) the number of healthy variants per variant maker, (c) the number of faulty variants per variant maker, and (d) the number of healthy but incompatible variants per variant maker. Finally, in this work, we introduce the Healthy Variant Gap (HVG). This metric captures the number of healthy variants a given variant maker failed to produce, even though others in the same group succeeded.

This metric is primarily employed to address research questions RQ2 and RQ3. In particular, instances where one strategy fails while others succeed provide valuable insights for further investigation into potential underlying bugs, whereas scenarios in which no strategy succeeds serve as indicators of possible defense mechanisms embedded within the app.

4.2 Experiment Overview

Figure 4.1 shows an overview of the experiment in six steps. An initial dataset comprising the 500 most popular apps from the Google Play Store was selected. In the first step, apps unsuitable for the study were excluded. The remaining apps underwent health check procedures, during which 200 observations were conducted; apps that failed at least once were removed, resulting in a curated dataset (step 3). From this curated set, repackaged variants were generated (step 4). Each variant was then subjected to three rounds of health checks (step 5), and the results were compared against those of the original apps (step 6).

The rigor imposed by the 200 observations ensures that the dataset consists of bug-free apps, such that any subsequent failure can be reliably attributed to the repackaging procedures, as detailed in Section 4.3.

The procedure used to compare the variants and classify the apps in the dataset as resilient or non-resilient to repackaging is detailed in Section 4.4.

The health check procedures applied to the variants were configured as detailed in Section 4.5. Sections 4.6 and 4.7 explain why we decided not to use stress testing in our experiments and the environment configuration we used to execute them.

4.3 Dataset Curation Procedures

To curate the dataset used in our study, we first selected the 500 most popular apps from the Google Play Store, as reported by the Android Rank project [86] on October 21, 2024. Subsequently, we excluded apps that are often pre-installed by vendors (e.g.,

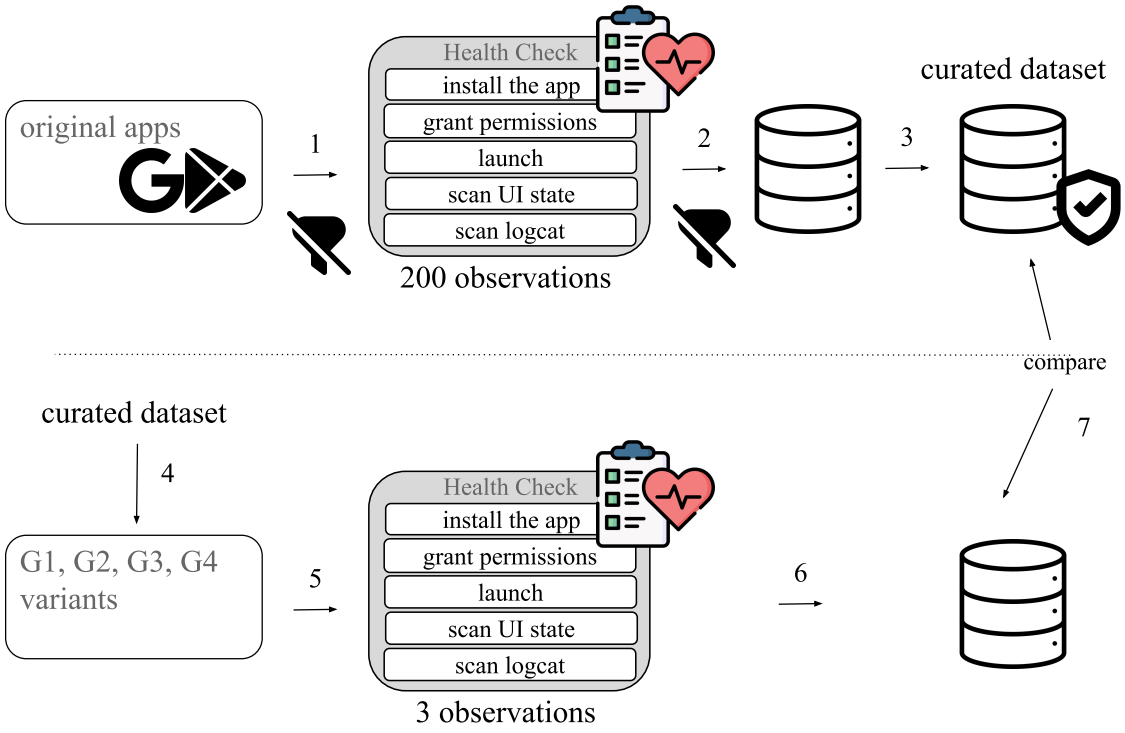


Figure 4.1: Experiment Setup

default apps from Google and Samsung) [1]. The rationale for this exclusion criterion is that this category of apps is pre-installed on numerous devices as part of the vendor firmware, meaning that the end user typically cannot uninstall them. Furthermore, attempting to repackage these apps would require a mandatory change to the app's package ID so that it could be installed on the testing devices that contain preexisting installations, a modification we deliberately sought to avoid to ensure consistent conditions across all tested apps. In addition, repackaging system apps presents a fundamentally different scenario, as discussed in Subsection 2.3.2. The remaining apps were subjected to the health check procedures (see Section 3.3). We configured our health checker to observe each app for a one-minute time window and record 200 observations. Only the original apps that completed all 200 iterations without crashing were included in the final dataset.

It is important to mention that the filtering applied was a deliberate choice rather than the outcome of a technical limitation. Robust repackage-proofing techniques [53, 14] commonly embed a network of integrity checkers (guards) that continuously monitor the integrity of the app. These mechanisms are designed to induce failures at points distant from the actual detection site, thereby manifesting as sporadic malfunctions or random buggy behaviors. To establish a reliable baseline, each app was executed 200

[1] Removed packages initiated with `com.android`, `com.sec`, `com.google`, `com.samsung`, `com.lenovo`, `com.motorola`, `com.miui`, `com.xiaomi`, `com.vivo` and `com.mi`

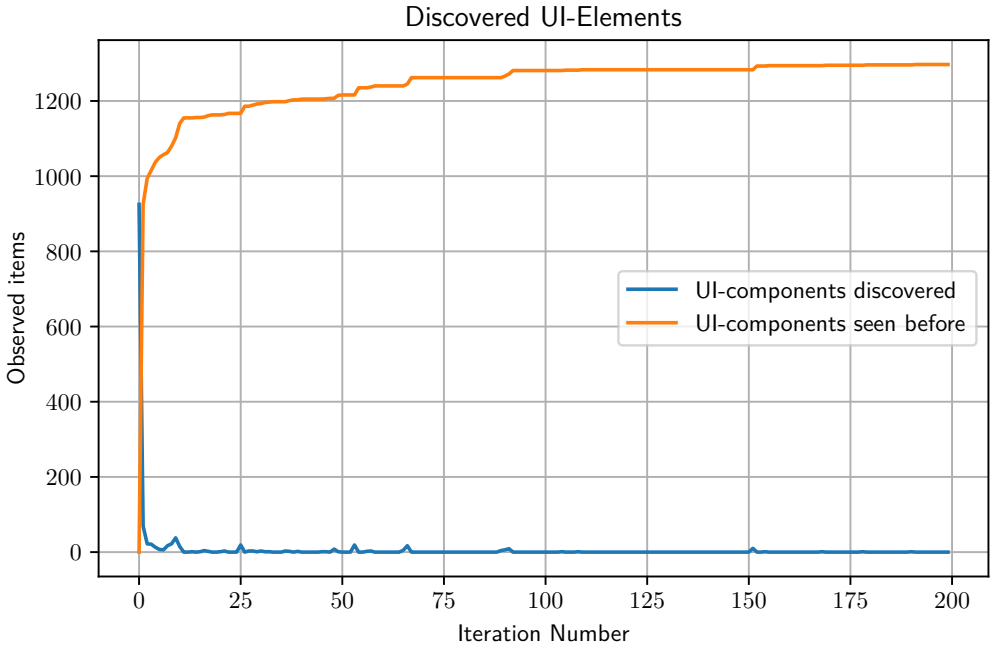
times, and any original app that exhibited bugs or crashes upon launch was excluded from the dataset. This procedure ensured that the dataset was composed exclusively of stable apps, thereby increasing confidence that any deviations observed in the repackaged variants could be directly attributed to the repackaging process.

As a result of the filtering process, nine apps remained in a loading state without resolving within one minute time frame. Three apps were identified as background services and therefore could not be directly launched. Nineteen apps redirected execution to other apps upon launch, representing a flow that could not be accommodated in the experimental setup. In addition, 105 apps were classified as firmware apps, while 208 failed to complete 200 iterations without crashes—of which 103 were games. The latter case is noteworthy, as games often impose specific hardware requirements that are difficult to emulate, which likely contributed to their sporadic failures. After applying these filtering criteria, a total of **156 apps** remained stable and did not crash during any of the 200 observation cycles. Consequently, they were included in our final dataset. Table 4.1 characterizes the dataset by market categories. From the remaining 156 stable apps, 43 were games, making this still the largest category (Table 4.1).

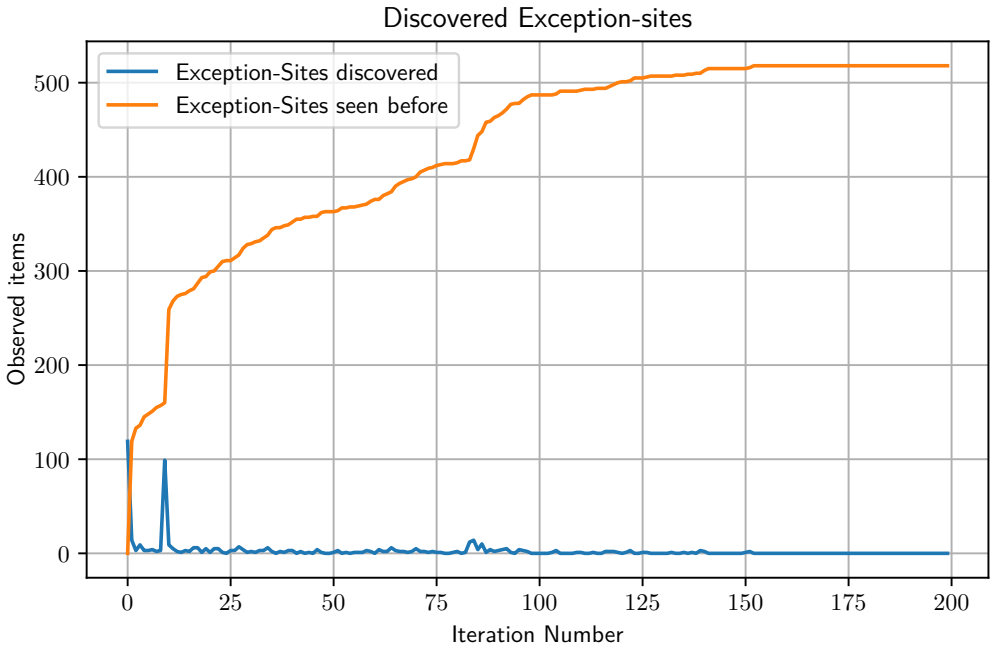
Figure 4.2 shows the UI elements and exeptions discovery process observed during the generation of the **baseline** for health check assessment, established by the original versions. In the top chart, the plot illustrates the relationship between the number of new UI elements discovered and the total number of cumulative UI elements observed. Similarly, in the bottom chart, the plot shows the discovery of new exception sites and the cumulative number of exception sites previously observed. This saturation process shows the completeness of the baseline, indicating that a sufficient number of observations have been conducted to capture all original apps' UI elements and exception sites during the 200 observations. Since each of the 200 observations lasts one minute, the total observation time per app exceeds three hours and is aligned with the practice of previous studies [87, 88].

Table 4.1: Distribution of the dataset by category. All values for average, smallest, and largest app are in megabytes.

Category.	**Avg**	**Smallest**	**Largest**	**#Apps**
Games	273	27	3.263	43
Productivity & Tools	111	4	370	33
Social & Communication	103	2	245	24
Media & Entertainment	101	6	261	20
Finance & Shopping	72	19	150	16
Travel & Navigation	117	30	282	10
Health & Lifestyle	89	67	124	5
Education	90	17	182	5



(a) UI-componentes saturation



(b) Exception-sites saturation

Figure 4.2: Saturation of discovered UI elements and exception sites over increasing iterations.

4.4 Classification of Original Apps Based on Repackaged Variants

The proposed experiment utilizes multiple variant makers to produce redundant variants—for instance, Me and MeM generate split and merged variants of the same app—thereby aiming to increase the likelihood of obtaining at least one healthy variant. Likewise, the use of various instrumentation techniques follows a similar strategy that is designed to increase the probability of obtaining at least one code-tampered variant. After evaluating each variant created by all variant makers, the entire app is classified as **non-resilient** to repackaging if at least one variant is found to be healthy and compatible with the baseline.

Conversely, the app is considered **resilient or divergent** if none of the variant makers could generate a healthy variant or if the variants exhibit sporadic failures, introduce new exception sites, or display unseen UI elements. This indicates that the repackaging procedures applied to a particular app have resulted in unintended behavioral differences from the original one. The final set of apps classified as non-resilient to repackaging is used retrospectively to guide the identification of possible repackaging failures in individual variant makers that failed to process them, while others succeeded.

It is also important to clarify that the successful generation of a healthy variant—despite the inherently positive connotation of the term—actually reflects a negative outcome from a security standpoint, as it indicates that the original app is non-resilient to repackaging.

4.5 Health Check Procedures for Variants

Since the selected instrumentation techniques may introduce additional code, resulting in increased execution overhead, InstruMate was configured to monitor variants for a time frame twice as long as its original counterpart. Specifically, each variant was observed for **two minutes** to allow any existing security mechanism to activate. In addition, each variant underwent the health check procedures three times to ensure that any potential security mechanisms requiring a restart of the app could be properly triggered.

4.6 A Note on Stress Testing

During the establishment of the baseline for the original apps, each app underwent stress testing using Monkey [89] to evaluate whether such testing would contribute meaningfully to the experiment. Preliminary results indicated that stress testing could lead to

various app behaviors, such as crashes, unexpected closures, or unintended navigation to other apps. To accurately distinguish these behaviors from those potentially caused by a reaction to repackaging or tampering, it would be essential to extend InstruMate to capture not only UI elements and exception sites but also correlate these events with the input sequences generated by the testing engine. Therefore, excluding integration with a stress testing engine is considered a favorable initial configuration for our experiment, as it helps eliminate the aforementioned noise. Moreover, the following considerations further support the relevance of the experiment as designed: 1) If a repackaged app successfully launches, it already poses a significant threat, as the variant could be exploited to distribute malware or exfiltrate sensitive data; 2) The current health check procedure provides a uniform input—the launch command—and aims to exercise each app's main Activity. Future research could explore the impact of stress testing on identifying failures and exposed defenses, using the simple launch as a foundational starting point.

4.7 Execution Environment

Each original app and the produced variants are tested on a freshly wiped emulator device with no prior usage. The emulator is configured as a Google Pixel 7 Pro AVD, with a portrait orientation, running Android 15 (API 35, x86_64), non-rooted, equipped with 20 GB of RAM and 20 GB of internal storage.

4.8 Chapter Summary

To empirically assess app resilience to repackaging, the study curated a dataset starting from the 500 most popular Android apps on the Google Play Store, as identified by the Android Rank project. Pre-installed vendor apps were excluded to ensure installation consistency and avoid altering package identifiers. Health check procedures were applied to all remaining apps using a one-minute observation window repeated across 200 iterations. Apps that crashed, failed to initialize, or triggered specific behaviors—such as redirects to the Play Store—were filtered out, resulting in a final dataset of 156 stable apps. Baseline completeness was validated through saturation analysis of UI elements and exception sites. Each app was subjected to repackaging via multiple variant makers, producing variants with modifications to signatures, manifest files, resources, and code. Health checks were then conducted on each variant for two minutes across three iterations, allowing sufficient time for latent defenses to activate. An app was classified as non-resilient if at least one variant maintained behavioral consistency with the original; otherwise, it was deemed resilient or divergent. Stress testing was deliberately excluded to

avoid noise and preserve attribution of failures to repackaging rather than random input. All tests were conducted on a consistent and emulator environment configured as a clean Google Pixel 7 Pro running Android 15, ensuring uniformity across executions.

Chapter 5

Results

We executed InstruMate on the 156 original apps, resulting in the successful generation of 1,693 variants, which corresponds to 72% of the expected total of 2,340. A significant portion of the creation failures was concentrated in the G4 group. Following their creation, each variant was subjected to the health check procedures described in Section 4.5.

A detailed summary of both the creation results and the subsequent verification results is presented in Table 5.1. Column **B** indicates the number of successfully built variants per variant maker, while column **H** represents the number of healthy variants. Similarly, column **F** denotes the faulty variants, and column **I** contains the number of healthy but incompatible variants. The data on the success rates of other variant makers in producing healthy repackaged apps in the same group serve as the basis for calculating the **HVG** column (**Healthy Variant Gap**). This column quantifies the number of healthy variants that a specific variant maker failed to produce while others in the same group succeeded. At the end of each group in Table 5.1, a special row indicates the number of apps that were built by all variant makers and classified as healthy and compatible with the baseline.

As a general assessment of the results presented in this Table 5.1, it is clear that the build process becomes significantly more problematic when instrumentation is applied.

The Sections 5.1 to 5.3 present findings related to the research questions. As discussed in Section 5.2 (RQ2), 17 apps exhibited defense mechanisms triggered either by signature modification or by the instrumentation process itself. The results also indicate that changes performed prior to code instrumentation tend to present fewer implementation challenges, whereas instrumentation introduces substantial complexity.

Table 5.1: Summary of health check results per variant maker. Columns: B (Built Variants), H (Healthy Compatible), F (Faulty), I (Healthy but Incompatible), HVG (Healthy Variant Gap). All percentage values are calculated with respect to the total of 156 apps.

Group	Variant Maker	B		H		F		I		HVG	
G1	Me - ApkEditor	156	100%	132	85%	21	13%	3	2%	2	1%
	Mz - Naive Zip	156	100%	117	75%	36	23%	3	2%	17	13%
	MeM - ApkEditor, Merged	156	100%	129	83%	25	16%	2	1%	5	4%
	Me + Mz + MeM			134	86%						
G2	Me - ApkEditor	130	83%	120	77%	10	6%	0	0%	7	6%
	MeM - ApkEditor, Merged	134	86%	127	81%	7	4%	0	0%	0	0%
	Me + MeM			127	81%						
G3	Me - ApkEditor	130	83%	125	80%	5	3%	0	0%	5	4%
	MeM - ApkEditor, Merged	134	86%	129	83%	5	3%	0	0%	1	1%
	Me + MeM			130	83%						
G4	Mand - Androlog	78	50%	45	29%	30	19%	3	2%	57	56%
	Macv - ACVTool	111	71%	24	15%	85	54%	2	1%	78	76%
	MaspE - ApkEditor, AspectJ	65	42%	14	9%	43	28%	8	5%	88	86%
	MfriE - ApkEditor, Frida	126	81%	51	33%	74	47%	1	1%	51	50%
	MandM - ApkEditor, Merged, Androlog	16	10%	9	6%	7	4%	0	0%	93	91%
	MacvM - Merged, ACVTool	106	68%	25	16%	81	52%	0	0%	77	75%
	MaspEM - ApkEditor, Merged, AspectJ	65	42%	14	9%	44	28%	7	4%	88	86%
	MfriEM - ApkEditor, Merged, Frida	130	83%	51	33%	79	51%	0	0%	51	50%
	Mand + Macv + ... + MfriEM (all G4)			102	65%						

5.1 (RQ1) How susceptible are the apps to repackaging?

We successfully built all variants using the **G1 group** of variant makers. Among these, Me produced 132 *healthy and compatible* variants, 13 *faulty* variants, and one *healthy but incompatible* variant, achieving a higher success rate compared to the variant makers Mz and MeM. Still focusing on the G1 group, out of the 156 apps, 134 (86%) were classified as **non-resilient** to signature modification. Note that this information is presented in Table 5.1, in the last row of the G1 group, and is obtained by counting the number of apps for which at least one healthy and compatible variant was successfully generated. Since signature modification is a prerequisite for producing variants from G2, G3, and G4 groups, these 134 apps, marked as non-resilient in G1, were selected for further evaluation by the other groups.

Concerning Manifest File Modifications (**group G2**), the variant makers Me and

MEM successfully built 130 and 134 variants, respectively. Among these, ME produced 120 *healthy and compatible* variants and 10 *faulty* ones, while MEM produced 127 *healthy and compatible* and seven *faulty* variants. MEM achieved a 86% success rate in producing *healthy and compatible* variants in group G2. No variant generated in group G2 was classified as *healthy but incompatible*.

We observed similar results for **group G3**, where neither ME nor MEM produced any *healthy but incompatible* variants. ME and MEM together achieved a 83% success rate in producing *healthy and compatible* variants in the group G3. Out of the 134 apps, 127 (81%) apps were classified as **non-resilient** to modifications in the G2 group and allowed execution in *debug mode*, and 130 apps (83%) were classified as **non-resilient** to modifications to resources stored in string files.

Finding 1. The lower HVG observed for MEM in the G2 and G3 groups suggests that minor decoding issues were mitigated by the merging process.

Finally, the variant makers in **group G4** demonstrated the lowest overall success rates, accompanied by high variability. Within this group, MFRIE and MFRIEM achieved the highest numbers of *healthy and compatible* variants, each producing 51. They were followed by MAND, which generated 45 such variants. MASPE and MASPEM were the lowest ranked, each producing 14 *healthy and compatible* AspectJ-instrumented variants.

No clear benefit was observed from the application of the merging process; conversely, its application in the case of Androlog resulted in decreased performance, with the number of *healthy and compatible* variants dropping to just nine. MACV and MACVM exhibited high construction rates; however, a substantial proportion of the repackaged variants they produced were faulty—54% and 52%, respectively. On average, within the G4 group, the success rate for generating repackaged variants was 56%, while the success rate for producing *healthy and compatible* variants dropped to **19%**. These numbers are significantly lower than those reported in the literature [18, 22, 20, 23].

Finding 2. Modifications performed before code instrumentation tend to present fewer implementation challenges. In contrast, code instrumentation techniques encounter difficulties not only during the creation process but also frequently result in variants that malfunction or fail to launch altogether.

However, such a reduction was expected, since our dataset contains complex split apps—an app type against which these tools had not been previously evaluated. Moreover,

the health check procedures introduced an additional layer of rigor, imposing stricter validation criteria that further reduced the number of resulting *healthy and compatible* variants. It is important to emphasize that the instrumentation applied in this study was deliberately kept simple. This suggests that more invasive instrumentation techniques are likely to yield even lower success rates. Of the 156 apps in the initial dataset, 102 (65%) were classified as **non-resilient** to code modifications.

Finding 3. No single instrumentation approach proved universally effective. Among the most successful *variant makers*, Frida-based instrumentation with merging (MFriEM) and Androlog-based instrumentation without merging (Mand) yielded better results, each producing 33% and 29% of healthy variants.

Finding 4. Code instrumentation often fails during the variant creation process or produces non-functional variants that cannot be launched. However, when multiple techniques are combined, instrumentation succeeded in 65% of the apps, suggesting that employing diverse methods can maximize the number of successfully instrumented variants.

5.2 (RQ2) Common Defenses Against Repackaging

During the experiments with variant makers in Groups G1 and G4, the following repackaging defenses were observed.

Signature Modification (G1)

Variants derived from two apps (Microsoft OneNote and Microsoft 365) displayed the message *"This app may not be from a trusted source"* and variants derived from one app (CamScanner) displayed the message *"This copy is not genuine"* (Figure 5.1). A manual inspection revealed a recurring pattern of exception sites originating from the same shared library (`libplat.so`) across five Android apps: Microsoft Excel, Microsoft 365, Microsoft OneNote, Microsoft PowerPoint, and Microsoft Word. Some variants of these apps displayed a modified user interface, while others simply closed upon launch. However, all variants exhibited exception sites that were traced back to `libplat.so`, suggesting that this library implements a recurring mechanism to prevent repackaging. Other three non-healthy variants (apps: Carrom Pool Disc Game, Roblox and Beach

Buggy Racing) produced IO exceptions[1] that were traced back to `libpairipcore.so`, which is publicly known to be a repackage-proofing solution[2]. One faulty variant (app: HP Printer Setup) reported no new exception site, but simply invoked `System.exit`, after loading the library `libfusg.so`. There is no public information about `libfusg.so`, however, InstruMate's static analysis found string symbols possibly related to repackaging defenses, and that could be revealed in certain circumstances: *"Stop poking around my nether bits you cretin!"*, *"And you keep wondering why nobody likes you?"* and *"Go pound bits!"*. Table 5.2 provides a detailed information of the exception-sites which contributed to detecting repackaging defense mechanisms. It is important to mention that each exception-site was thoroughly examined through manual inspection, as detailed in Section 5.4.

Table 5.2: Exception Sites in the G1 group linked to repackage defenses.

ID	Exception-sites
com.microsoft.office.excel	{('java.lang.NullPointerException', 'com.microsoft.office.plat.archiveextraction.c.b')}
com.microsoft.office.officehubrow	{('java.lang.NullPointerException', 'com.microsoft.office.plat.archiveextraction.b.b')}
com.microsoft.office.onenote	{('java.lang.NullPointerException', 'com.microsoft.office.plat.archiveextraction.c.b')}
com.microsoft.office.powerpoint	{('java.lang.NullPointerException', 'com.microsoft.office.plat.archiveextraction.c.b')}
com.microsoft.office.word	{('java.lang.NullPointerException', 'com.microsoft.office.plat.archiveextraction.c.b')}
com.miniclip.carrom	{('java.io.IOException', 'com.pairip.VMRunner.readByteCode')}
com.roblox.client	{('java.io.IOException', 'com.pairip.VMRunner.readByteCode')}
com.vectorunit.purple.googleplay	{('java.io.IOException', 'com.pairip.VMRunner.readByteCode')}

Code Modifications (G4)

Variants derived from one app (Game Pooking Billiards City) generated new UI elements displaying intimidating messages that prompt the user to grant the app permission to capture sensitive data (*"Allow Billards-mate to access all device logs"*) and variants derived from another app (MyJio) exposed an indication of repackaging detection, showing the message *"Rooted device detected"*, despite the fact that the emulator device was not rooted (Figure 5.2). Other variants derived from two apps (Score! Hero and YouCam Makeup)

[1] `java.io.IOException` **at** `c.p.VMRunner.readByteCode`
[2] https://github.com/Solaree/pairipcore

presented the messages *"Your device may not be compatible"* and *"Your installation is corrupted"* and prompted the user to reinstall the app. One exception site[3] appeared in three non-healthy variants. This exception site was traced back to the **Google Dynamite library**, which forced the app to restart. In such cases, the Logcat message *"Module config changed, forcing restart due to module ads.dynamite"* was observed. Although public information about this proprietary library is limited and does not confirm whether the observed behavior is intentional, it effectively prevented the completion of the health check procedures on the generated variants. Table 5.3 provides a detailed information of the exception-sites which contributed to detecting repackaging defense mechanisms. Each exception-site was thoroughly examined through manual inspection, as detailed in Section 5.4.

Table 5.3: Exception Sites in the G4 group linked to repackage defenses.

ID	Exception-sites
com.myntra.android	{('java.lang.VerifyError', 'com.cyberfend.cyfsecurity.CYFMonitor.a'), ('java.lang.Exception', 'com.appsflyer.internal.AFg1xSDK.AFInAppEventType'), ('android.os.DeadObjectException', 'm.ct.a'), ('java.lang.ExceptionInInitializerError', 'com.appsflyer.internal.AFg1xSDK.AFInAppEventType'), ('java.io.IOException', 'm.ct.a')}
com.utorrent.client	{('android.os.Parcel.readException', 'asdf.a'), ('java.lang.VerifyError', 'Companion.scheduleTimeout'), ('android.os.Parcel.readException', 'armj.a'), ('android.os.DeadObjectException', 'm.ct.a'), ('android.os.Parcel.readException', 'arrj.a'), ('java.io.IOException', 'm.ct.a')}
com.vkontakte.android	{('android.os.DeadObjectException', 'm.ct.a'), ('java.lang.UnsatisfiedLinkError', 'J.N.M6xubM8G'), ('java.io.IOException', 'm.ct.a')}

Finding 5. The results of our assessment reveal recurrent approaches that prevent the execution of repackaged apps, produce user interface alerts, trigger privilege-escalation attempts, or induce restart loops.

[3] `java.io.IOException` at `m.ct.a`

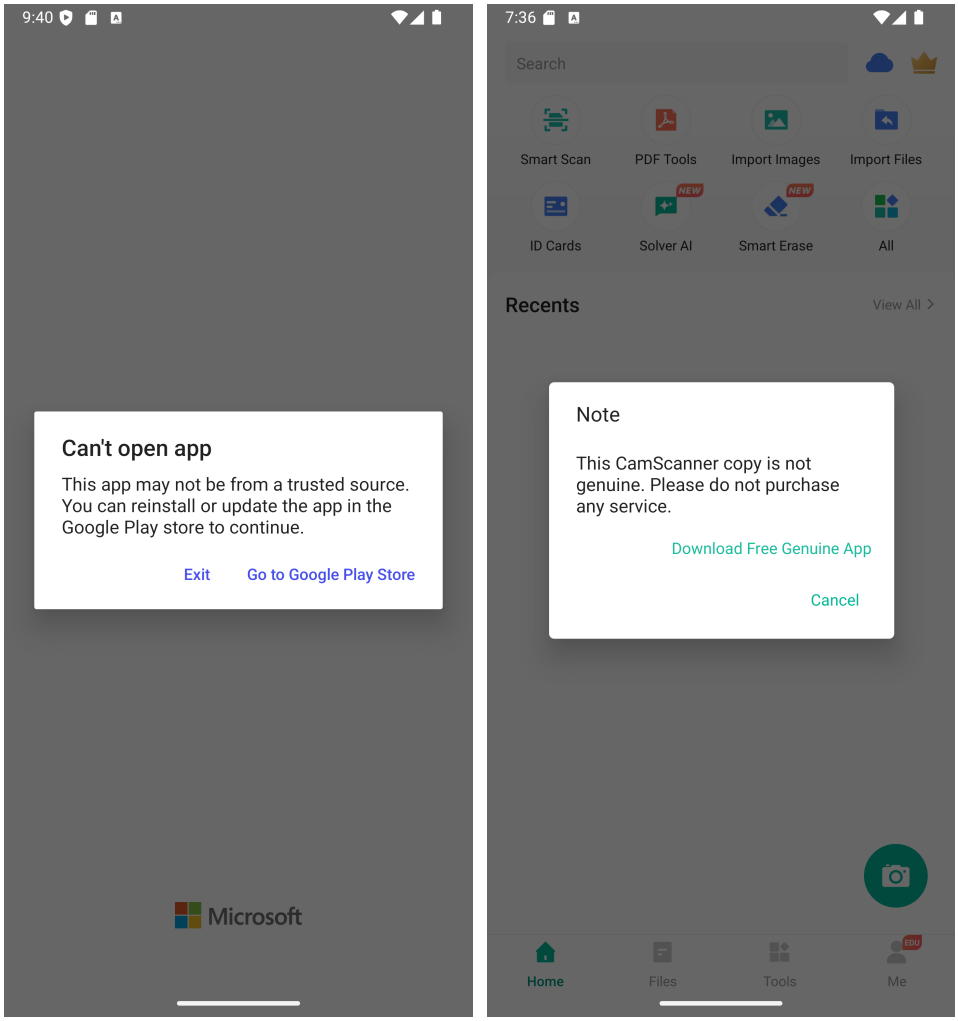


Figure 5.1: Messages observed only in variants (group G1).

5.3 (RQ3) Root Cause of Failures

In addition to repackage-proofing measures, we also identified other likely root causes of repackaging failures. As shown in Table 5.1, Mz (the naive ZIP-based variant maker) participated in the generation of 117 healthy and compatible variants, but 134 were classified as non-resilient to signature modifications. To account for the increase to 134, other variant makers successfully generated healthy and compatible variants for apps that Mz could not produce. This gap is represented in the HVG column of Table 5.1. Variant maker Mz failed to generate variants because certain files within the APKs are required to be stored uncompressed in the ZIP archive. An example is the shared object file, which has the `.so` extension, and is read directly from the APK and mapped to memory for execution. Analysis of Mz's HVG revealed IO exceptions during access of files directly stored in the APK, expecting them to be uncompressed and ready to be mapped to RAM memory at runtime, a technique which is particularly useful for large apps, such as games.

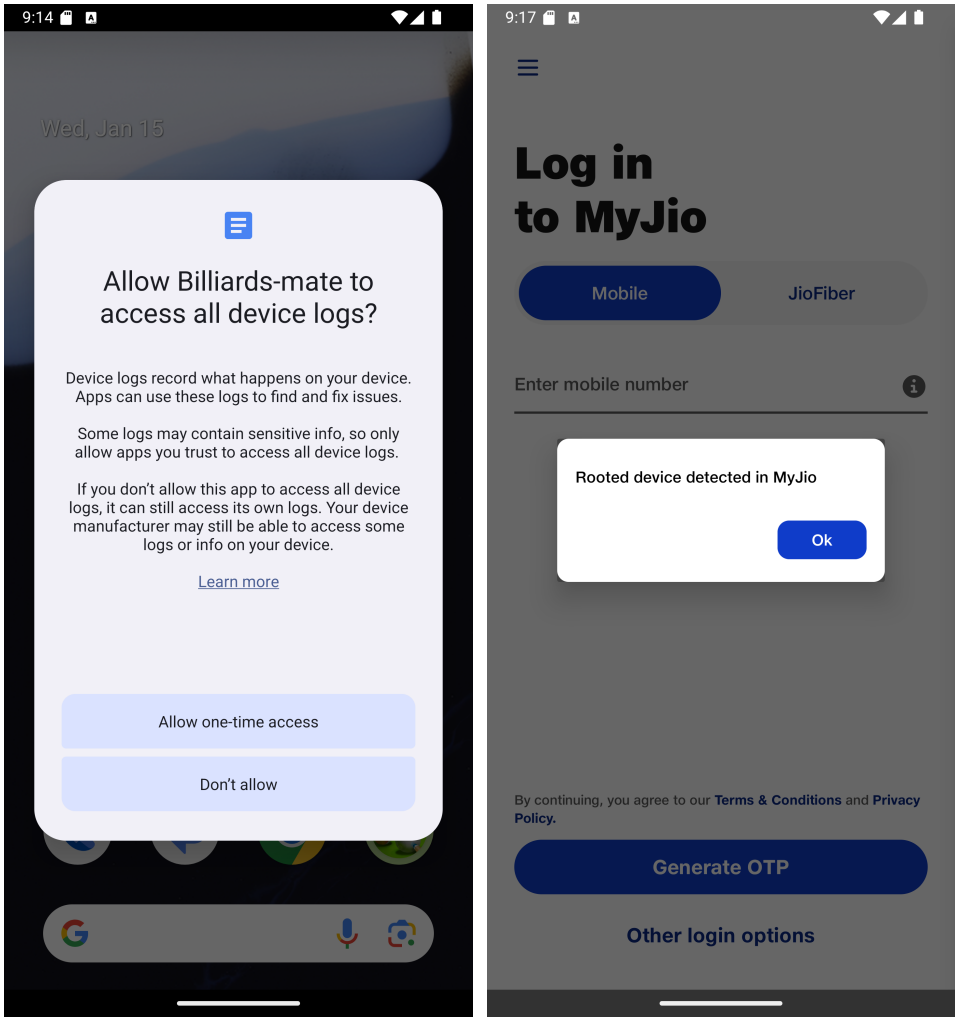


Figure 5.2: Messages observed only in variants (group G4).

With respect to code instrumentation, some variant makers failed during the build process, and others generated APKs that could not be installed. While many of the issues require in-depth, app-specific investigation—beyond the scope of this experiment—some failures were consistent and could be isolated. For instance, failures affecting variant creation for 33 apps were traced to potential issues such as concurrency bugs and misinterpretations of DEX opcodes in the Soot engine, which is used by Androlog. ACVTool failed to instrument certain methods due to register management issues, which triggered a Python `KeyError` in its register map. Additionally, it encountered a "list index out of range" error in `insn3rc.py`, suggesting the presence of a potential bug. Dex2jar failed to generate the expected JAR file from the provided DEX file for 26 apps, while Frida, in embedded mode, encountered issues when loading pre-existing shared objects in 10 apps. Figures 5.3 to 5.5 show snippets of these issues.

Finding 6. We observed significant failures in instrumentation tools consistent with underlying bugs, affecting at least 69 apps.

```
Exception in thread "Thread-43" Exception in thread "main" java.util.ConcurrentModificationExceptior
    at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
    at java.util.ArrayList$Itr.next(ArrayList.java:859)
    at soot.SootClass.getMethodsByNameAndParamCount(SootClass.java:1319)
    at soot.FastHierarchy.getSignaturePolymorphicMethod(FastHierarchy.java:982)
    at soot.FastHierarchy.resolveMethod(FastHierarchy.java:881)
    at soot.FastHierarchy.resolveMethod(FastHierarchy.java:832)
    at soot.SootMethodRefImpl.tryResolve(SootMethodRefImpl.java:229)
    at soot.SootMethodRefImpl.resolve(SootMethodRefImpl.java:275)
    at soot.SootMethodRefImpl.resolve(SootMethodRefImpl.java:207)
    at soot.jimple.internal.AbstractInvokeExpr.getMethod(AbstractInvokeExpr.java:60)
    at soot.jimple.validation.InvokeArgumentValidator.validate(InvokeArgumentValidator.java:61)
    at soot.jimple.JimpleBody.validate(JimpleBody.java:132)
    at soot.jimple.JimpleBody.validate(JimpleBody.java:114)
    at com.jordansamhi.androspecter.instrumentation.Logger.addLogStatement(Logger.java:150)
    at com.jordansamhi.androspecter.instrumentation.Logger.addLogToMethod(Logger.java:223)
    at com.jordansamhi.androspecter.instrumentation.Logger.lambda$logAllMethods$2(Logger.java:245)
    at com.jordansamhi.androspecter.instrumentation.Logger$1.internalTransform(Logger.java:104)
    at soot.BodyTransformer.transform(BodyTransformer.java:47)
    at soot.Transform.apply(Transform.java:126)
    at soot.BodyPack.internalApply(BodyPack.java:49)
    at soot.Pack.apply(Pack.java:126)
    at soot.PackManager.runBodyPacks(PackManager.java:992)
    at soot.PackManager.lambda$runBodyPacks$0(PackManager.java:667)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
    at java.lang.Thread.run(Thread.java:748)
java.util.ConcurrentModificationException
    at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
    at java.util.ArrayList$Itr.next(ArrayList.java:859)
    at soot.SootClass.getMethodsByNameAndParamCount(SootClass.java:1319)
    at soot.FastHierarchy.getSignaturePolymorphicMethod(FastHierarchy.java:982)
    at soot.FastHierarchy.resolveMethod(FastHierarchy.java:881)
    at soot.FastHierarchy.resolveMethod(FastHierarchy.java:832)
    at soot.SootMethodRefImpl.tryResolve(SootMethodRefImpl.java:229)
    at soot.SootMethodRefImpl.resolve(SootMethodRefImpl.java:275)
    at soot.SootMethodRefImpl.resolve(SootMethodRefImpl.java:207)
    at soot.jimple.internal.AbstractInvokeExpr.getMethod(AbstractInvokeExpr.java:60)
    at soot.jimple.validation.InvokeArgumentValidator.validate(InvokeArgumentValidator.java:61)
    at soot.jimple.JimpleBody.validate(JimpleBody.java:132)
    at soot.jimple.JimpleBody.validate(JimpleBody.java:114)
    at com.jordansamhi.androspecter.instrumentation.Logger.addLogStatement(Logger.java:150)
    at com.jordansamhi.androspecter.instrumentation.Logger.addLogToMethod(Logger.java:223)
    at com.jordansamhi.androspecter.instrumentation.Logger.lambda$logAllMethods$2(Logger.java:245)
    at com.jordansamhi.androspecter.instrumentation.Logger$1.internalTransform(Logger.java:104)
    at soot.BodyTransformer.transform(BodyTransformer.java:47)
    at soot.Transform.apply(Transform.java:126)
    at soot.BodyPack.internalApply(BodyPack.java:49)
    at soot.Pack.apply(Pack.java:126)
    at soot.PackManager.runBodyPacks(PackManager.java:992)
    at soot.PackManager.lambda$runBodyPacks$0(PackManager.java:667)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
```

Figure 5.3: Possible concurrency bug in the Soot engine.

5.4 Identifying Defenses via Exception-Sites

All possible evidence indicating that the app includes defenses against repackaging is analyzed through a sequence of steps: 1) first, the exception-site is identified; 2) next, we parse the log to locate the exception-site; 3) attempt to establish a relation between the exception-site and a broader component, such as a shared object library or a specific component responsible for the exception. 4) Finally, these findings are related to the results of the static analysis phase, including indicators such as string literals, function names, and the presence or usage of cryptographic algorithms.

```
  File
  "/home/leandro/dev/forensicmate-static-analysis/tools/acvtool/acvtool/smiler/instrumen
  ting/core/method_instrumenter.py", line 266, in instrument_method
    insns, cover_index= self.get_instrumented_insns_and_labels(method, reg_map, regs,
    cover_index, instrument)
                        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                        ^^^^^^^^^^^^^^^^^^^^^^
  File
  "/home/leandro/dev/forensicmate-static-analysis/tools/acvtool/acvtool/smiler/instrumen
  ting/core/method_instrumenter.py", line 120, in get_instrumented_insns_and_labels
    line = MethodInstrumenter.get_insn_change_registers(insn, reg_map)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File
  "/home/leandro/dev/forensicmate-static-analysis/tools/acvtool/acvtool/smiler/instrumen
  ting/core/method_instrumenter.py", line 193, in get_insn_change_registers
    line = MethodInstrumenter.replace_registers(line, matched, reg_map)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File
  "/home/leandro/dev/forensicmate-static-analysis/tools/acvtool/acvtool/smiler/instrumen
  ting/core/method_instrumenter.py", line 199, in replace_registers
    line = re.sub('(%s)' % reg, reg_map[reg], line, 1)
                               ~~~~~~~~^^^^^
KeyError: 'p1'
```

Figure 5.4: ACVTool's registers management.

Figure 5.6 illustrates this process using the example of the **PairipCore**[4] library, which was exposed in three apps. In this figure, the box labeled with number four (4) contains symbols that were extracted through InstruMate's static analysis and are used as an additional source of information about the component that produced evidence of the repackaging defense.

5.5 Combined Results

Table 5.4 presents a summary of the apps, categorized according to their classification groups. In this table, column NR denotes the number of apps in each group that were deemed **non-resilient** to repackaging. This classification indicates that at least one of the variant makers succeeded in producing a repackaged variant that exhibited behavior consistent with the original app in terms of both UI Elements and exception-sites.

Conversely, column RD identifies apps that demonstrated resilent or divergent behavior after repackaging. These variants exhibited differences in UI Elements or exception-sites, indicating a disruption in functionality or intentional defense.

Apps that exposed identifiable defenses against repackaging—traceable through the process outlined in Section 5.4—are summarized in Table 5.5.

5.6 Chapter Summary

The evaluation of InstruMate on a curated dataset of 156 Android apps resulted in the successful generation of 1,693 repackaged variants—72% of the expected total. While

[4] https://github.com/Solaree/pairipcore

package_name	app_version	failure_reason
br.gov.caixa.fgts.trabalhador	4.0.2	can't find app PID
br.gov.serpro.cnhe	6.10.3	can't install app
com.billiards.city.pool.nation.club	3.0.86	can't find app PID
com.daraz.android	9.6.0	can't find app PID
com.discord	253.18 - Stable	can't install app
com.ea.game.nfs14_row	8.0.0	App has PID, but window has problems
com.ea.game.nfs14_row	8.0.0	can't find app PID
com.ea.game.nfs14_row	8.0.0	can't install app
com.ea.games.simsfreeplay_row	5.88.2	App has PID, but window has problems
com.ea.games.simsfreeplay_row	5.88.2	can't find app PID
com.flipkart.android	8.11	App has PID, but window has problems
com.flipkart.android	8.11	can't find app PID
com.indeed.android.jobsearch	204.0	App has PID, but window has problems
com.indeed.android.jobsearch	204.0	can't find app PID
com.innersloth.spacemafia	2024.10.29	can't find app PID
com.ismaker.android.simsimi	8.8.3	can't find app PID
com.ixigo.train.ixitrain	7.0.6.7	can't install app
com.joeware.android.gpulumera	6.0.90-play	can't find app PID
com.lazada.android	7.63.2	can't find app PID
com.meesho.supply	20.9	can't find app PID
com.myairtelapp	4.105.4	can't find app PID
com.rubygames.assassin	1.993	can't install app
com.socialnmobile.dictapps.notepad.color.note	4.5.3	App has PID, but window has problems
com.socialnmobile.dictapps.notepad.color.note	4.5.3	can't find app PID
com.teacapps.barcodescanner	3.2.2-L	can't install app

Figure 5.5: Failures during launching ApectJ instrumented apps.

Groups G1 to G3 (signature, manifest, and resource modifications) achieved high success rates in producing healthy and compatible variants, instrumentation-based variants in Group G4 encountered significantly more failures. The Healthy Variant Gap (HVG) metric was introduced to assess variant maker performance relative to peers, revealing that merged APKs slightly mitigated decoding issues in G2 and G3, but offered limited benefit in G4. The experiment identified specific anti-repackaging defenses embedded in apps, misleading UI elements, as well as spotted root causes of failure. Notably, no single instrumentation technique proved universally effective.

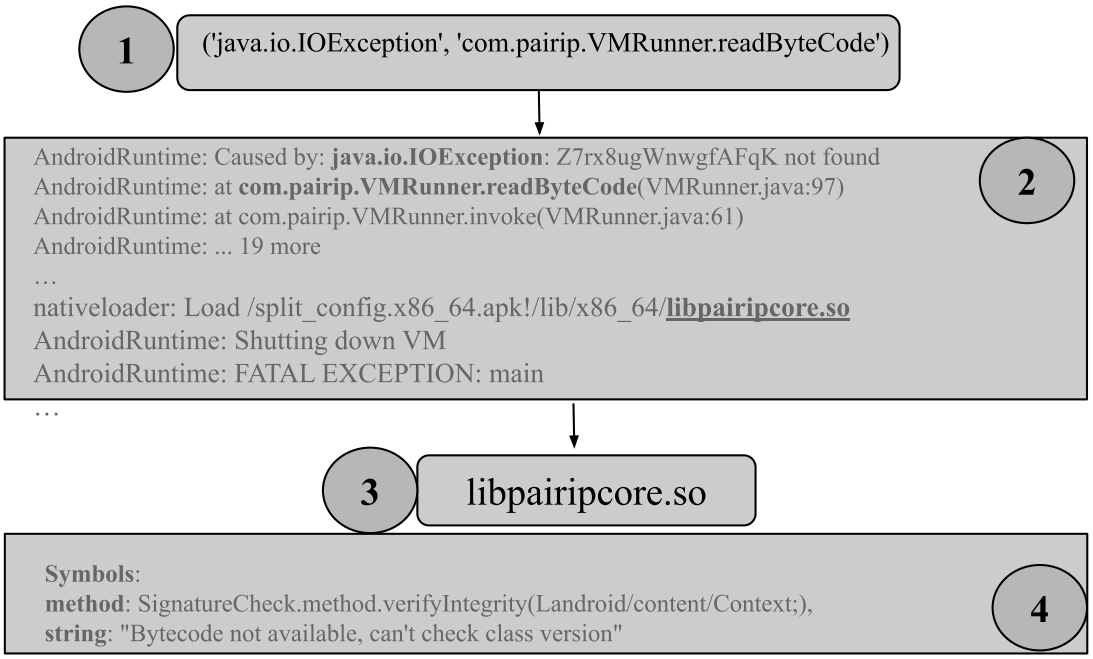


Figure 5.6: Exception-site analysis

Table 5.4: Classification of the original apps per group. Columns: NR (Non-Resilient), RD (Resilient/Divergent). In this table, the percentage values are calculated with respect to the number of apps indicated in the first column (#Apps).

#Apps		NR		RD	
G1	156 100%	134	86%	22	14%
G2	134 100%	127	95%	7	5%
G3	134 100%	130	97%	4	3%
G4	134 100%	102	76%	32	24%

Table 5.5: Apps that exposed defenses. The columns UI, ES, and K indicate defense exposure through UI elements, exception-sites, and self-termination, respectively.

Package ID	G	UI	ES	K
com.hp.android.printservice	G1			✓
com.intsig.camscanner	G1	✓		
com.microsoft.office.excel	G1		✓	
com.microsoft.office.officehubrow	G1	✓	✓	
com.microsoft.office.onenote	G1	✓	✓	
com.microsoft.office.powerpoint	G1		✓	
com.microsoft.office.word	G1		✓	
com.miniclip.carrom	G1		✓	
com.roblox.client	G1		✓	
com.vectorunit.purple.googleplay	G1		✓	
com.billiards.city.pool.nation.club	G4	✓		
com.jio.myjio	G4	✓		
com.firsttouchgames.story	G4	✓		
com.cyberlink.youcammakeup	G4	✓		
com.myntra.android	G4		✓	
com.utorrent.client	G4		✓	
com.vkontakte.android	G4		✓	

Chapter 6

Final Remarks

In this chapter, we answer our research questions, present some implications of our research, and highlight some threats to the validity of our study.

6.1 Answers to the Research Questions

Out of the 156 initial apps, 134 (86%) were found to be non-resilient to the simplest form of repackaging—merely changing the app's signature. Our analysis also revealed that 127 apps (81%) permitted execution in debug mode, enabling reverse engineering techniques that attackers could potentially exploit. Additionally, 130 apps (83%) offered no resistance to superficial modifications, such as changes to the app name. Attackers can leverage these basic modifications to undermine developer revenue or redirect user traffic for data exfiltration. These findings provide critical insights toward answering our first research question.

> **Answer to RQ1:** The variant makers successfully repackaged 86% of the apps in our curated dataset, revealing that popular Android apps remain highly susceptible to repackaging.

Furthermore, using black-box static and dynamic instrumentation techniques, it was confirmed that 102 apps (65%) were susceptible to modifications in their code sections. Although no extensive stress testing was performed to determine if other parts of the selected apps were protected, the presence of healthy variants that remain compatible with their original counterparts at initial launch presents a significant security risk, as they could serve as a vehicle for malware distribution. From those apps that InstruMate failed to repackage, we found a few recurrent patterns of defense mechanisms that prevent

the execution of repackaged apps, produce user interface alerts, trigger privilege-escalation attempts, or induce restart loops.

> **Answer to RQ2:** A small number of apps use libraries such as `libplat.so` and `libpairipcore.so` to detect repackaging attempts and prevent app variants from running, even when only simple modifications are applied. InstruMate proved effective in collecting evidence of UI modifications and new exception sites through its health check procedures. Some of these modifications can be attributed to the repackage proofing mechanisms implemented by a subset of the apps.

Besides defense mechanisms, the variant makers also failed to instrument several apps. For example, only 14 apps were successfully repackaged using aspect-oriented instrumentation, whereas Frida and Androlog (without merging) were able to instrument 51 and 45 apps, respectively. We identified several factors that prevented variant makers from producing working variants, including the internal structure of APKs, potential concurrency bugs, and the misinterpretation of DEX opcodes.

> **Answer to RQ3:** Variant makers in group **G4** struggle to build working variants due to the internal structure of the APKs, potential bugs in the tools, and misinterpretation of DEX opcodes.

6.2 Implications

No single instrumentation approach proved universally effective, as different strategies succeeded in instrumenting apps where others failed. Consequently, employing multiple strategies through **InstruMate** offers a practical means of maximizing the likelihood of successfully instrumenting Android apps—a valuable approach for researchers seeking in-depth analyses, such as those aimed at identifying cryptographic API misuses or other blackbox reasearch on Android apps. In addition, software engineers who rely on repackaging should be mindful of the possibility of unintended behavioral modifications in the monitored app, which could be classified as healthy, but incompatible with its original counterpart.

Although this research does not compare coverage between the original and repackaged versions, variants that report unseen exception sites or unseen UI elements exercise different code sections when compared to their original, indicating that the program followed a different execution path.

6.3 Threats to Validity

Our empirical assessment relies on several assumptions that could potentially threaten the validity of our results. Regarding external validity, one might argue that our curated dataset of 156 apps is insufficient to support broad generalizations. However, our initial dataset consisted of the 500 most popular Android apps, and we applied a rigorous filtering procedure. Specifically, we excluded pre-installed vendor apps and those that did not crash during a set of 200 one-minute health check procedures. This exclusion process, together with the extensive set of observations that formed the baseline, as detailed in Section 4.3, provides strong evidence that any deviations in user interface elements or exceptions could be attributed to the repackaging process. Importantly, we did not apply any arbitrary selection criteria that might have favored certain apps over others. Therefore, we believe that increasing the number of apps in our dataset would not significantly alter our main findings—such as the success rates achieved by the variant makers.

It is noteworthy that recent research on Google Play often relies on AndroZoo as the primary source of samples. For example, studies such as [90] and [91] analyzed millions of Google Play apps archived by AndroZoo. The AndroZoo project is a repository of single-APK packages, collected via a reverse-engineered Protobuf client associated with a specific (undisclosed) device configuration. However, single-APK packages no longer reflect the current state of popular apps. To ensure that our analysis focused on real-world apps, we chose to obtain them directly from Google Play which imposes an additional challenge related to app download restrictions: per account and per IP address, as also noted by AndroZoo [10].

We conducted an investigation to source the identical 156 apps utilized in our study from AndroZoo. Through package name and version number searches, 107 were successfully identified. However, only 31 of these exhibited partial matches with the base APK hash. Our analysis indicates that AndroZoo maintains solely the primary APK, evidenced by the absence of splits for these 31 apps and the presence of divergent hashes in 76 additional cases compared to those acquired via direct Google Play downloads. Moreover, among the 107 retrieved packages, merely 69 demonstrated successful launch capabilities, suggesting that AndroZoo's collection methodology yields device-specific implementations tied to its particular hardware configuration. These empirical findings substantiate our methodological decision to procure apps directly from the Google Play Store, pending AndroZoo's adaptation of its collection framework to accommodate dynamic delivery mechanisms inherent in the AAB format.

Although our variant construction and health check pipelines are largely automated, the number of generated variants increases linearly with the number of original apps. This growth poses challenges to the experiment's scalability.

Another decision that may threaten the validity of our work is the choice of the InstruMate variant makers already integrated into the framework. For the variant maker groups G1–G3, we use two widely adopted tools in Android research involving repackaging: `ApkEditor` and scripts for unpacking and repacking APKs using standard Zip utilities. For group G4, we explored a variety of tools for static and dynamic code instrumentation, ranging from general-purpose solutions such as AspectJ to more specialized tools such as Androlog [18], ACVTool [22], and the Frida dynamic instrumentation toolkit. Androlog is a recently published instrumentation and code coverage analysis tool, while Frida is a powerful toolkit commonly used by security experts [81]. Rather than representing a limitation, we argue that these pre-integrated variant makers explore distinct abstraction levels commonly discussed in the instrumentation literature: dynamic instrumentation, the Jimple [84] intermediate representation, the Java bytecode level, and direct manipulation of SMALI code.

Some might argue that our health check procedures are too weak. While we could have employed UI test generation tools—such as Monkey or DroidBot [49]—to explore apps more deeply, preliminary studies we conducted revealed that such approaches often introduce unnecessary noise, potentially compromising the effectiveness of the InstruMate health checker procedures. Moreover, as discussed in Chapter 6, if a repackaged app launches successfully, it already poses a significant security risk, as the variant could be used to distribute malware or exfiltrate sensitive data. We, therefore, conjecture that exploring deeper application states is not necessary to demonstrate that a repackaged Android app poses a risk to end users. Additionally, we argue that our health check procedures are more robust than those used in previous research that classified variants as healthy if they successfully launched and did not crash within a three-second window [22].

In this study, we use the term *"healthy but incompatible"* to describe a repackaged variant that remains functional but exhibits user interface elements or exceptions not present in the original counterpart. Although such cases were deemed unsuitable within the scope of our repackaging attempt, leading us to classify the corresponding app as resilient to repackaging, it is important to acknowledge that these variants may hold value in other contexts. For example, studies focusing on specific functionalities of an app could potentially make use of such variants. This reflects the fact that our terminology is somewhat specific to the goals of this research. Nevertheless, we argue that our definitions remain aligned with terminology already established in the instrumentation literature [18, 22, 20], with the deliberate introduction of the *"incompatible"* qualifier. This qualifier was adopted with the explicit purpose of drawing attention to differences between variants and their original counterparts, thereby raising awareness for future research to properly address these discrepancies.

Finally, our exception site identification mechanism relies on exceptions that are explicitly thrown and recorded. Consequently, apps with security mechanisms that silently log repackaging attempts—without raising exceptions or modifying the user interface—would go undetected in our experiment. While this limitation affects the precision of resiliency classification, such silent variants still pose a significant security threat.

6.4 Reproducibility and Code Availability

Reproducibility is a fundamental principle of scientific research. To support the validation and extension of our findings, we make InstruMate code and associated dataset available upon request to members of the scientific community conducting research in this domain. While the code is not openly published at this time, access can be granted for non-commercial, academic purposes, particularly to researchers aiming to replicate or build upon the work presented in this study.

Comprehensive descriptions of the selected apps, including version information and integrity hashes, are thoroughly documented and included within the codebase.

Researchers interested in accessing the implementation are encouraged to contact the authors directly. We are committed to fostering transparency and collaboration within the academic community and will make every effort to support reproducibility in line with ethical and responsible research practices.

6.5 Future Work

This Section outlines potential directions that could extend and enhance current work.

- Stress testing: Future work could explore instrumentation approaches in combination with stress testing engines such as Droidbot [49] or Google Monkey.
- Machine Learning-Enhanced Instrumentation: use machine learning to predict optimal instrumentation techniques based on app characteristics. This approach could learn from failed instrumentation attempts to identify patterns and automatically adjust strategies.
- Large-Scale Dataset Analysis: Conducting experiments on larger datasets (1000+ apps) would strengthen the generalizability of findings.
- Defense Mechanism Classification: Developing machine learning models to automatically classify and categorize different types of repackage-proofing mechanisms based on behavioral patterns, exception sites, and UI responses.

- User Guide: Develop a guide to help end users identify when they have been lured into installing repackaged apps. This would include creating instructions on recognizing warning signs such as unofficial app sources, suspicious permission requests, unexpected behavior changes, and security alerts. The guide should also provide step-by-step instructions for verifying app authenticity through official channels and recovering from potential incidents.
- Developer Guide: Develop a guide to help developers assess their apps' susceptibility to repackaging. This would involve developing a simplified version of InstruMate's assessment flow that developers can integrate into their development and testing workflows. The guide should include automated scanning procedures, best practice recommendations for implementing repackage-proofing mechanisms, and guidance on monitoring for unauthorized versions of their apps in the wild.
- Hybrid Instrumentation: Combining multiple instrumentation strategies such as Androlog and Frida to develop a fully-fledged sandbox capable of tracing both DEX and native code execution. This integrated approach would leverage the strengths of static instrumentation (Androlog) for DEX coverage and dynamic instrumentation (Frida) for runtime native code analysis.

6.6 Conclusion

This work presented and evaluated InstruMate, a systematic approach for repackaging Android apps by integrating multiple *variant makers* that apply transformations ranging from APK signature changes to static and dynamic code instrumentation. We evaluated InstruMate on a curated dataset of 156 Android apps, generating 1,693 repackaged variants, which were evaluated through a novel *health check procedure*. The results show that 86% of the apps were susceptible to repackaging, from basic modifications to advanced code tampering. Our empirical assessment also demonstrates that InstruMate and its health check procedures effectively detect common repackaging defenses by capturing traces detectable on any Android device. The only requirements are Android Debug Bridge (ADB) and UI Automator. For security-focused apps, it may be critical to conceal the specific components responsible for repackaging detection. We believe that InstruMate can foster further research in the area of repackaging, and that our findings can assist development teams in incorporating effective anti-repackaging defenses early in the design and development of Android applications.

Appendix A

Additional Figures

This appendix contains the following additional figures:

- Figure A.1 presents the set of APKs related to the TikTok app, as installed in an emulator device.
- Figure A.2 presents the items that are present in the root of the base APK.
- Figure A.3 presents the content of the arm64/v8a split APK, which contains 155 non-DEX shared objects.
- Figure A.4 presents the Cloner Premium app as shown in the Google Play Store.
- Figure A.5 presents the Mod Heaven app as shown in the Google Play Store.
- Figure A.6 presents the set of modifications that can be performed with the Cloner Premium app.
- Figure A.7 presents the ApkTool GUI, a tool that interacts with the ApkTool command line interface.
- Figure A.8 presents part of the interface of a website that enables the download of decrypted iOS apps.
- Figure A.9 presents two unofficial WhatsApp variants.
- Figure A.10 illustrates a report containing string resources that were referenced within binary files. This example demonstrates how strings such as *"Go Pound Bits"*, discussed in Section 5.2, are extracted and reported by the tool.
- Figure A.11 shows a report related to the app's user interface. It also highlights how specific content described in Section 5.2 is identified and documented by InstruMate. Notably, the message *"This CamScanner copy is not genuine"*, which is associated with repackaging defenses, is extracted along with its internal attributes.

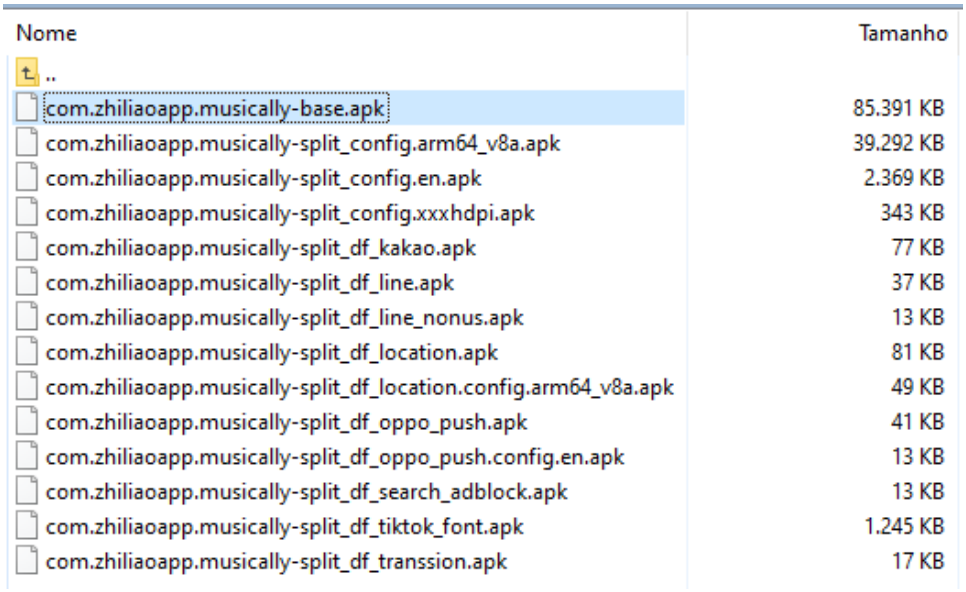
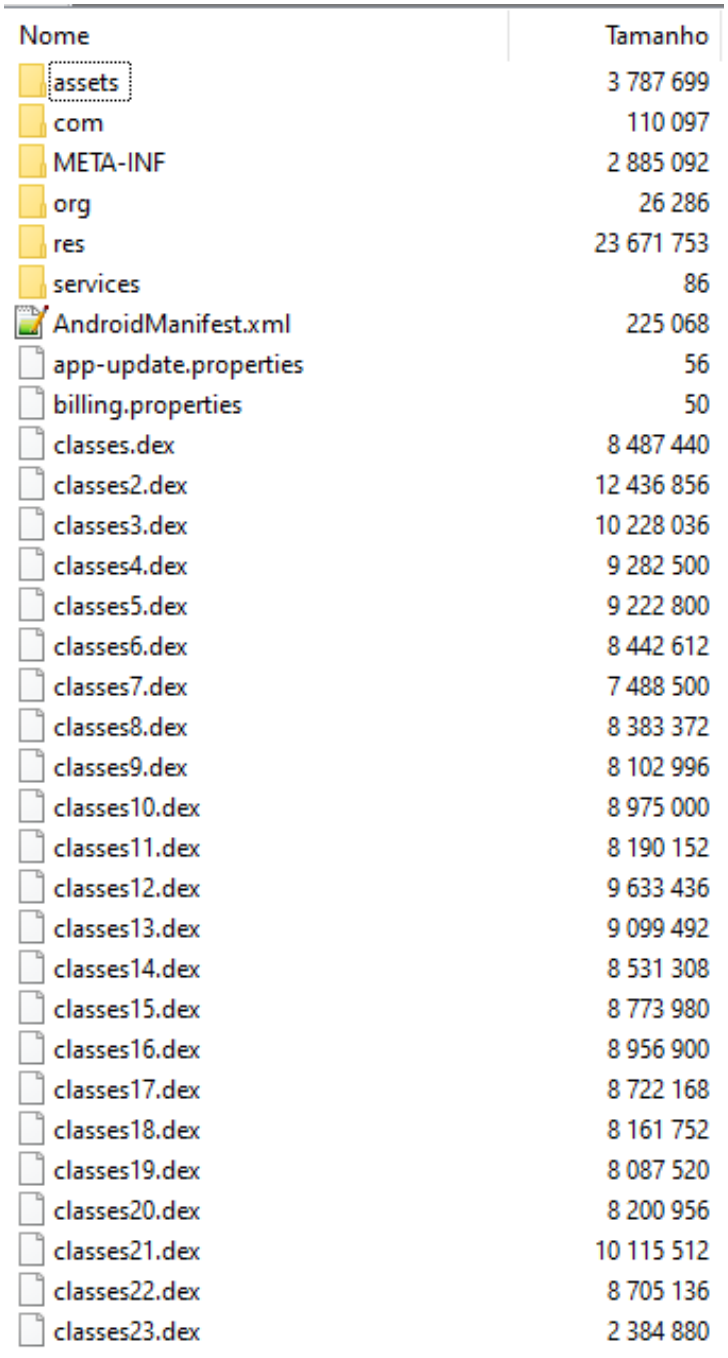
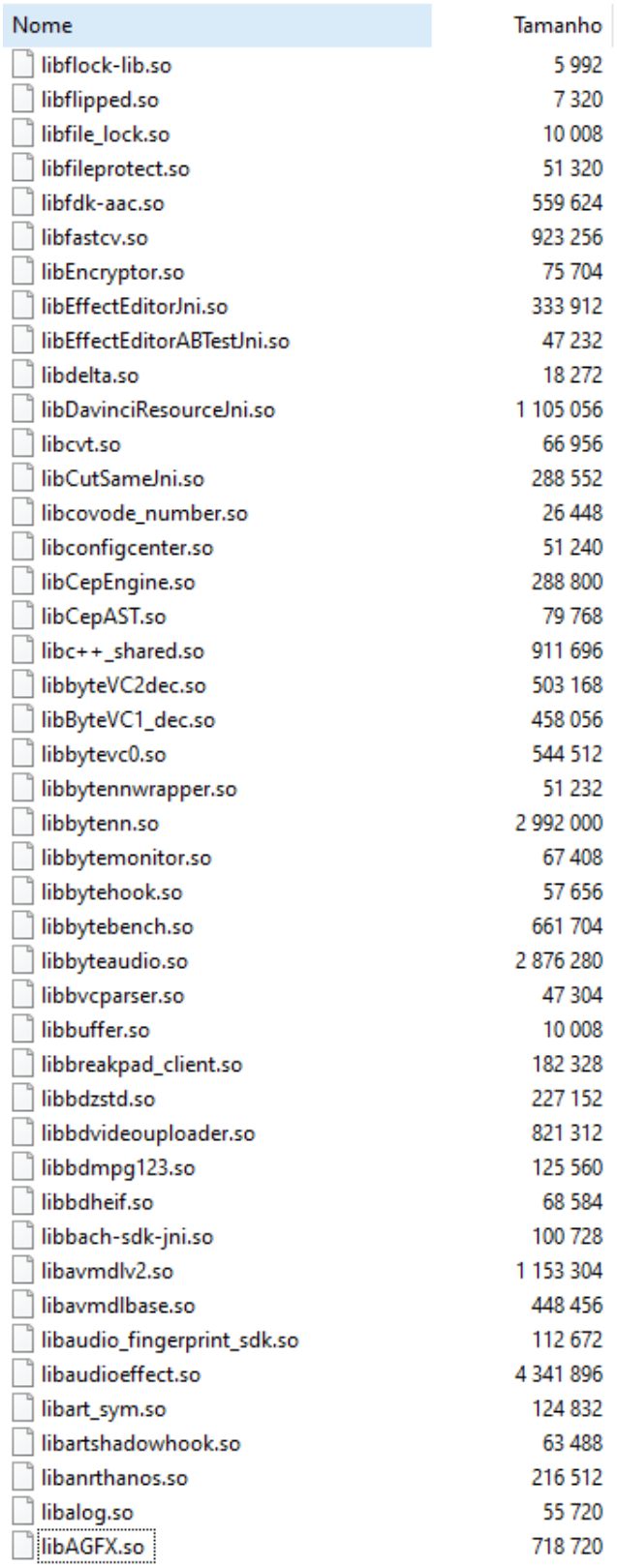


Figure A.1: Tiktok APKs that are delivered to an emulator device for proper installation.



Nome	Tamanho
assets	3 787 699
com	110 097
META-INF	2 885 092
org	26 286
res	23 671 753
services	86
AndroidManifest.xml	225 068
app-update.properties	56
billing.properties	50
classes.dex	8 487 440
classes2.dex	12 436 856
classes3.dex	10 228 036
classes4.dex	9 282 500
classes5.dex	9 222 800
classes6.dex	8 442 612
classes7.dex	7 488 500
classes8.dex	8 383 372
classes9.dex	8 102 996
classes10.dex	8 975 000
classes11.dex	8 190 152
classes12.dex	9 633 436
classes13.dex	9 099 492
classes14.dex	8 531 308
classes15.dex	8 773 980
classes16.dex	8 956 900
classes17.dex	8 722 168
classes18.dex	8 161 752
classes19.dex	8 087 520
classes20.dex	8 200 956
classes21.dex	10 115 512
classes22.dex	8 705 136
classes23.dex	2 384 880

Figure A.2: The contents of the Tiktok's base APK.



Nome	Tamanho
libflock-lib.so	5 992
libflipped.so	7 320
libfile_lock.so	10 008
libfileprotect.so	51 320
libfdk-aac.so	559 624
libfastcv.so	923 256
libEncryptor.so	75 704
libEffectEditorJni.so	333 912
libEffectEditorABTestJni.so	47 232
libdelta.so	18 272
libDavinciResourceJni.so	1 105 056
libcvt.so	66 956
libCutSameJni.so	288 552
libcovode_number.so	26 448
libconfigcenter.so	51 240
libCepEngine.so	288 800
libCepAST.so	79 768
libc++_shared.so	911 696
libbyteVC2dec.so	503 168
libByteVC1_dec.so	458 056
libbytevc0.so	544 512
libbytennwrapper.so	51 232
libbytenn.so	2 992 000
libbytemonitor.so	67 408
libbytehook.so	57 656
libbytebench.so	661 704
libbyteaudio.so	2 876 280
libbvcparser.so	47 304
libbuffer.so	10 008
libbreakpad_client.so	182 328
libbdzstd.so	227 152
libbdvideouploader.so	821 312
libbdmpg123.so	125 560
libbdheif.so	68 584
libbach-sdk-jni.so	100 728
libavmdlv2.so	1 153 304
libavmdlbase.so	448 456
libaudio_fingerprint_sdk.so	112 672
libaudioeffect.so	4 341 896
libart_sym.so	124 832
libartshadowhook.so	63 488
libanrthanos.so	216 512
libalog.so	55 720
libAGFX.so	718 720

Figure A.3: The contents of the *arm64/v8a* split APK, which contains 155 non-DEX shared objects (`.so` extension)—only a partial listing is shown.

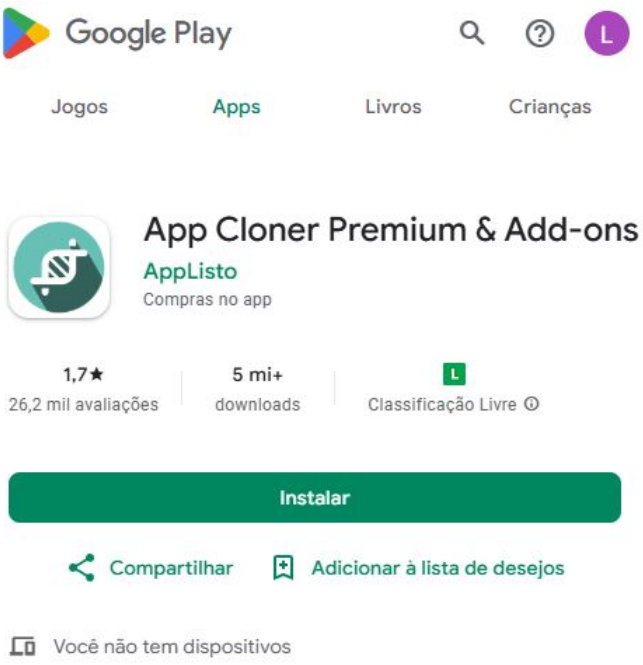


Figure A.4: The App Cloner Premium, available at the Google Play Store. URL: https://play.google.com/store/apps/details?id=com.applisto.appcloner.premium.

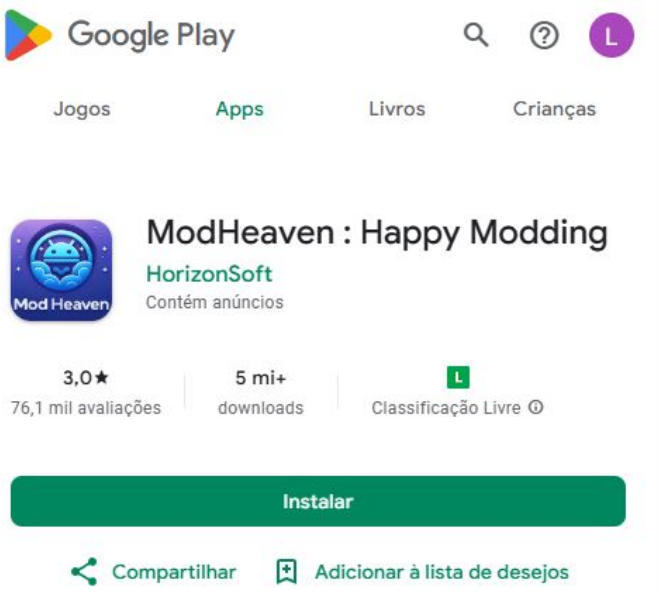


Figure A.5: The ModHeaven App, available at the Google Play Store. URL: https://play.google.com/store/apps/details?id=com.modheaven.

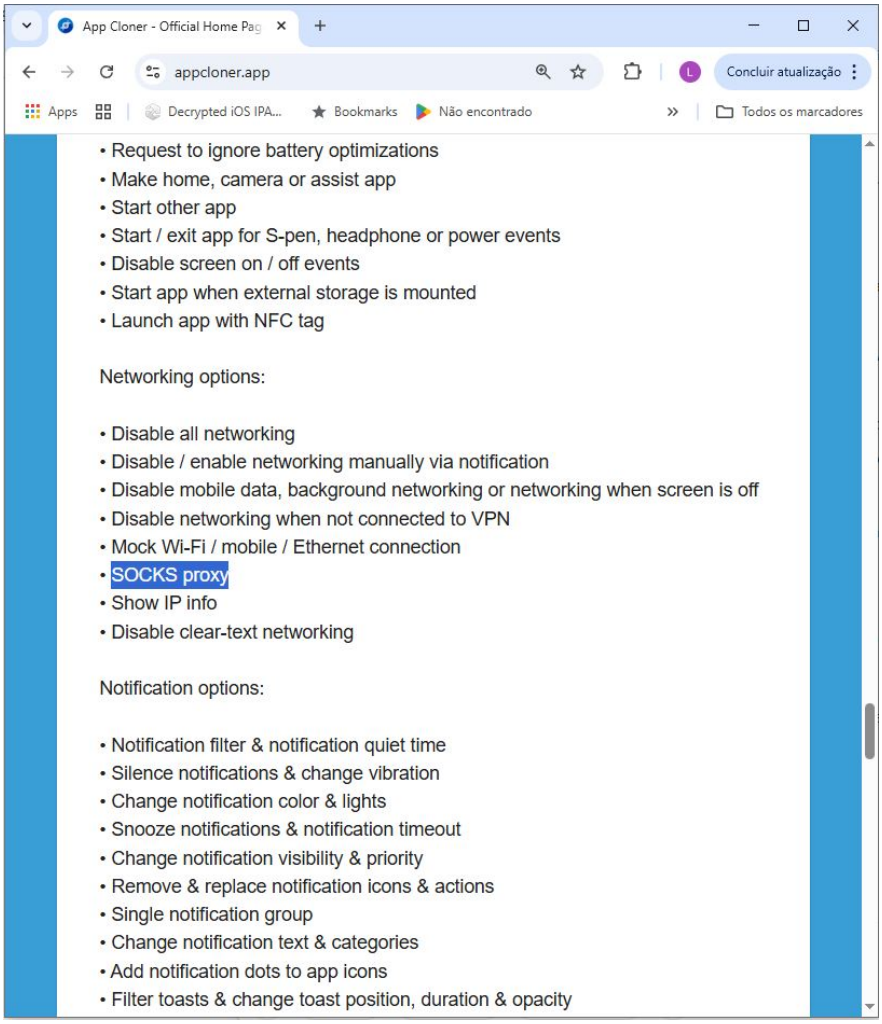


Figure A.6: Capabilities made available by the App Cloner Premium. URL: https://appcloner.app/.

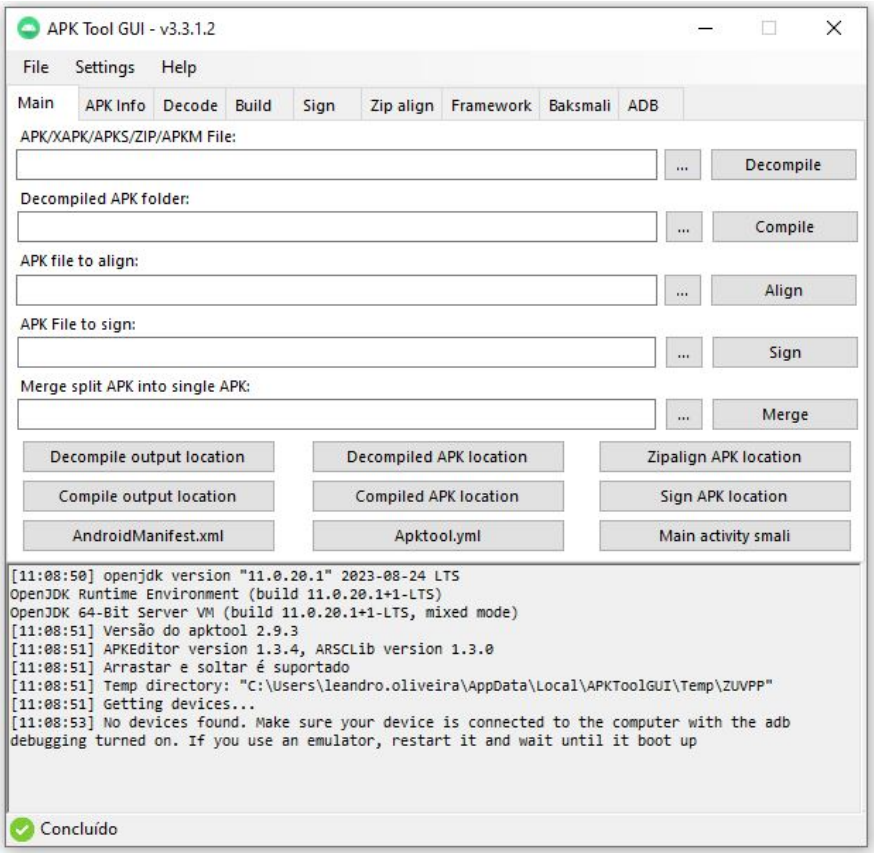


Figure A.7: Popular free tool for creating repackaged variants. URL: https://github.com/AndnixSH/APKToolGUI.

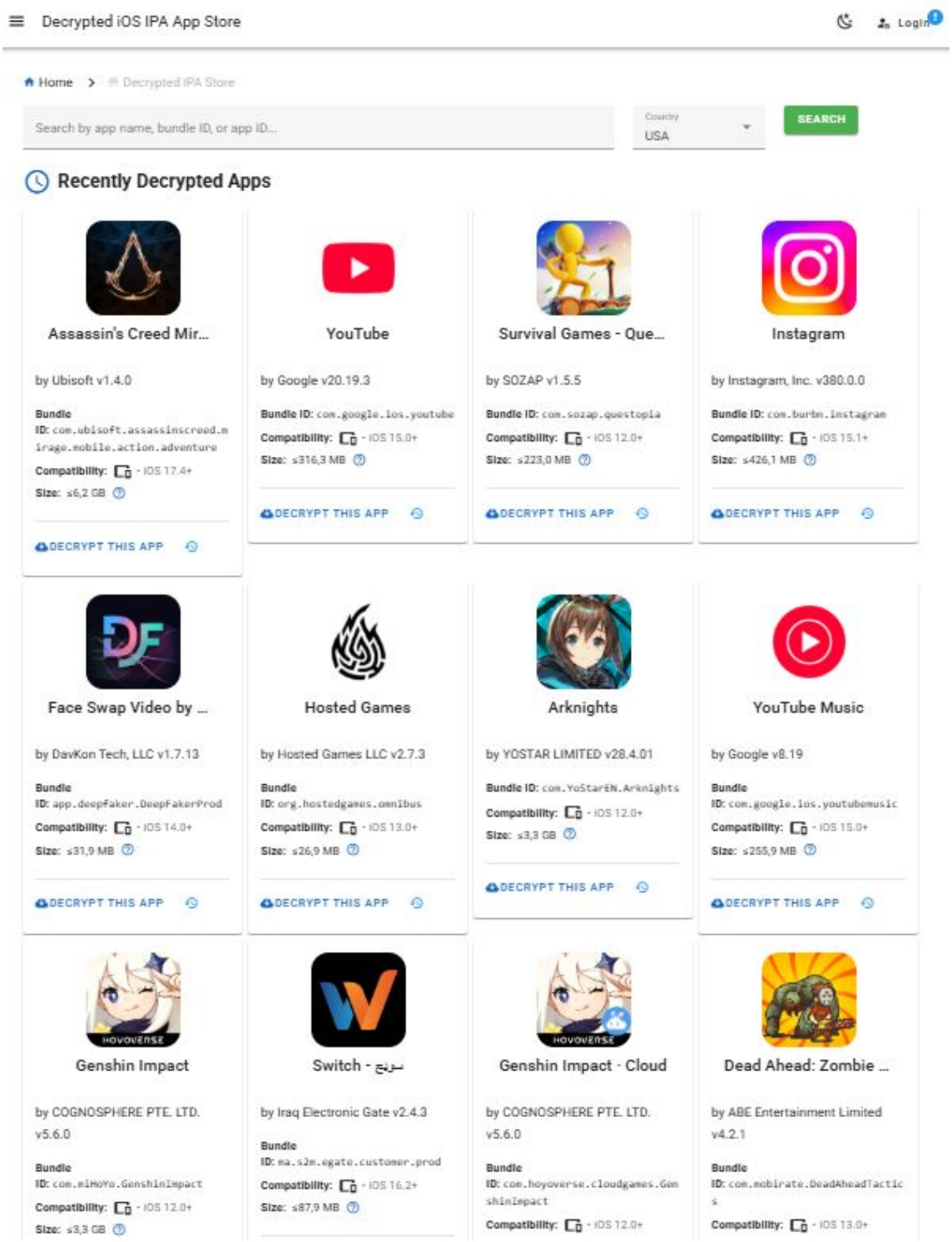


Figure A.8: Shows a listing of iOS apps available to be downloaded. The apps are advertised as being already decrypted IPA (iOS App Package format).

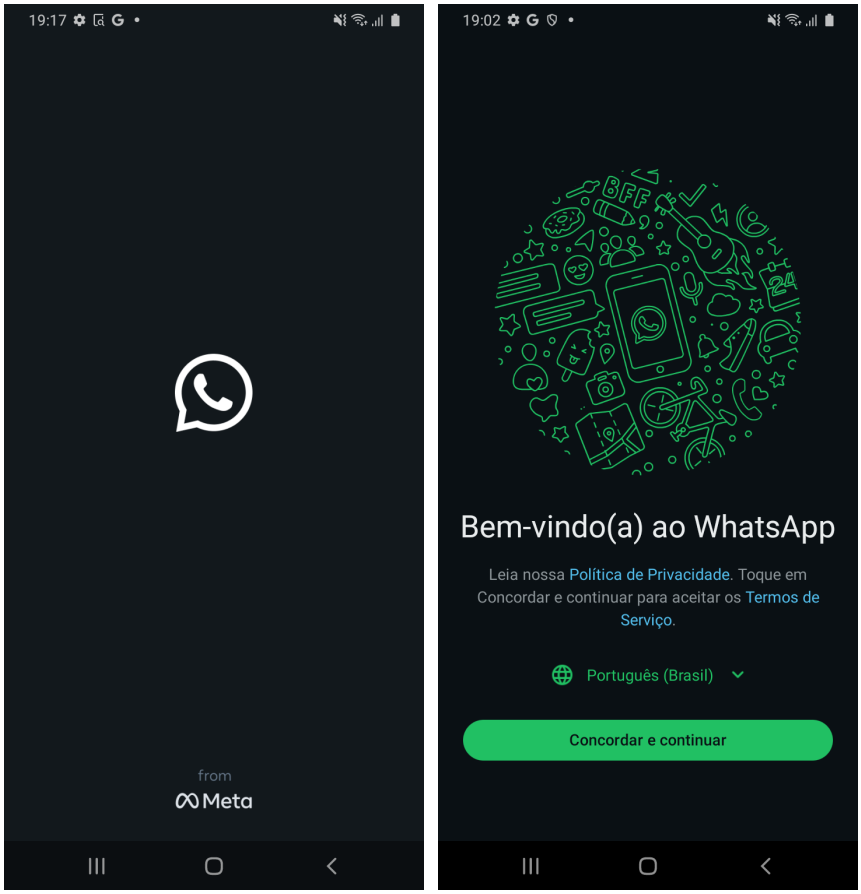


Figure A.9: In the initial stage, these variants display a logo and user interface elements identical to those of the official app. Additionally, they offer extra functionalities.

```
}, {
    "vaddr": 2213,
    "paddr": 2213,
    "ordinal": 9,
    "size": 107,
    "length": 106,
    "section": ".rodata",
    "type": "ascii",
    "string": "Stop! Who approacheth the Bridge of Death must answer me these questions t
}, {
    "vaddr": 2342,
    "paddr": 2342,
    "ordinal": 11,
    "size": 15,
    "length": 14,
    "section": ".rodata",
    "type": "ascii",
    "string": "Go pound bits!"
}, {
    "vaddr": 2357,
    "paddr": 2357,
    "ordinal": 12,
    "size": 45,
    "length": 44,
    "section": ".rodata",
    "type": "ascii",
    "string": "And you keep wondering why nobody likes you?"
}, {
```


Figure A.10: Presents a portion of the static analysis result. This output is also mentioned at Section 5.2.

```
"80c83d6701bec6dfef513d13fe04cad8": {
    "android_class": "android.widget.TextView",
    "package": "com.intsig.camscanner",
    "uniqueId": "id/no_id/9",
    "parentUniqueId": "id/no_id/8",
    "resourceID": "com.intsig.camscanner:id/message_panel",
    "contentDesc": "",
    "text": "This CamScanner copy is not genuine. Please do not purchase any service.",
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,
    "scrollable": false,
    "visibility": -1,
    "password": false,
    "selected": false,
    "signature": "80c83d6701bec6dfef513d13fe04cad8"
},
"a681724f7eaa8f1357cf0ffbc49c1d12": {
    "android_class": "android.widget.LinearLayout",
    "package": "com.intsig.camscanner",
    "uniqueId": "id/no_id/10",
    "parentUniqueId": "id/no_id/4",
    "resourceID": "com.intsig.camscanner:id/bottom_panel",
    "contentDesc": "",
    "text": "",
    "checkable": false,
    "checked": false,
    "clickable": false,
    "enabled": true,
    "focusable": false,
    "focused": false,
    "scrollable": false,
    "visibility": -1,
    "password": false,
    "selected": false,
    "signature": "a681724f7eaa8f1357cf0ffbc49c1d12"
},
```


Figure A.11: Presents a portion of the heatlh check procedures result, containing a description of the UI-Elements. This output is also mentioned at Section 5.2.

Appendix B

Dataset

Table B.1: Detailed listing of the apps in the dataset.

Package ID	Name	Category	Installs	Rating	Splits	Size (MB)	DEX	Native
com.facebook.katana	Facebook	SOCIAL	5000000000	4.25	1	81.74	1	14
com.facebook.orca	Messenger	COMMUNICATION	5000000000	4.07	1	75.74	10	14
com.instagram.android	Instagram	SOCIAL	5000000000	4.01	1	88.76	12	13
com.whatsapp	WhatsApp Messenger	COMMUNICATION	5000000000	4.3	2	85.59	6	2
com.facebook.lite	Facebook Lite	SOCIAL	1000000000	3.73	1	2.53	1	4
com.fingersoft.hillclimb	Hill Climb Racing	GAME_RACING	1000000000	4.42	1	104.03	6	4
com.imo.android.imoim	imo-International Calls	COMMUNICATION	1000000000	4.22	10	85.93	14	53
com.lemon.lvoverseas	CapCut - Video Editor	VIDEO_PLAYERS	1000000000	4.32	4	150.72	24	139
com.linkedin.android	LinkedIn	BUSINESS	1000000000	4.11	4	112.95	8	10
com.microsoft.appmanager	Link to Windows	PRODUCTIVITY	1000000000	3.97	1	104.21	11	22
com.microsoft.office.excel	Microsoft Excel: Spreadsheets	PRODUCTIVITY	1000000000	4.78	1	231.67	4	76
com.microsoft.office.officehubrow	Microsoft 365 (Office)	PRODUCTIVITY	1000000000	4.71	5	370.76	8	101
com.microsoft.office.powerpoint	Microsoft PowerPoint	PRODUCTIVITY	1000000000	4.75	1	227.47	4	79
com.microsoft.office.word	Microsoft Word: Edit Documents	PRODUCTIVITY	1000000000	4.78	1	249.08	4	84
com.outfit7.mytalkingtomfree	My Talking Tom	GAME_CASUAL	1000000000	4.3	2	162.21	7	20
com.picsart.studio	Picsart: AI Photo Video Editor	PHOTOGRAPHY	1000000000	4.05	4	97.7	8	15
com.pinterest	Pinterest	LIFESTYLE	1000000000	4.51	4	69.12	6	11
com.roblox.client	Roblox	GAME_ADVENTURE	1000000000	4.44	2	163.81	2	9
com.skype.raider	Skype	COMMUNICATION	1000000000	4.27	1	75.2	5	82
com.snapchat.android	Snapchat	COMMUNICATION	1000000000	4.09	4	157.77	9	13
com.touchtype.swiftkey	Microsoft SwiftKey AI Keyboard	PERSONALIZATION	1000000000	4.5	7	37.04	6	5
com.twitter.android	X	SOCIAL	1000000000	3.69	4	105.55	8	13
com.whatsapp.w4b	WhatsApp Business	COMMUNICATION	1000000000	4.4	1	65.8	7	2
com.zhiliaoapp.musically	TikTok	SOCIAL	1000000000	4.19	14	125.95	9	155
org.telegram.messenger	Telegram	COMMUNICATION	1000000000	4.3	4	69.26	4	2
us.zoom.videomeetings	Zoom Workplace	BUSINESS	1000000000	4.11	3	138.5	33	85
air.com.hypah.io.slither	slither.io	GAME_ACTION	500000000	3.94	1	51.29	2	3
cn.wps.moffice_eng	WPS Office-PDF,Word,Sheet,PPT	PRODUCTIVITY	500000000	4.59	4	145.17	16	37
cn.xender	Xender - Share Music Transfer	TOOLS	500000000	4.4	1	31.21	6	4
com.amazon.mShop.android.shopping	Amazon Shopping	SHOPPING	500000000	4.4	3	126.6	8	47
com.booking	Booking.com: Hotels	TRAVEL_AND_LOCAL	500000000	4.51	3	161.84	12	5
com.ea.game.pvzfree_row	Plants vs. Zombies™	GAME_STRATEGY	500000000	4.17	2	127.01	6	10
com.fgol.HungrySharkEvolution	Hungry Shark Evolution	GAME_ARCADE	500000000	4.46	2	199.85	4	5

Continued on next page

Table B.1 – *continued from previous page*

Package ID	Name	Category	Installs	Rating	Splits	Size (MB)	DEX	Native
com.flipkart.android	Flipkart Online Shopping App	SHOPPING	500000000	4.3	3	32.23	4	35
com.gameloft.android.ANMP.GloftDMHM	Minion Rush: Running Game	GAME_CASUAL	500000000	4.38	4	163.52	7	16
com.heytap.browser	Internet Browser	TOOLS	500000000	3.71	2	155.01	15	11
com.hp.android.printservice	HP Print Service Plugin	PRODUCTIVITY	500000000	4.15	1	60.86	3	68
com.innersloth.spacemafia	Among Us	GAME_ACTION	500000000	3.85	3	720.47	2	8
com.jio.myjio	MyJio: For Everything Jio	PRODUCTIVITY	500000000	4.3	3	215.33	15	65
com.king.candycrushsodasaga	Candy Crush Soda Saga	GAME_CASUAL	500000000	4.61	4	132.33	4	3
com.kwai.video	Kwai - download	VIDEO_PLAYERS	500000000	4.48	6	6.09	5	2
com.meesho.supply	Meesho: Online Shopping App	SHOPPING	500000000	4.15	3	27.6	3	1
com.microsoft.office.onenote	Microsoft OneNote: Save Notes	PRODUCTIVITY	500000000	4.66	3	167.47	4	34
com.nianticlabs.pokemongo	Pokémon GO	GAME_ADVENTURE	500000000	3.97	2	203.51	3	24
com.opera.mini.native	Opera Mini: Fast Web Browser	COMMUNICATION	500000000	4.54	7	42.83	5	17
com.outfit7.mytalkingangelafree	My Talking Angela	GAME_CASUAL	500000000	3.93	2	143.32	7	20
com.outfit7.mytalkingtom2	My Talking Tom 2	GAME_CASUAL	500000000	4.24	2	160.54	9	12
com.outfit7.mytalkingtomfriends	My Talking Tom Friends	GAME_CASUAL	500000000	4.24	2	154.74	7	13
com.outfit7.talkingtom	Talking Tom Cat	GAME_CASUAL	500000000	4.01	3	91.88	7	13
com.outfit7.talkingtomgoldrun	Talking Tom Gold Run	GAME_ACTION	500000000	4.27	2	162.44	7	19
com.phonepe.app	PhonePe UPI, Payment, Recharge	FINANCE	500000000	4.37	3	150.53	10	26
com.playit.videoplayer	PLAYit-All in One Video Player	VIDEO_PLAYERS	500000000	4.46	4	44.74	5	20
com.quvideo.xiaoying	VivaVideo - Video Editor	VIDEO_PLAYERS	500000000	4.41	4	128.77	15	74
com.rubygames.assassin	Hunter Assassin	GAME_ACTION	500000000	4.27	4	124.1	10	15
com.transsion.filemanagerx	File Manager	TOOLS	500000000	4.33	1	8.27	2	0
com.transsion.magicshow	Visha-Video Player All Formats	VIDEO_PLAYERS	500000000	4.45	4	67.1	6	29
com.waze	Waze Navigation	MAPS_AND_NAVIGATION	500000000	4.35	2	106.22	7	8
com.zaz.translate	Hi Translate - Chat translator	TOOLS	500000000	4.42	4	35.5	4	6
com.zzkko	SHEIN-Shopping Online	SHOPPING	500000000	4.28	3	126.9	8	31
flipboard.app	Flipboard: The Social Magazine	NEWS_AND_MAGAZINES	500000000	3.59	4	39.28	5	3
jp.naver.line.android	LINE: Calls	COMMUNICATION	500000000	3.86	3	245.24	10	84
me.pou.app	Pou	GAME_CASUAL	500000000	4.33	1	27.79	2	0
sg.bigo.live	Bigo Live - Live Streaming App	SOCIAL	500000000	4.47	4	85.36	18	54
air.com.lunime.gachalife	Gacha Life	GAME_CASUAL	100000000	4.43	1	99.56	2	3
app.source.getcontact	Getcontact	COMMUNICATION	100000000	3.9	3	90.65	14	20
br.com.gabba.Caixa	CAIXA	FINANCE	100000000	4.58	4	70.86	6	11
br.gov.caixa.fgts.trabalhador	FGTS	FINANCE	100000000	3.45	1	30.75	2	19

Continued on next page

Table B.1 – *continued from previous page*

Package ID	Name	Category	Installs	Rating	Splits	Size (MB)	DEX	Native
br.gov.caixa.tem	CAIXA Tem	FINANCE	100000000	3.81	1	19.23	3	14
co.brainly	Brainly: AI Homework Helper	EDUCATION	100000000	4.46	4	81.41	9	9
com.adobe.scan.android	Adobe Scan: PDF Scanner, OCR	BUSINESS	100000000	4.75	4	102.61	2	26
com.amazon.kindle	Amazon Kindle	BOOKS_AND_REFERENCE	100000000	4.73	1	119.12	10	44
com.antivirus	AVG AntiVirus	TOOLS	100000000	4.71	4	72.73	8	6
com.application.zomato	Zomato: Food Delivery	FOOD_AND_DRINK	100000000	4.51	2	83.64	8	10
com.apusapps.launcher	APUS System: Theme Launcher	PERSONALIZATION	100000000	4.47	1	31.82	5	46
com.bigwinepot.nwdn.international	Remini - AI Photo Enhancer	PHOTOGRAPHY	100000000	4.04	4	261.95	10	7
com.billiards.city.pool.nation.club	Pooking - Billiards City	GAME_SPORTS	100000000	4.55	1	83.33	5	19
com.bradesco	Bradesco: Conta, Cartão e Pix!	FINANCE	100000000	4.51	3	102.67	15	32
com.canva.editor	Canva: Design, Art	ART_AND_DESIGN	100000000	4.81	3	27.11	4	4
com.cyberlink.youcammakeup	YouCam Makeup - Selfie Editor	PHOTOGRAPHY	100000000	4.32	3	113.71	6	16
com.daraz.android	Daraz Online Shopping App	SHOPPING	100000000	4.17	3	69.43	6	46
com.devuni.flashlight	Tiny Flashlight + LED	TOOLS	100000000	4.58	4	6.32	2	0
com.discord	Discord - Talk, Play, Hang Out	COMMUNICATION	100000000	3.13	4	133.66	4	66
com.dubox.drive	TeraBox: Cloud Storage Space	TOOLS	100000000	4.51	4	79.0	10	34
com.dvloper.granny	Granny	GAME_ARCADE	100000000	4.27	2	164.84	2	3
com.dywx.larkplayer	Lark Player:Music Player	MUSIC_AND_AUDIO	100000000	4.5	1	20.37	6	19
com.ea.game.nfs14_row	Need for Speed™ No Limits	GAME_RACING	100000000	4.43	3	186.77	5	10
com.ea.game.simcitymobile_row	SimCity BuildIt	GAME_SIMULATION	100000000	4.25	5	205.67	7	26
com.ea.games.simsfreeplay_row	The Sims™ FreePlay	GAME_SIMULATION	100000000	4.13	1	70.26	5	20
com.ebay.mobile	eBay online shopping	SHOPPING	100000000	4.7	4	139.73	7	8
com.fdgentertainment.bananakong	Banana Kong	GAME_ACTION	100000000	4.62	2	121.67	4	3
com.firsttouchgames.story	Score! Hero - Soccer Games	GAME_SPORTS	100000000	4.36	7	243.83	5	8
com.frontrow.vlog	VN - Video Editor	VIDEO_PLAYERS	100000000	4.72	4	223.82	11	42
com.gameloft.android.ANMP.GloftA9HM	Asphalt Legends Unite	GAME_RACING	100000000	4.42	14	3263.62	6	17
com.global.foodpanda.android	foodpanda: food	FOOD_AND_DRINK	100000000	4.02	4	67.63	7	1
com.gojek.app	Gojek - Food	TRAVEL_AND_LOCAL	100000000	4.34	3	87.07	13	10
com.grabtaxi.passenger	Grab - Taxi	TRAVEL_AND_LOCAL	100000000	4.81	4	282.96	21	23
com.indeed.android.jobsearch	Indeed Job Search	BUSINESS	100000000	4.68	4	68.8	3	55
com.instabridge.android	Instabridge: WiFi Map	PRODUCTIVITY	100000000	4.16	4	175.3	17	26
com.intsig.camscanner	CamScanner- scanner, PDF maker	PRODUCTIVITY	100000000	4.79	3	174.09	11	49
com.ismaker.android.simsimi	SimSimi	ENTERTAINMENT	100000000	4.29	3	95.45	4	70
com.ixigo.train.ixitrain	ixigo Trains: Ticket Booking	TRAVEL_AND_LOCAL	100000000	4.57	3	30.77	8	2

Continued on next page

Table B.1 – *continued from previous page*

Package ID	Name	Category	Installs	Rating	Splits	Size (MB)	DEX	Native
com.joeware.android.gpulumera	Candy Camera - photo editor	PHOTOGRAPHY	100000000	4.47	2	203.81	3	24
com.kakao.talk	KakaoTalk : Messenger	COMMUNICATION	100000000	4.06	4	193.15	22	27
com.king.petrescuesaga	Pet Rescue Saga	GAME_CASUAL	100000000	4.48	4	159.87	4	3
com.lazada.android	Lazada 8.8	SHOPPING	100000000	4.7	3	66.13	6	41
com.mediocre.smashhit	Smash Hit	GAME_ARCADE	100000000	4.45	1	80.7	3	20
com.melesta.coffeeshop	My Cafe — Restaurant Game	GAME_CASUAL	100000000	4.54	1	81.8	7	12
com.mercadolibre	Mercado Libre: Compras online	SHOPPING	100000000	4.57	3	43.9	8	6
com.mercadopago.wallet	Mercado Pago: cuenta digital	FINANCE	100000000	4.73	5	68.88	24	12
com.microblink.photomath	Photomath	EDUCATION	100000000	4.44	3	17.31	2	5
com.microsoft.teams	Microsoft Teams	BUSINESS	100000000	4.6	4	166.49	8	63
com.midasplayer.apps.bubblewitchsaga2	Bubble Witch 2 Saga	GAME_CASUAL	100000000	4.59	4	100.1	4	3
com.miniclip.carrom	Carrom Pool: Disc Game	GAME_SPORTS	100000000	4.54	2	144.4	13	17
com.mojang.minecrafttrialpe	Minecraft Trial	GAME_ARCADE	100000000	4.0	5	422.4	2	4
com.myairtelapp	Airtel Thanks: Recharge	FINANCE	100000000	4.13	3	45.49	5	45
com.myntra.android	Myntra - Fashion Shopping App	SHOPPING	100000000	4.37	3	39.59	2	57
com.netmarble.mherosgb	MARVEL Future Fight	GAME_ROLE_PLAYING	100000000	3.92	2	103.54	2	11
com.nexstreaming.app.kinemasterfree	KineMaster-Video Editor	VIDEO_PLAYERS	100000000	4.22	4	124.18	6	16
com.opera.browser	Opera browser with AI	COMMUNICATION	100000000	4.64	6	139.23	6	11
com.piriform.ccleaner	CCleaner – Phone Cleaner	TOOLS	100000000	4.6	4	54.02	6	2
com.pixonic.wwr	War Robots Multiplayer Battles	GAME_ACTION	100000000	4.27	2	158.58	11	21
com.playmini.miniworld	Mini World: CREATA	GAME_ADVENTURE	100000000	4.11	5	1138.97	9	29
com.playrix.township	Township	GAME_CASUAL	100000000	4.74	5	250.11	5	5
com.radio.pocketfm	Pocket FM: Audio Series	MUSIC_AND_AUDIO	100000000	4.53	4	105.15	12	10
com.reddit.frontpage	Reddit	SOCIAL	100000000	4.05	4	109.61	11	5
com.robtopx.geometryjumplite	Geometry Dash Lite	GAME_ARCADE	100000000	4.32	6	202.89	7	3
com.sandboxol.blockymods	Blockman Go	GAME_ARCADE	100000000	4.24	4	660.34	9	28
com.sega.sonicdash	Sonic Dash - Endless Running	GAME_ARCADE	100000000	4.65	3	258.92	7	5
com.sirma.mobile.bible.android	YouVersion Bible App + Audio	BOOKS_AND_REFERENCE	100000000	4.92	4	52.39	2	4
com.smule.singandroid	Smule: Karaoke Songs	MUSIC_AND_AUDIO	100000000	3.85	3	74.25	7	8
com.socialnmobile.dictapps.notepad.color.note	ColorNote Notepad Notes	PRODUCTIVITY	100000000	4.87	1	4.21	1	0
com.superking.parchisi.star	Parchisi STAR Online	GAME_BOARD	100000000	4.4	3	136.71	6	9
com.taxis99	99 - Private Driver and Taxi	MAPS_AND_NAVIGATION	100000000	4.21	3	111.14	11	62
com.teacapps.barcodescanner	QR	PRODUCTIVITY	100000000	4.57	3	7.7	2	0
com.tumblr	Tumblr—Fandom, Art, Chaos	SOCIAL	100000000	4.0	3	59.12	9	12

Continued on next page

Table B.1 – *continued from previous page*

Package ID	Name	Category	Installs	Rating	Splits	Size (MB)	DEX	Native
com.ubercab.driver	Uber - Driver: Drive	BUSINESS	100000000	4.57	6	162.78	34	17
com.utorrent.client	µTorrent®- Torrent Downloader	VIDEO_PLAYERS	100000000	4.57	4	72.67	7	14
com.vectorunit.purple.googleplay	Beach Buggy Racing	GAME_RACING	100000000	4.47	4	116.96	7	17
com.vkontakte.android	VK: music, video, messenger	SOCIAL	100000000	3.79	3	175.24	20	35
com.weather.Weather	The Weather Channel - Radar	WEATHER	100000000	4.66	1	85.23	4	18
com.xvideostudio.videoeditor	Video Editor	VIDEO_PLAYERS	100000000	4.55	1	133.34	6	32
com.yahoo.mobile.client.android.mail	Yahoo Mail – Organized Email	COMMUNICATION	100000000	4.54	4	61.94	6	4
com.zeptolab.ctr.ads	Cut the Rope	GAME_PUZZLE	100000000	4.48	4	122.09	7	10
cris.org.in.prs.ima	IRCTC Rail Connect	TRAVEL_AND_LOCAL	100000000	3.62	1	42.38	4	8
ee.mtakso.client	Bolt: Request a Ride	MAPS_AND_NAVIGATION	100000000	4.74	4	79.26	9	9
fb.video.downloader	Video Downloader For Facebook	TOOLS	100000000	4.14	1	11.16	2	4
free.vpn.unblock.proxy.turbovpn	Turbo VPN - Secure VPN Proxy	TOOLS	100000000	4.73	1	27.54	4	15
in.swiggy.android	Swiggy: Food Instamart Dineout	FOOD_AND_DRINK	100000000	4.33	3	100.56	11	55
io.wifimap.wifimap	WiFi Map®: Internet, eSIM, VPN	PRODUCTIVITY	100000000	4.46	3	194.94	19	29
jp.garud.ssimulator	SAKURA School Simulator	GAME_SIMULATION	100000000	4.28	3	289.41	4	13
org.iggymedia.periodtracker	Flo Period	HEALTH_AND_FITNESS	100000000	4.68	4	124.34	7	7
org.mozilla.firefox	Firefox Fast	COMMUNICATION	100000000	4.57	3	118.2	3	18
ru.yandex.taxi	Yandex Go: Taxi Food Delivery	MAPS_AND_NAVIGATION	100000000	0.0	3	185.32	9	15
videoeditor.videorecorder.screenrecorder	Screen Recorder - XRecorder	VIDEO_PLAYERS	100000000	4.76	4	35.96	2	18
wp.wattpad	Wattpad - Read	BOOKS_AND_REFERENCE	100000000	4.11	4	182.47	17	11
br.gov.serpro.cnhe	Carteira Digital de Trânsito	TOOLS	50000000	4.73	6	49.27	2	2

References

[1] Sep 2024. https://play.google.com/store/apps/details?id=com.applisto.appcloner.premium. viii, 1, 8

[2] Sep 2024. https://play.google.com/store/apps/details?id=com.modheaven. viii, 1, 8

[3] Jung, Jin Hyuk, Ju Young Kim, Hyeong Chan Lee, and Jeong Hyun Yi: *Repackaging attack on android banking applications and its countermeasures*. Wireless Personal Communications, 73(4):1421–1437, 2013. xv, 10

[4] I, Pavlov, Oct 2025. https://www.7-zip.org/. xv, 10, 11

[5] Kalinin, Dmitry: *Analysis of a spy module inside a whatsapp mod*, Nov 2023. https://securelist.com/spyware-whatsapp-mod/110984/. xv, 18, 19, 20

[6] Lyu, Fang, Yapin Lin, Junfeng Yang, and Junhai Zhou: *Suidroid: An efficient hardening-resilient approach to android app clone detection*. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 511–518. IEEE, 2016. 1, 14

[7] Shen, Yun, Pierre Antoine Vervier, and Gianluca Stringhini: *A large-scale temporal measurement of android malicious apps: Persistence, migration, and lessons learned*. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1167–1184, 2022. 1

[8] Khanmohammadi, Kobra, Neda Ebrahimi, Abdelwahab Hamou-Lhadj, and Raphaël Khoury: *Empirical study of android repackaged applications*. Empirical Software Engineering, 24:3587–3629, 2019. 1, 36

[9] Zhou, Yajin and Xuxian Jiang: *Dissecting android malware: Characterization and evolution*. In *2012 IEEE symposium on security and privacy*, pages 95–109. IEEE, 2012. 1

[10] Allix, Kevin, 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*, pages 468–471, 2016. 1, 11, 59

[11] Li, Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro: *Understanding android app piggybacking*. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 359–361. IEEE, 2017. 1, 6, 12, 13, 28, 36

[12] Crussell, Jonathan, Clint Gibler, and Hao Chen: *Attack of the clones: Detecting cloned applications on android markets*. In *Computer Security–ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings 17*, pages 37–54. Springer, 2012. 1, 15

[13] Crussell, Jonathan, Clint Gibler, and Hao Chen: *Scalable semantics-based detection of similar android applications*. In *Proc. of ESORICS*, volume 13. Citeseer, 2013. 1

[14] Luo, Lannan, Yu Fu, Dinghao Wu, Sencun Zhu, and Peng Liu: *Repackage-proofing android apps*. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 550–561. IEEE, 2016. 1, 2, 12, 14, 16, 17, 27, 38

[15] Li, Li, Tegawendé F Bissyandé, and Jacques Klein: *Rebooting research on detecting repackaged android apps: Literature review and benchmark*. IEEE Transactions on Software Engineering, 47(4):676–693, 2019. 1, 6, 12, 13, 14, 15

[16] Zhou, Wu, Yajin Zhou, Xuxian Jiang, and Peng Ning: *Detecting repackaged smartphone applications in third-party android marketplaces*. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317–326, 2012. 1

[17] Ma, Jun, Qing Wei Sun, Chang Xu, and Xian Ping Tao: *Griddroid—an effective and efficient approach for android repackaging detection based on runtime graphical user interface*. Journal of Computer Science and Technology, 37(1):147–181, 2022. 1, 16

[18] Samhi, Jordan and Andreas Zeller: *Androlog: Android instrumentation and code coverage analysis*. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 597–601, 2024. 1, 2, 7, 13, 14, 20, 21, 22, 28, 46, 60

[19] Daian, Philip, Ylies Falcone, Patrick Meredith, Traian Florin Şerbănuţă, Shin'ichi Shiriashi, Akihito Iwai, and Grigore Rosu: *Rv-android: Efficient parametric android runtime verification, a brief tutorial*. In *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings*, pages 342–357. Springer, 2015. 1, 7, 13, 21, 22, 28

[20] Romdhana, Andrea, Mariano Ceccato, Gabriel Claudiu Georgiu, Alessio Merlo, and Paolo Tonella: *Cosmo: Code coverage made easier for android*. In *2021 14th IEEE conference on software testing, verification and validation (ICST)*, pages 417–423. IEEE, 2021. 1, 21, 22, 46, 60

[21] Pilgun, Aleksandr, Olga Gadyatskaya, Stanislav Dashevskyi, Yury Zhauniarovich, and Artsiom Kushniarou: *An effective android code coverage tool*. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2189–2191, 2018. 1, 7

[22] Pilgun, Aleksandr, Olga Gadyatskaya, Yury Zhauniarovich, Stanislav Dashevskyi, Artsiom Kushniarou, and Sjouke Mauw: *Fine-grained code coverage measurement in automated black-box android testing*. ACM Transactions on Software Engineering and

Methodology (TOSEM), 29(4):1–35, 2020. 1, 4, 7, 12, 13, 14, 21, 22, 27, 28, 30, 46, 60

[23] Bao, Lingfeng, Tien Duy B Le, and David Lo: *Mining sandboxes: Are we there yet?* In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 445–455. IEEE, 2018. 1, 7, 20, 22, 46

[24] Kelty, Christopher M: *Two bits: The cultural significance of free software*. Duke University Press, 2020. 6

[25] Caneill, Matthieu and Stefano Zacchiroli: *Debsources: Live and historical views on macro-level software evolution*. In *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2014. 6

[26] Kim, Dongjin, Yesol Kim, Jeongoh Moon, Seong Je Cho, Jinwoon Woo, and Ilsun You: *Identifying windows installer package files for detection of pirated software*. In *2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 287–290. IEEE, 2014. 6

[27] Claes, Maelick, Tom Mens, Roberto Di Cosmo, and Jérôme Vouillon: *A historical analysis of debian package incompatibilities. in 2015 ieee/acm 12th working conference on mining software repositories (pp. 212-223)*, 2015. 6

[28] Gibler, Clint, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi: *Adrob: Examining the landscape and impact of android application plagiarism*. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 431–444, 2013. 6, 12, 13, 27

[29] Broadcom: *Trojanized notepad++ installer delivers malware.*, Dec 2021. https://www.broadcom.com/support/security-center/protection-bulletin/trojanized-notepad-installer-delivers-malware. 7

[30] Wolff, Evan D, KatE M GroWlEy, Maida O Lerner, Matthew B Welling, Michael G Gruden, and Jacob Canter: *Navigating the solarwinds supply chain attack*. Procurement Law., 56:3, 2021. 7

[31] Jamrozik, Konrad, Philipp von Styp-Rekowsky, and Andreas Zeller: *Mining sandboxes*. In *Proceedings of the 38th International Conference on Software Engineering*, pages 37–48, 2016. 7, 20, 21

[32] Cai, Haipeng and Barbara G Ryder: *Droidfax: A toolkit for systematic characterization of android applications*. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 643–647. IEEE, 2017. 7, 20, 21, 22

[33] Garg, Shivi and Niyati Baliyan: *Comparative analysis of android and ios from security viewpoint*. Computer Science Review, 40:100372, 2021. 8

[34] Soares, Alberto Magno Muniz: *Análise de objetos a partir da extração da memória ram de sistemas sobre android run-time (art)*. 2017. 8

[35] Nguyen, Trung, Kyungtae Kim, Antonio Bianchi, and Dave Jing Tian: *Truemu: an extensible, open-source, whole-system ios emulator*. Blackhat USA'22, 2022. 8

[36] Schütte, Julian and Dennis Titze: *lios: Lifting ios apps for fun and profit*. In *2019 International Workshop on Secure Internet of Things (SIOT)*, pages 1–10. IEEE, 2019. 8

[37] AndnixSH, Oct 2025. https://github.com/AndnixSH/APKToolGUI. 9

[38] *Android Penetration Tools Walkthrough Series: Apktool*. https://www.infosecinstitute.com/resources/penetration-testing/android-penetration-tools-walkthrough-series-apktool/. [Accessed 03-07-2025]. 9, 14

[39] Giedrimas, Vaidas and Samir Omanovič: *The impact of mobile architectures on component-based software engineering*. In *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, pages 1–6. IEEE, 2015. 9

[40] Yan, Jiwei, Shixin Zhang, Yepang Liu, Xi Deng, Jun Yan, and Jian Zhang: *A comprehensive evaluation of android icc resolution techniques*. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022. 9

[41] Anonymous: *About Android App Bundles | Android Developers — developer.android.com*. https://developer.android.com/guide/app-bundle, 2021. [Accessed 08-06-2024]. 10

[42] Bagheri, Hamid, Jianghao Wang, Jarod Aerts, Negar Ghorbani, and Sam Malek: *Flair: efficient analysis of android inter-component vulnerabilities in response to incremental changes*. Empirical Software Engineering, 26:1–37, 2021. 10, 11, 12

[43] Liu, Yi, Yun Ma, Xusheng Xiao, Tao Xie, and Xuanzhe Liu: *Legodroid: flexible android app decomposition and instant installation*. Science China Information Sciences, 66(4):142103, 2023. 10

[44] Alecci, Marco, Pedro JR Jiménez, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein: *Androzoo: A retrospective with a glimpse into the future. in 2024 ieee/acm 21st international conference on mining software repositories (msr)*. IEEE Computer Society, 2024. 11

[45] Merlo, Alessio, Antonio Ruggia, Luigi Sciolla, and Luca Verderame: *You shall not repackage! demystifying anti-repackaging on android*. Computers & Security, 103:102181, 2021. 12, 13, 14, 16, 28

[46] *How to merge splited apk files*, November 2022. https://xdaforums.com/t/how-to-merge-splited-apk-files.4529011/, visited on 2024-11-27. 12

[47] AndnixSH: *How to merge split apk's into standalone apk*, February 2024. https://platinmods.com/threads/how-to-merge-split-apks-into-standalone-apk.188936/, visited on 2024-11-27. 12

[48] LuigiVampa92: *Github - luigivampa92/merge-apks: Simple python script that merges multiple "splitted" apk files into a single universal "fat" apk file that contains all native libraries for all architectures, all dpi-dependent resources, strings, etc*, 2023. https://github.com/LuigiVampa92/merge-apks, visited on 2024-11-27. 12

[49] Li, Yuanchun, Ziyue Yang, Yao Guo, and Xiangqun Chen: *Droidbot: a lightweight ui-guided test input generator for android.* In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26. IEEE, 2017. 13, 27, 60, 61

[50] Berlato, Stefano and Mariano Ceccato: *A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps.* Journal of Information Security and Applications, 52:102463, 2020. 13, 27

[51] Wang, Xueqiang, 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 23)*, pages 3403–3420, 2023. 13, 27

[52] *Pushing Modified framework-res.apk?* https://xdaforums.com/t/q-pushing-modified-framework-res-apk.3892218/. [Accessed 03-07-2025]. 14

[53] Ma, Haoyu, Shijia Li, Debin Gao, Daoyuan Wu, Qiaowen Jia, and Chunfu Jia: *Active warden attack: On the (in) effectiveness of android app repackage-proofing.* IEEE Transactions on Dependable and Secure Computing, 19(5):3508–3520, 2021. 14, 16, 38

[54] Wang, Yan and Atanas Rountev: *Who changed you? obfuscator identification for android.* In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 154–164. IEEE, 2017. 14

[55] Salem, Aleieldin, F Franziska Paulus, and Alexander Pretschner: *Repackman: A tool for automatic repackaging of android apps.* In *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*, pages 25–28, 2018. 14

[56] Desnos, Anthony and Geoffroy Gueguen: *Android: From reversing to decompilation.* Proc. of Black Hat Abu Dhabi, 1:1–24, 2011. 14, 15, 26

[57] Sun, Xin, Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie: *Detecting code reuse in android applications using component-based control flow graph.* In *IFIP international information security conference*, pages 142–155. Springer, 2014. 15

[58] Ren, Chuangang, Kai Chen, and Peng Liu: *Droidmarking: resilient software watermarking for impeding android application repackaging.* In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 635–646, 2014. 15, 17

[59] Shao, Yuru, 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*, pages 56–65, 2014. 15

[60] Tian, Ke, Danfeng Yao, Barbara G Ryder, Gang Tan, and Guojun Peng: *Detection of repackaged android malware with code-heterogeneity features*. IEEE Transactions on Dependable and Secure Computing, 17(1):64–77, 2017. 15

[61] Gonzalez, Hugo, Andi A Kadir, Natalia Stakhanova, Abdullah J Alzahrani, and Ali A Ghorbani: *Exploring reverse engineering symptoms in android apps*. In *Proceedings of the Eighth European Workshop on System Security*, pages 1–7, 2015. 15

[62] Sep 2024. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. 15

[63] Protsenko, Mykola, Sebastien Kreuter, and Tilo Müller: *Dynamic self-protection and tamperproofing for android apps using native code*. In *2015 10th International Conference on Availability, Reliability and Security*, pages 129–138. IEEE, 2015. 16

[64] Huang, Heqing, Sencun Zhu, Peng Liu, and Dinghao Wu: *A framework for evaluating mobile app repackaging detection algorithms*. In *Trust and Trustworthy Computing: 6th International Conference, TRUST 2013, London, UK, June 17-19, 2013. Proceedings 6*, pages 169–186. Springer, 2013. 16, 21

[65] Collberg, Christian S. and Clark Thomborson: *Watermarking, tamper-proofing, and obfuscation-tools for software protection*. IEEE Transactions on software engineering, 28(8):735–746, 2002. 16

[66] Ma, Haoyu, Shijia Li, Debin Gao, and Chunfu Jia: *Secure repackage-proofing framework for android apps using collatz conjecture*. IEEE Transactions on Dependable and Secure Computing, 19(5):3271–3285, 2021. 17

[67] Inc., Google: *Play integrity and signing services*. https://developer.android.com/google/play/integrity?hl=en. 17

[68] Lakshmanan, Ravie, Oct 2025. https://thehackernews.com/2025/08/google-to-verify-all-android-developers.html. 17

[69] Inc., Apple: *App code signing process in ios, ipados, tvos, watchos, and visionos*. https://support.apple.com/en/guide/security/sec7c917bf14/web. 17

[70] Inc., Apple: *App review - distribute*. https://developer.apple.com/distribute/app-review/. 17

[71] Anonymous: *Decrypted ios ipa app store*, 2025. https://armconverter.com/decryptedappstore/us, Accessed: 11 May 2025. 18

[72] Merali, Alameen Karim: *Malware analysis of gbwhatsapp*, Nov 2023. https://medium.com/@brotheralameen/malware-analysis-of-gbwhatsapp-21e4b70c7bb2. 18

[73] Avelino, Yan: *Yowhatsapp, cópia maliciosa do whatsapp, É pego roubando contas de usuários • tecnoblog*, Oct 2022. https://tecnoblog.net/noticias/yowhatsapp-copia-maliciosa-do-whatsapp-e-pego-roubando-contas-de-usuarios/. 18

[74] Wahaz, Rizaldi, Rakha Nadhifa Harmana, Amiruddin Amiruddin, and Ardya Suryadinata: *Is whatsapp plus malicious? a review using static analysis*. In *2021 6th International Workshop on Big Data and Information Security (IWBIS)*, pages 91–96. IEEE, 2021. 18

[75] Mussolino, Domenico: *Fouad whatsapp, scraping through innovation*, Apr 2023. https://www.cybercrimeclues.com/fouad-whatsapp-scraping-through-innovation/. 18

[76] Anonymous: *Fouad wa e fm whatsapp (fmwa) versão mais recente do apk (oficial) abril de 2025 [anti-ban] - fouad whatsapp*, 2025. https://fouadmods.net/pt/fouad-whatsapp-pt/#fouad-wa, Accessed: 11 May 2025. 18

[77] Anonymous: *Download gbwhatsapp apk latest version (virus free) 2025*, 2025. https://gbappsz.com.pk/download-gbwhatsapp/, Accessed: 11 May 2025. 18

[78] Anonymous: *Whatsapp plus apk download (official) latest version may 2025 (updated)*, 2025. https://gbappsz.com.pk/whatsapp-plus-apk/, Accessed: 11 May 2025. 18

[79] Anonymous: *Yo whatsapp apk download latest for android (updated) may 2025*, 2025. https://gbappsz.com.pk/yo-whatsapp/, Accessed: 11 May 2025. 18

[80] Tikir, Mustafa M and Jeffrey K Hollingsworth: *Efficient instrumentation for code coverage testing*. ACM SIGSOFT Software Engineering Notes, 27(4):86–96, 2002. 19, 20

[81] Császár, István Attila and Radu Razvan Slavescu: *Building fast and reliable reverse engineering tools with frida and rust*. In *2022 IEEE 18th International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 289–294. IEEE, 2022. 20, 21, 60

[82] Bellizzi, Jennifer, Mark Vella, Christian Colombo, and Julio Hernandez-Castro: *Responding to targeted stealthy attacks on android using timely-captured memory dumps*. IEEE Access, 10:35172–35218, 2022. 21

[83] Vallée-Rai, Raja, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan: *Soot: A java bytecode optimization framework*. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010. 21, 22

[84] Vallee-Rai, Raja and Laurie J Hendren: *Jimple: Simplifying java bytecode for analyses and transformations*. Technical report, Technical report, McGill University, 1998. 21, 22, 60

[85] Neuner, Sebastian, Victor Van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar Weippl: *Enter sandbox: Android sandbox comparison*. arXiv preprint arXiv:1410.7749, 2014. 21

[86] *Android Market data, History, Rankings*. https://www.androidrank.org/. [Accessed 21-08-2024]. 37

[87] Wang, Wenyu, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie: *An empirical study of android test generation tools in industrial cases*. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 738–748, 2018. 39

[88] Riganelli, Oliviero, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani: *Data loss detector: automatically revealing data loss bugs in android apps*. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 141–152, 2020. 39

[89] Patel, Priyam, Gokul Srinivasan, Sydur Rahaman, and Iulian Neamtiu: *On the effectiveness of random testing for android: or how i learned to stop worrying and love the monkey*. In *Proceedings of the 13th International Workshop on Automation of Software Test*, pages 34–37, 2018. 41

[90] Nguyen, Trung Tin and Ben Stock: *Open access alert: Studying the privacy risks in android webview's web permission enforcement*. 2025. 59

[91] Jiménez, Pedro Jesús Ruiz, Jordan Samhi, Tegawendé F Bissyandé, and Jacques Klein: *Dissecting apks from google play: Trends, insights and security implications*. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 728–739. IEEE, 2025. 59