CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Manifest-Level Permission Debloating on Android Applications

Master's thesis in Computer science and engineering

Sotiri De Pinto

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Manifest-Level Permission Debloating on Android Applications

Sotiri De Pinto

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Manifest-Level Permission Debloating on Android Applications
Sotiri De Pinto

Supervisor: Ahmed Ali-Eldin Hassan, Computer Science and Engineering
Examiner: Ahmed Ali-Eldin Hassan, Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Manifest-Level Permission Debloating on Android Applications
Sotiri De Pinto
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Modern Android applications frequently contain functionality that most users never need or use, a phenomenon known as software bloat. One specific manifestation of bloat is permission over-declaration, where applications declare permissions in their manifest whose corresponding Android API methods are never invoked in the application's code. Over-declared permissions mislead users, unnecessarily expand the attack surface of the application, and may signal access to sensitive resources such as the camera or microphone without any functional justification.

This thesis proposes and evaluates a two-phase hybrid debloating pipeline for Android applications. The first phase is a custom static analysis tool that automatically detects over-declared permissions by scanning the application's smali bytecode for permission-protected API calls, and removes unused permission declarations from the manifest without modifying any executable code. The second phase applies MiniMon, an existing monitor-based debloating framework, to the output of the first phase, removing methods that were not exercised during monitored usage sessions.

The pipeline is evaluated on a dataset of 38 real-world Android applications. The results show that permission over-declaration is widespread, affecting 44.7% of the applications in the dataset. The first phase reduced APK size in 16 out of 17 affected applications with an average reduction of 122.14 KB, while preserving functional correctness by construction. The combined pipeline produced additional size reduction over monitor-based debloating alone in 11 out of 28 applications, demonstrating that the two approaches are complementary: permission-based debloating addresses manifest-level bloat that is invisible to usage-based approaches, while monitor-based debloating addresses code-level bloat that manifest analysis cannot reach.

Keywords: Android, debloating, permissions, static analysis, software bloat, APK, manifest, security.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Ahmed Ali-Eldin Hassan, for his guidance, feedback, and support throughout this thesis project. His input helped shape both the direction and the quality of this work.

I would also like to thank the authors of MiniMon for generously providing their tool, replication package, and execution logs, without which the second phase of this work would not have been possible.

Finally, I would like to thank my family and friends for their encouragement and support throughout my studies.

Sotiri De Pinto, Gothenburg, 2026-07-06

Contents

List of Figures

List of Tables

1

Introduction

Modern Android applications often contain numerous features designed to support diverse user needs. These features include different usage for different users, support for different hardware configurations, and integration of third-party libraries. On the one hand, this design increases flexibility and usability, but on the other side, it also introduces a significant amount of unused or rarely used functionality, commonly referred to as software bloat [1].

Software bloat arises because developers tend to include more features than any single user requires. As a result, a big portion of an application's code may never be executed. This phenomenon happens often in mobile applications, where apps are designed to serve many people where everyone is using the apps in their own ways. Thus, many installed applications contain redundant code that is not helping the user's experience.

In the context of Android, software bloat can also be seen on the permission-dependent features. Applications often request permissions for various functionalities in order to enable optional features, even though users may not utilize all of them. This creates opportunities for removing unnecessary code associated with unused permissions [2]. The presence of this bloat has important implications for security, privacy, and performance.

From a security perspective, unused code increases the attack surface of an application. Even if certain features are not actively used, vulnerabilities within their implementation can still be used by an attacker. Reducing the amount of code in an application can decrease the likelihood of security breaches by eliminating possible entry points for attackers [3].

In terms of privacy, unnecessary features sometimes rely on sensitive permissions such as access to location, microphone, or storage. The application usually request permissions that are not essential to the user's needs [4]. These permissions may expose sensitive information without a clear reason. Removing this unused functionality can help reduce privacy risks.

Performance is also affected by software bloat. Larger applications require more memory, storage space, and energy, all of which are important variables in mobile

environments [1]. Additionally, bloated applications may need more time to start run, as well as reducing the responsiveness.

Given all these negative effects of software bloat, it is a good idea to develop techniques that can automatically remove unnecessary code from Android applications. This process, known as application debloating, aims to produce an improved version of the app in which the bloated code is removed, while the required behavior does not change.

Traditional approaches to debloating often rely on static program analysis, such as constructing complete call graphs to identify unreachable code. However, these methods can be computationally expensive [5], [6] and may struggle to accurately capture real-world application behavior.

To address these limitations, recent work has explored alternative strategies. One direction is monitor based debloating, where systems such as MiniMon [7] monitor the execution of an application from a user, to identify which parts of the code are actually used. Another approach focuses on permission-based debloating, like MiniAppPerm [2], where functionality associated with disallowed permissions is removed, and the features of these functionalities cannot be executed. These approaches highlight the potential for combining monitor-based and permission-based debloating techniques to achieve more effective and efficient debloating.

The main objective of this thesis is to investigate whether manifest-level permission-based debloating can serve as an effective preprocessing step for monitor-based debloating of Android applications. Specifically, the thesis proposes and evaluates a two-phase pipeline in which a static analysis tool first identifies permissions declared in the `AndroidManifest.xml` (a configuration file bundled with every Android app that declares its permissions, components, and other metadata) that are never invoked in the application's bytecode, and removes them from the manifest. The output of this first phase is then passed to MiniMon [7], a dynamic, monitor-based debloating framework that performs method-level debloating guided by pre-collected runtime execution logs.

A key distinction of the first phase is that it targets over-declared permissions. These are permissions that the developer declared but whose corresponding Android API methods are never called anywhere in the application's *smali* bytecode, a human-readable text representation of the compiled Dalvik bytecode that Android apps run as. This is different from approaches that remove permissions the user wishes to disallow [2]. Over-declared permissions serve no functional purpose and can be removed safely without modifying any executable code, making the approach conservative by construction.

The second phase complements this static analysis with a dynamic, usage-driven perspective. Rather than relying on manifest declarations, MiniMon [7] observes which methods of the application are actually exercised during a real usage session,

and treats methods that are never triggered, and are not closely related to triggered methods through its clustering-based specification, as candidates for removal. This targets a different source of bloat than Phase 1: over-declared permissions are unused regardless of how the app is used, whereas Phase 2 identifies code that may be entirely permission-compliant but is nonetheless never exercised by a given user. Applying Phase 2 after Phase 1 therefore extends debloating from the manifest level down to the level of individual methods, without requiring any additional device interaction beyond the pre-collected execution logs used in this thesis.

The thesis investigates three research questions. First, how often do real-world Android applications have over-declared permissions. Second, whether manifest-level permission removal can reduce APK (Android Package Kit) size and attack surface without affecting application functionality. Third, whether applying permission-based debloating as a preprocessing step produces better debloating outcomes than running monitor-based debloating alone.

This thesis makes the following contributions to the field of Android application debloating. First of all, it presents the design and implementation of a static permission-based debloating tool that automatically detects and removes over-declared permissions from Android applications through smali bytecode scanning, without requiring any user input or code modification. The tool operates by mapping each declared permission to the set of Android API signatures that require it, scanning the decoded smali bytecode for any invocation of those signatures, and removing from the manifest any permission whose API is never called. The tool was evaluated on a dataset of 38 real-world Android applications, where it detected over-declared permissions in 44.7% of applications and achieved an average APK size reduction of 122.14 KB among affected applications.

It proposes a two-phase hybrid debloating pipeline in which the permission-based tool serves as a preprocessing step for MiniMon [7]. The pipeline is evaluated against a baseline of running MiniMon alone on the original APKs, providing a direct comparison of the two approaches across 28 applications for which execution logs were available. The combined pipeline produced additional size reduction over MiniMon alone in 11 out of 28 applications, demonstrating that the two approaches are complementary.

Finally, the thesis provides an empirical analysis of permission over-declaration patterns in real-world Android applications, identifying `VIBRATE` and `ACCESS_WIFI_STATE` as the most commonly over-declared permissions, and highlighting that dangerous permissions including `CAMERA`, `RECORD_AUDIO`, and `READ_PHONE_STATE` are also found among over-declared permissions in the dataset, with direct implications for user privacy and application security.

2

Background

This chapter introduces the two tools that form the technical foundation of this thesis. Section 2.1 describes MiniMon, the monitor-based debloating framework used as the second phase of the proposed pipeline. Section 2.2 describes MiniAppPerm, a permission-based debloating tool that is not used directly in this work but is the most closely related prior approach to Phase 1. Section 2.3 discusses how these two tools relate to the pipeline proposed in this thesis.

2.1 MiniMon

MiniMon [7] is a monitoring-based debloating framework for Android applications. Rather than relying solely on static analysis to estimate what code is needed, MiniMon observes how a specific user actually interacts with an app and debloats it accordingly.

The pipeline has three phases. First, MiniMon instruments the APK by inserting logging probes at the start of each method using the Soot framework [6], a program analysis and bytecode transformation framework for Java, producing an instrumented version of the app. Users install this instrumented APK and use it normally, during which the executed method signatures are recorded in a log. In parallel, MiniMon builds a call graph of the application using FlowDroid [5], which models Android-specific entry points such as lifecycle callbacks and inter-component communication.

Given the logged methods and the call graph, MiniMon applies a clustering-based specification to identify additional methods that the user is likely to need but may not have triggered during monitoring. This handles the case where a feature is desired but was not exercised in the usage session. Each method is encoded as a weighted vector using TF-IDF (Term Frequency–Inverse Document Frequency) over static execution paths derived from the call graph, and methods are grouped into clusters based on their behavioural similarity through hierarchical clustering. Methods that fall into the same cluster as an observed method are considered related to user-desired functionality and marked for retention. MiniMon tests ten similarity thresholds and selects the one that maximises the F-debloat score, a weighted harmonic mean of recall and debloating rate that prioritises recall to reduce the risk of removing needed functionality.

Once the set of methods to keep is determined, MiniMon stubs out all remain-

ing methods using Soot and produces a signed, debloated APK. The threshold for similarity can be adjusted, allowing users to trade off aggressiveness of debloating against risk of removing needed functionality.

2.2 MiniAppPerm

MiniAppPerm [2] is a tool for permission-based debloating of Android applications. It takes as input an APK and a list of permissions that the user wishes to disallow, then removes all code associated with those permissions from the application.

The core insight is that if a permission is disallowed, the features that depend on it will never execute. This makes their corresponding code safe to remove without affecting any functionality the user actually needs. The challenge is identifying precisely which methods depend on a given permission.

Rather than constructing a complete call graph of the entire application, MiniAppPerm builds a partial call graph containing only the methods reachable from permission-related API calls. It first uses a permission-to-API mapping derived from PScout [8] to identify which Android API methods require each disallowed permission. It then scans the app's bytecode using `dexdump`, an Android SDK (Software Development Kit) tool that disassembles compiled DEX (Dalvik executable) bytecode into readable text, to find invocations of those APIs, and performs a targeted backwards analysis using BackDroid, a backward program slicing tool, to build a partial call graph rooted at those call sites. A forward and backward slicer then traverses this partial graph to identify all methods that can be safely removed. Finally, the Debloater component stubs out those methods and repackages the APK.

This approach significantly reduces call graph construction time compared to tools that rely on a full FlowDroid analysis. In preliminary experiments on ten real-world apps, MiniAppPerm reduced call graph construction time by up to 85.3% while producing debloated apps that installed and ran without crashes.

2.3 Relationship to This Work

The two tools described in this section form the foundation of the approach proposed in this thesis. MiniMon [7] is used directly as the second phase of the pipeline, providing method-level debloating guided by runtime execution logs. MiniAppPerm [2] is not used directly, but it is the most closely related work to Phase 1 of the pipeline. Both Phase 1 and MiniAppPerm are built on the same observation, that each Android permission corresponds to a set of API calls, and that the presence or absence of those calls in the bytecode determines whether the permission is needed or not.

The key difference is in what triggers the removal. MiniAppPerm targets permissions that a user explicitly wishes to disallow, removing the code associated with those permissions even if it is actively used in the bytecode. Phase 1 of this thesis targets a different and complementary problem, permissions that the developer accidentally over-declared and that are never invoked anywhere in the bytecode. This distinction means the two approaches address different sources of permission-related bloat and can in principle be combined. The background provided in this section therefore serves two purposes: it introduces MiniMon as a tool whose internals are directly relevant to understanding Phase 2, and it contextualises MiniAppPerm as the prior work against which Phase 1 is most naturally positioned.

3

Related Work

This chapter surveys existing techniques for debloating Android applications. Debloating techniques can be categorized based on how they identify removable functionality, including monitor-based, permission-based, and feature-oriented approaches. A key distinction also exists between static and dynamic analysis techniques, which differ in how they approximate application behavior. Section 3.1 discusses activity-based debloating. Section 3.2 discusses permission-based debloating approaches, including the most closely related prior work to Phase 1 of this thesis. Section 3.3 discusses dynamic debloating techniques that avoid static APK modification. Section 3.4 discusses the broader distinction between static and dynamic analysis and positions this thesis's approach within it.

3.1 Activity-based debloating

AutoDebloater [9] approaches debloating at the level of Android activities, which are the primary UI components through which users interact with an application. The tool automatically explores an app using StoryDistiller, which combines static call graph analysis with dynamic exploration via ADB (Android Debug Bridge) to generate an Activity Transition Graph (ATG). The ATG captures which activities exist in the app and how they can be reached from one another, along with screenshots of each activity.

Users are presented with the ATG through a web interface and select which activities to keep. AutoDebloater then removes the excluded activities and their associated methods using forward slicing on the call graph, implemented via the Soot framework [6]. Methods that are only reachable through removed activities are stubbed out, and the APK is repackaged. In a user study on five apps across three categories, the tool achieved an average debloating time of under 20 seconds and received a stability score of 4.8 out of 5.

Activity-based debloating is intuitive because activities map directly to visible features that users can recognize and reason about. However, it operates at a coarse granularity. Dependencies between activities may not be fully captured by the ATG, and removing an activity does not guarantee removal of all associated dead code deeper in the call graph.

3.2 Permission-based debloating

XDebloat [10] is a feature-oriented debloating framework that supports three feature location strategies, one of which is permission-based. Its permission-based approach uses PScout [8] to map Android permissions to the API calls that require them, then identifies code paths in the application that invoke those APIs. The associated code is collected into a feature, which developers can then choose to remove.

Beyond permission-based location, XDebloat also supports activity-based and modularity-based feature identification, the latter using the Louvain community detection algorithm [11], a graph-partitioning technique that groups densely connected nodes into communities by optimizing modularity, on the application's call graph. It supports both pruning-based debloating, which removes selected features from the APK, and module-based debloating, which restructures the app into Android instant apps or app bundles for on-demand delivery. Evaluated on 200 open-source and 1,000 commercial apps, XDebloat achieves average code reduction rates of around 32% and completes debloating within 10 minutes.

MiniAppPerm [2], discussed in Chapter 2, is specifically focused on improving the efficiency of the permission-based debloating step. Where XDebloat and earlier tools rely on a full FlowDroid [5] call graph, MiniAppPerm constructs only a partial call graph rooted at permission-related APIs, reducing call graph construction time by up to 85.3% in preliminary experiments.

3.3 Dynamic debloating

A different direction is taken by 3DNDroid [3], which proposes dynamic debloating without any static modification of the APK. Existing static debloating approaches repackage the APK and resign it, which breaks Android's built-in anti-tampering mechanisms such as public key verification and code integrity checks. In a pilot study, the authors found that 49 out of the top 200 Google Play apps could not be repackaged at all using current tools, and a further 31 repackaged APKs could not be installed or launched.

3DNDroid addresses this by operating at runtime inside a customized version of the Android OS. It intercepts DEX method invocations before they are passed to the interpreter, preventing their compilation and execution without modifying the APK file. For native library methods, it locates the target methods in memory during library loading and zero-fills their code, inserting a return instruction so that invocations do not crash the app. A management app communicates the debloating schema to the customized Android Runtime via a ContentProvider, and the schema can be updated at runtime without reinstalling the app.

Evaluated on 55 real-world apps, 3DNDroid successfully debloated all 187 targeted DEX methods and 30 native methods, and reduced over 13,000 potential

Return-Oriented Programming gadgets. It also demonstrated median CPU usage reductions of 14.2% for DEX debloating and 28.8% for native debloating compared to running the original apps. The main limitation is that it requires users to flash a custom Android OS image, which is impractical for personal devices running vendor-specific firmware.

3.4 Static vs dynamic analysis

A fundamental distinction in debloating techniques is whether they rely on static or dynamic analysis. Static approaches, including AutoDebloater, XDebloat, and MiniAppPerm, analyze the application's bytecode without executing it. They construct call graphs or data-flow models to identify code that is theoretically unreachable or permission-dependent. These methods can be applied without a device or usage data, but they tend to be conservative, retaining code that would never execute in practice, and can be computationally expensive for large applications.

Dynamic approaches observe actual execution behavior. MiniMon [7] instruments the app and collects method-level execution logs from real user sessions, identifying code that was never triggered. This allows more aggressive debloating but introduces a coverage dependency: features not exercised during monitoring may be incorrectly treated as unused. MiniMon addresses this partially through its clustering-based specification, which groups methods by behavioural similarity using TF-IDF encoding over the call graph.

The approach proposed in this thesis occupies a position between these two paradigms. The first phase is fully static, scanning smali bytecode to detect over-declared permissions without executing the app. The second phase is dynamic, relying on runtime logs collected by MiniMon. Combining the two allows the static phase to reduce the attack surface and clean the manifest before the dynamic phase performs finer-grained method-level debloating.

4

Methodology

This chapter presents the methodology used to investigate the research questions introduced in Section 4.1. Section 4.1 states the three research questions that guide this thesis. Section 4.2 describes the overall two-phase pipeline and the rationale behind its ordering. Section 4.3 details the design of Phase 1, the permission-based debloating step proposed in this thesis. Section 4.4 describes how Phase 2 applies MiniMon to the output of Phase 1. Section 4.5 describes the dataset used to evaluate the pipeline.

4.1 Research Questions

This thesis investigates whether combining permission-based static analysis with monitor-based dynamic debloating produces a more effective debloating pipeline than either approach applied in isolation. To guide the investigation, the following research questions are defined:

RQ1: To what extent do real-world Android applications declare permissions that are never invoked in their bytecode?

This question establishes the scope of the over-declaration problem. If few apps declare unused permissions, the motivation for a permission-based preprocessing step is weak. If over-declaration is common, it justifies the first phase of the pipeline.

RQ2: Can manifest-level permission debloating reduce APK size and attack surface without breaking application functionality?

This question evaluates the correctness and effectiveness of Phase 1 in isolation. A debloated APK is only useful if it installs and runs correctly. Size and permission count are used as proxies for attack surface reduction.

RQ3: Does applying permission-based debloating as a preprocessing step improve the results of monitor-based debloating compared to applying MiniMon on the original APK directly?

This is the central question of the thesis. It asks whether chaining the two phases produces measurably better outcomes than Phase 2 alone, in terms of additional size reduction or permission cleanup.

4.2 Overall Approach

The proposed approach is a two-phase hybrid debloating pipeline for Android applications. The first phase performs static, manifest-level permission debloating. The second phase applies MiniMon [7], a monitor-based debloating framework, to the output of the first phase. The two phases are chained such that the debloated APK produced by Phase 1 becomes the input to Phase 2. Figure 4.1 illustrates the complete pipeline end to end.

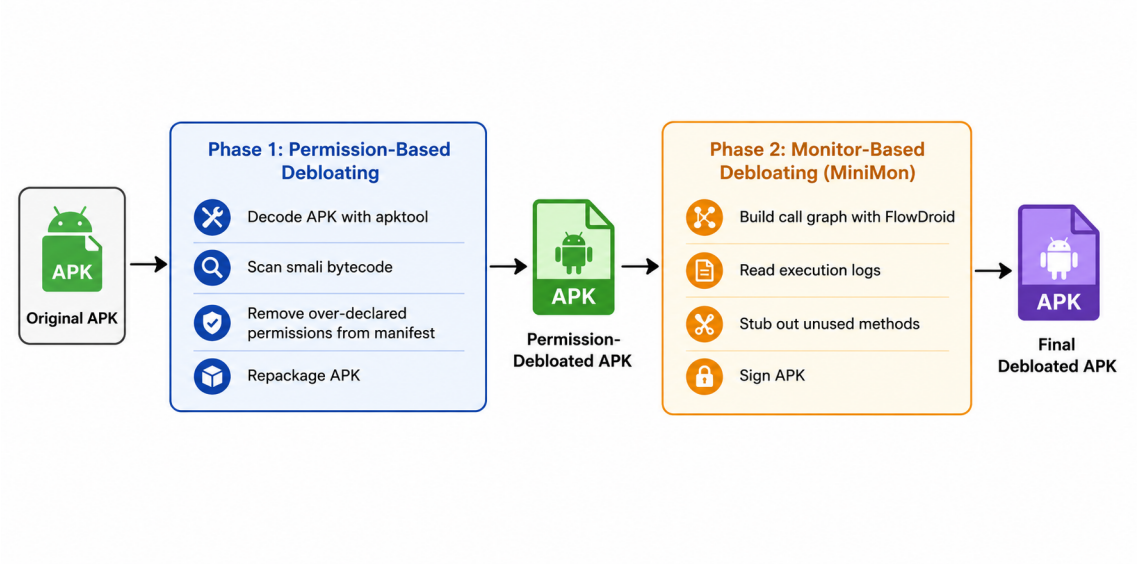

Figure 4.1: Overview of the two-phase debloating pipeline, from the original APK through Phase 1 (permission-based debloating) and Phase 2 (MiniMon's monitor-based debloating) to the final debloated APK.

The rationale for this ordering is that permission-based debloating is conservative, fast, and requires no device interaction. It can be applied automatically to any APK without usage data. Monitor-based debloating is more powerful but requires a usage session and is sensitive to the quality of the call graph built from the input APK. Running permission debloating first reduces the manifest footprint of the app before the more expensive dynamic phase begins.

Concretely, the pipeline takes an original APK as input. Phase 1 decodes the APK using apktool [12], a reverse engineering tool that unpacks a compiled APK into its manifest, resources, and smali source files, scans the smali bytecode to detect over-declared permissions, removes them from the manifest, and repackages the result into a permission-debloated APK. This permission-debloated APK is then passed as input to Phase 2, which constructs a call graph using FlowDroid [5], reads the pre-collected execution logs, and produces a final debloated APK with unused methods stubbed out. If Phase 1 finds no over-declared permissions, the original APK is passed directly to Phase 2 unchanged.

An alternative ordering would be to run MiniMon first and then apply Phase 1 to the debloated output. However, this produces no benefit over the current ordering. MiniMon stubs out unused methods rather than deleting them, meaning their API signatures remain present in the bytecode of the debloated APK. Phase 1's scanner would therefore still detect those signatures and retain the associated permissions, producing the same manifest result as running Phase 1 on the original APK directly. The current ordering is preferred because a cleaner manifest as input to Phase 2 ensures that FlowDroid constructs its call graph from an APK that already reflects the application's actual permission footprint.

4.3 Phase 1: Permission-Based Debloating

The first phase identifies permissions declared in the `AndroidManifest.xml` whose corresponding Android API methods are never invoked in the application's bytecode, and removes them from the manifest. No code is modified in this phase.

4.3.1 Permission Classification

The phase begins by parsing all `<uses-permission>` entries from the decoded manifest. Each declared permission is then looked up in a manually constructed mapping file, `permissions_map.json`, which maps Android permission strings to the set of Smali-level API method signatures that require that permission. This mapping was constructed from multiple sources, including the Android developer documentation, the PScout and Axplorer academic datasets [8], [13], [14], and covers 47 permissions in total.

Each declared permission is classified into one of three categories. A permission is classified as *used* if at least one of its mapped API signatures is found in the application's smali bytecode. It is classified as *over-declared* if it has a mapping in `permissions_map.json` but none of its API signatures appear in the bytecode. It is classified as *not in map* if no mapping exists for it. Permissions in the third category are conservatively kept, since their usage cannot be determined from the available mapping. Figure 4.2 summarises this classification logic.

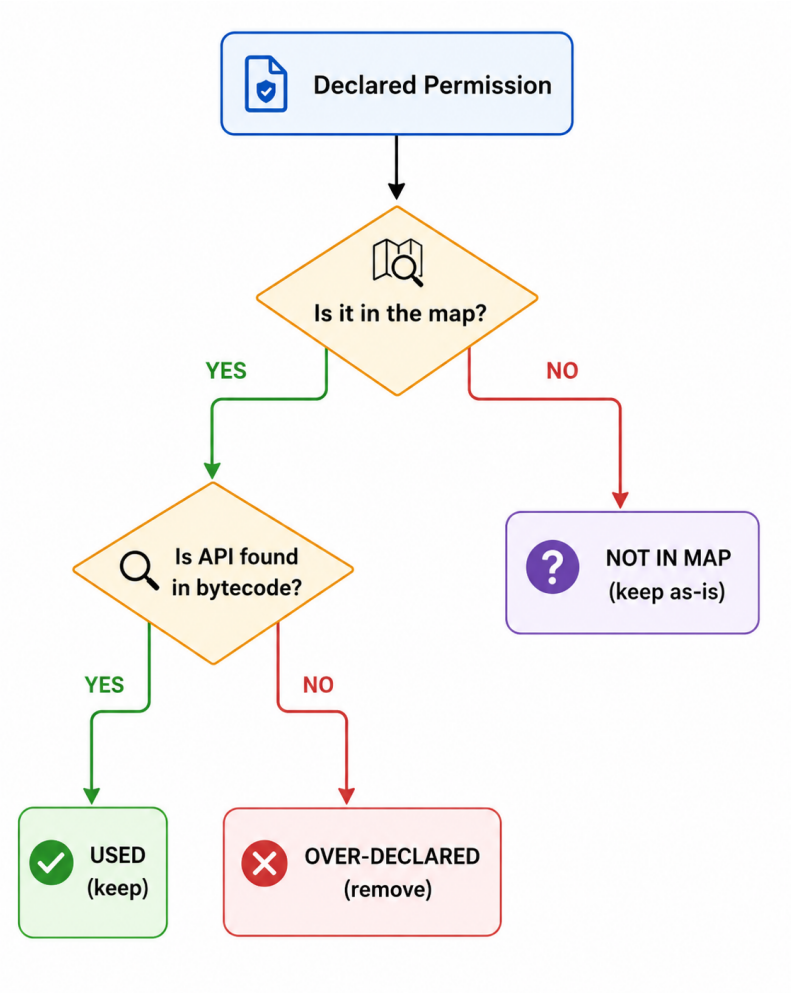

Figure 4.2: Decision logic used to classify each declared permission as used, over-declared, or not in map.

Certain permissions are intentionally excluded from the map and are always retained regardless of whether their API is found in the bytecode. These fall into two categories. Broadcast-only permissions such as `RECEIVE_BOOT_COMPLETED` and `PROCESS_OUTGOING_CALLS` are enforced by the Android OS at broadcast delivery time and have no associated API call to scan for, making bytecode scanning an inappropriate detection mechanism for them. Third-party permissions such as `com.google.android.c2dm.permission.RECEIVE` and `com.android.vending.BILLING` protect inter-application communication rather than Android framework APIs, and their usage cannot be determined by scanning for Android SDK method signatures.

4.3.2 Smali Scanning

The APK is first decoded using apktool, which produces the smali source files for all dex classes, including multi-dex apps where classes are split across `smali/`, `smali_classes2/`, and further directories. All `.smali` files are scanned recursively. For each declared permission with a known mapping, the scanner checks whether

any of its associated API signatures appear as a substring in any smali file. If a match is found, the permission is marked as used.

This approach does not require call graph construction and runs in linear time relative to the size of the smali output. It is conservative in one direction: a permission API invoked via Java reflection will not be detected by the scanner, meaning a genuinely used permission may be incorrectly classified as over-declared. This is acknowledged as a threat to validity and discussed in Chapter 7.

4.3.3 Manifest Modification and Repackaging

Over-declared permissions are removed from the manifest by parsing `AndroidManifest.xml` with a DOM (Document Object Model) parser, removing the corresponding `<uses-permission>` nodes, and writing the modified manifest back to disk. The decoded directory is then repackaged into a new APK using "`apktool b`".

One additional step is required for compatibility with Phase 2. When apktool repackages an APK, it regenerates the `resources.arsc` file, which stores compiled resource tables. FlowDroid's ARSC parser, used by MiniMon during call graph construction, does not accept the regenerated format and throws a parse error. To address this, the original `resources.arsc` is extracted from the original APK using ZIP I/O and copied byte-for-byte into the repackaged APK. This is safe because Phase 1 modifies only `AndroidManifest.xml` and does not touch any resource definitions. The output of Phase 1 is a repackaged APK with the same bytecode as the original but a reduced manifest, stored in `out/PermDebloatedAPKs/<appName>/`.

4.4 Phase 2: Monitor-Based Debloating

The second phase applies the debloating component of MiniMon [7] to the permission-debloated APK produced by Phase 1. MiniMon is a monitoring-based debloating framework whose full pipeline includes an instrumentation phase, a device usage phase, and a debloating phase. In this thesis, the instrumentation and usage phases are not repeated. Instead, the execution logs provided as part of the MiniMon research artifact are used directly, as these logs were collected by the original authors during controlled usage sessions on the same set of applications. The MiniMon tool and its associated replication package were provided directly by the authors for use in this thesis. Figure 4.3 illustrates MiniMon's full eight-step process, where steps 1 and 2 are not repeated in this thesis, and the pipeline below begins from the pre-collected execution logs at step 3.

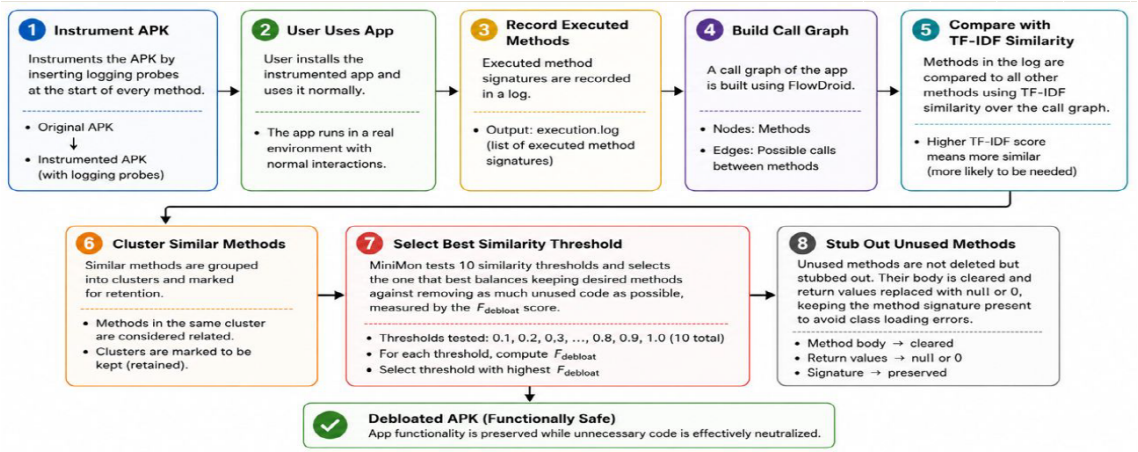

Figure 4.3: The eight-step MiniMon debloating process. Steps 1–2 (instrumentation and usage) were performed by the original MiniMon authors; steps 3–8 are applied in this thesis using the pre-collected execution logs.

The debloating phase takes two inputs: the execution logs, which record the method signatures executed during the monitored usage sessions, and the call graph of the input APK, constructed using FlowDroid. Given these inputs, MiniMon builds method similarity matrices using TF-IDF encoding over static execution paths derived from the call graph, then groups methods into clusters based on behavioural similarity. It tests similarity thresholds from 0.1 to 1.0, and for each threshold identifies the set of methods considered similar enough to the executed methods to be retained. The threshold that maximises the F_debloat score is selected, where F_debloat is a weighted harmonic mean of recall and debloating rate that prioritises recall to reduce the risk of removing needed functionality. Methods outside the retained set are stubbed out using Soot's [6] `MethodTransformer`, and the resulting APK is signed and written to `out/DebloatedTest/<packageName>/`.

It is worth noting that the execution logs were collected from the original, undebloated versions of the apps. Applying these logs to the permission-debloated APK is valid because Phase 1 modifies only the manifest and does not alter the bytecode. The set of executable methods in the Phase 1 output is identical to that of the original APK, meaning the logs remain applicable without modification. This is a direct consequence of the manifest-only design of Phase 1, since no bytecode is changed, MiniMon's analysis of the permission-debloated APK produces the same method-level call graph and the same debloating decisions as it would on the original.

4.5 Dataset

The pipeline is evaluated on a dataset of 38 real-world Android applications sourced from the MiniMon research artifact [7], stored in `src/apk/`. The dataset was originally collected by Liu et al. for the MiniMon study, with apps selected from the top applications listed on AndroidRank across multiple categories. The apps span a range of categories and permission profiles, with an average of 7.4 permissions

declared per application. A batch scan of the full dataset revealed that `VIBRATE` and `ACCESS_WIFI_STATE` are among the most commonly declared permissions, appearing in multiple applications, and that over-declared permissions were detected in 44.7% of the dataset. Of the 38 applications, 28 had pre-collected execution logs available in the MiniMon replication package, making them eligible for Phase 2 evaluation. The remaining 10 applications were evaluated for Phase 1 only.

5

Implementation

The following chapter describes the implementation of the proposed two-phase Android application debloating pipeline. Section 5.1 outlines the development environment and build configuration used throughout the project. Section 5.2 presents the overall tool architecture, describing how the two phases are structured and connected. Section 5.3 details the Phase 1 implementation, covering each step of the permission-based debloating pipeline from manifest parsing through smali scanning, permission classification, manifest modification, and repackaging with resource restoration. Section 5.4 describes the Phase 2 implementation, which integrates the debloating component of MiniMon [7] into the pipeline using the permission-debloated APK produced by Phase 1 as input. Section 5.5 presents the batch evaluation infrastructure developed to execute the pipeline across the full application dataset. Section 5.6 lists all external tools and libraries used throughout the implementation.

5.1 Development Environment

The tool is implemented in Java and built using Apache Maven 3.9.8. Development was carried out in IntelliJ IDEA on Windows, using Java 8 (JDK 1.8.0_461). Java 8 is a hard requirement imposed by Soot [6] and FlowDroid [5], both of which are incompatible with newer JVM (Java Virtual Machine) versions due to internal API changes introduced in Java 9 and later. Attempting to run either framework under a newer JVM results in runtime errors that prevent call graph construction and bytecode transformation from functioning correctly.

The project is structured as a single Maven module containing the permission-based debloating component alongside the adapted MiniMon codebase. All dependencies are declared and resolved through `pom.xml`, and include Soot, which is used in Phase 2 for bytecode analysis and method stubbing via `MethodTransformer`; FlowDroid, which is used in Phase 2 to construct the application call graph inside `CGBuilder.java`; apktool [12], which is invoked via `ProcessBuilder` in both `Main.java` and `BatchEval.java` for APK decoding and repackaging; jose4j [15], whose bundled json-simple parser is used in `PermissionBasedDebloating.java` to load and parse `permissions_map.json` at runtime; and the MiniMon artifact [7], obtained from the replication package published by Liu et al., which provides the monitor-based debloating framework used in Phase 2.

5.2 Tool Architecture

The entry point of the tool is `Main.java`, which orchestrates the full two-phase debloating pipeline. The class is designed to process a single APK at a time, with the APK path specified directly in the source code. This design choice was made to allow fine-grained control over individual runs during development and testing, while batch processing across the full dataset is delegated to `BatchEval.java`, described in Section 5.5.

Given an input APK path, `Main.java` begins by decoding the APK using apktool, invoked via a `ProcessBuilder` call with the `-f` flag. The force re-decode flag is essential to ensure that the decoded directory always reflects the contents of the original APK, preventing stale manifest changes from a previous run from persisting into the current one. The decoded output is written to a directory derived from the APK path by replacing the `.apk` extension with `-decoded`.

Once decoding completes successfully, `Main.java` invokes `PermissionBasedDebloating.main()`, passing both the decoded directory path and the original APK path. This method executes the full Phase 1 pipeline and returns the absolute path to the permission-debloated APK. If no over-declared permissions are found during Phase 1, the original APK path is returned unchanged, ensuring Phase 2 always receives a valid input regardless of whether any manifest modifications were made.

The returned path is then passed directly to `MonitorDebloat.main()`, which executes Phase 2 using the permission-debloated APK as its input. Phase 2 operates independently of Phase 1 in terms of its internal analysis, it constructs its own call graph from the input APK using FlowDroid and reads the pre-collected execution logs associated with the application's package name. The two phases are therefore loosely coupled, connected only through the APK path returned by Phase 1, which makes the architecture straightforward to extend or modify independently on either side.

5.3 Phase 1 Implementation

Phase 1 is implemented in `PermissionBasedDebloating.java`. The class exposes a static `main()` method that accepts the decoded APK directory path and the original APK path, executes the full seven-step debloating pipeline discussed in Sections 5.3.1–5.3.6, and returns the path to the debloated APK. The method also accepts a boolean `evaluate` flag which, when set to `true`, it activates the CSV output path and triggers a call to `appendCsvRow()` at the end of processing to record the results for that APK, but this flag is only set to true in `BatchEval.java` file.

5.3.1 Manifest Parsing

The `AndroidManifest.xml` file in the decoded directory is parsed using Java's built-in DOM parser (`javax.xml.parsers.DocumentBuilder`). The `DocumentBuilderFactory`

is configured with `setNamespaceAware(true)` to correctly resolve namespace-qualified attributes, which is necessary because the permission name is stored under the Android XML namespace as `android:name` rather than as a plain attribute. Without namespace awareness, retrieving the attribute by its qualified name returns an empty string for all elements. The correct value is instead retrieved using `el.getAttributeNS()` with the Android XML namespace URI and the local attribute name "`name`" as parameters. All `<uses-permission>` elements are retrieved by tag name from the parsed document and iterated to collect their name values into a `List<String>` of declared permissions, which is passed to subsequent steps.

5.3.2 Permission-to-API Mapping

The mapping between Android permissions and their corresponding Smali-level API signatures is stored in `permissions_map.json`, located at the project root. The file is loaded at runtime using the json-simple parser bundled within the jose4j library, parsed into a `JSONObject`, and converted into a `Map<String, List<String>>` where each key is a permission string and each value is a list of Smali method signatures that require that permission. A `Map` is used to preserve insertion order during iteration, which ensures consistent and reproducible scanning behaviour across runs.

The mapping covers 47 Android permissions and was constructed by consulting the Android developer documentation [13], the PScout [8] and Axplorer [14] academic permission-to-API mapping datasets. Each entry in the map contains one or more Smali method descriptors in the format `Lpackage/Class;->method(params)returnType`, which matches the exact string representation used in compiled Smali bytecode and allows direct substring matching without any additional parsing.

Permissions intentionally excluded from the map include broadcast-only permissions such as `RECEIVE_BOOT_COMPLETED` and `PROCESS_OUTGOING_CALLS`, which are enforced by the Android OS at broadcast delivery time and have no associated API call to scan for, and third-party permissions such as `com.google.android.c2dm.permission.RECEIVE` and `com.android.vending.BILLING`, which protect inter-application communication rather than Android framework APIs. Including such permissions in the map would not be meaningful since they cannot be detected through bytecode scanning. Permissions absent from the map are therefore conservatively retained without modification, avoiding false positives.

5.3.3 Smali Scanning

The decoded APK directory is walked recursively using `java.nio.file.Files.walk()`, which returns a `Stream<Path>` of all files and directories reachable from the root. The stream is filtered to retain only paths whose filename ends with the `.smali` extension, producing a flat list of all smali source files in the decoded output. This approach correctly handles multi-dex applications, where the Android build system splits bytecode across multiple dex containers compiled into separate directories such as `smali/`, `smali_classes2/`, and `smali_classes3/` by apktool. All directories are

traversed in a single pass without any special handling.

For each declared permission that has a corresponding entry in the permissions map, the scanner iterates over all smali files and reads each one into a `String` using `new String(Files.readAllBytes(path))`. It then checks whether any of the mapped API signatures appear as a substring within that string using `String.contains()`. This approach is intentionally simple and fast, since Smali method invocation instructions always contain the full method descriptor as a literal string in the bytecode, substring matching is sufficient to detect any direct invocation of a permission-protected API. If a match is found for any signature associated with a permission, the permission is immediately added to the set of used permissions and the inner loop breaks early, avoiding unnecessary scanning of remaining files for that permission.

5.3.4 Permission Classification

Once scanning is complete, each declared permission is classified into one of three mutually exclusive categories based on the results. A permission is classified as *used* if it has an entry in `permissions_map.json` and at least one of its mapped API signatures was found in the smali bytecode during scanning. It is classified as *over-declared* if it has an entry in the map but no corresponding API call was detected in any smali file, indicating that the permission is declared in the manifest but never exercised in code. It is classified as *not in map* if no entry exists for it in `permissions_map.json`, in which case no determination can be made and it is conservatively retained. The used count reported in the CSV is derived as the total declared count minus the over-declared count minus the not-in-map count. Only permissions classified as over-declared are passed to the manifest modification step.

5.3.5 Manifest Modification

If the list of over-declared permissions is empty, repackaging is skipped entirely and the original APK path is returned to the caller unchanged, avoiding unnecessary processing overhead for applications that have no removable permissions. Otherwise, the decoded directory is first copied in its entirety to a dedicated working directory under `out/PermDebloatedAPKs/<appName>/`. This is achieved by walking the source directory recursively and copying each file and subdirectory to the corresponding target path, creating intermediate directories as needed and overwriting any existing files at the destination. This copy step is critical as it preserves the original decoded directory intact, ensuring that the original APK can always be re-decoded cleanly on subsequent runs without any previously applied manifest modifications carrying forward.

Permissions are removed from the copied manifest by retrieving all `<uses-permission>` elements from the parsed DOM as a node list, iterating over them, and removing each node whose `android:name` attribute matches a permission in the over-declared list by calling the removal method on its parent node. The modified DOM is then serialized back to disk using a transformer obtained from the transformer factory,

configured with UTF-8 character encoding and indented output, and written to a file output stream targeting the manifest file path within the working directory.

5.3.6 Repackaging and Resource Restoration

The modified working directory is repackaged into a new APK using apktool, invoked via a process builder with the build subcommand, the working directory as input, and the output path set to `out/PermDebloatedAPKs/<appName>-permission-debloated.apk`. Standard input and output are inherited from the parent process so that apktool's progress output remains visible during execution. If apktool exits with a non-zero exit code, an exception is thrown immediately to prevent a corrupted or incomplete APK from being passed to Phase 2.

Following repackaging, the `resources.arsc` file inside the new APK must be replaced with the original one from the input APK. This step is necessary because apktool regenerates `resources.arsc` during repackaging using its own resource compiler, producing a binary format that FlowDroid's ARSC parser does not accept, which causes Phase 2 to fail with a parse error during call graph construction. The restoration is implemented entirely in memory without extracting any files to disk. The original APK is opened as a ZIP input stream to locate and read the `resources.arsc` entry into a byte array, and the repackaged APK is then streamed entry by entry into a temporary output file, substituting the `resources.arsc` entry with the byte array read from the original. Since Phase 1 modifies only `AndroidManifest.xml` and does not alter any resource definitions, values, or identifiers, replacing `resources.arsc` byte-for-byte from the original is safe and produces a fully functional APK.

5.4 Phase 2 Implementation

Phase 2 integrates the debloating component of MiniMon [7] into the pipeline, operating on the permission-debloated APK produced by Phase 1. The MiniMon tool and its associated replication package, including the pre-collected execution logs, were provided directly by the authors of the original MiniMon paper for use in this thesis. Unlike Phase 1, which is a purely static analysis step requiring no device interaction, Phase 2 is a dynamic debloating approach that relies on runtime execution data to identify which methods of the application were actually exercised during usage. The entry point for this phase is `MonitorDebloat.main()`, which is called from `Main.java` with the path to the permission-debloated APK and a debug mode flag that controls whether the tool processes a single application or a directory of applications. Figure 4.3 in Section 4.4 gives an overview of this eight-step process; the remainder of this section describes its concrete implementation, class by class.

The first step inside `MonitorDebloat` is to verify that execution logs exist for the target application. The package name is extracted from the APK using `Tool.getPackageName()`, and the log directory is resolved via `Config.getExecutedLogDir(packageN`. If no log directory exists for the given package name, the method returns early without producing a debloated APK. The execution logs used in this thesis were obtained from

the replication package published alongside the original MiniMon paper [7], available at the associated provided repository. These logs record method-level execution traces collected during controlled usage sessions on the applications in the dataset. Since Phase 1 modifies only the manifest and does not alter the dex bytecode, the logs collected from the original APKs remain fully valid for the permission-debloated versions.

Assuming logs are present, an `ExecutionCollector` is instantiated with the package name and log directory path. This component reads the execution logs and produces two sets: the set of executed method signatures (`executedMethods`), and a test case set (`testCase`) used for evaluating specification quality. If either set is empty, the method returns early, as there is insufficient runtime data to guide debloating decisions. The call graph of the input APK is then constructed by instantiating a `CGBuilder` with the permission-debloated APK path and calling `getProgramGraph()`. This produces a `ProgramGraph` object representing the method-level call graph of the application, which is used in the subsequent clustering step to reason about method similarity and reachability.

Given the executed methods and the call graph, MiniMon applies a clustering-based specification, implemented in the `KeepClusteringSpecification` class, to determine which methods should be retained in the debloated APK. The specification works by first computing a similarity embedding for each method in the call graph using TF-IDF encoding over static execution paths, then grouping methods into clusters based on their behavioral similarity through hierarchical clustering. Methods that fall into the same cluster as an executed method are considered related to user-desired functionality and are marked for retention.

To find the most effective debloating configuration, this process is repeated ten times, once for each similarity threshold value between 0.1 and 1.0 in increments of 0.1. A lower threshold retains more methods by considering a broader set of methods as similar to the executed ones, while a higher threshold is more selective and results in a more aggressively debloated APK. For each threshold, the resulting set of methods to remove is evaluated against the test case set using the F_debloat score, which is a weighted harmonic mean of recall and debloating rate that prioritises recall to reduce the risk of removing methods that are needed for correct application behaviour. The threshold that produces the highest F_debloat score is selected as the best specification, and its corresponding set of methods to remove is passed to the debloating step.

The debloating step is performed using Soot's instrumentation framework. Soot is first initialised with the output directory and input APK path, after which a `MethodTransformer` is instantiated with the package name and the set of methods identified for removal. Rather than deleting methods entirely from the bytecode, each targeted method is stubbed out by clearing its body and replacing its return value with `null` for reference types or `0` for numeric types. This approach ensures that the method signatures remain present in the bytecode, which is necessary to avoid class loading errors at runtime while effectively neutralising the method's

functionality. The number of successfully stubbed methods is logged for reporting purposes. Finally, the debloated APK is signed and written to the output directory under `out/DebloatedTest/<packageName>/`, completing the two-phase pipeline.

5.5 Evaluation Infrastructure

Batch evaluation across the full dataset is handled by two dedicated classes, `BatchEval.java` and `BatchRQ3.java`, each targeting a different subset of the research questions.

`BatchEval.java` addresses RQ1 and RQ2 by running Phase 1 across all APKs in the dataset. It scans the `src/apk/` directory for all files ending in `.apk`, excluding any files already suffixed with `-permission-debloated.apk` to avoid processing pipeline outputs. For each APK, it invokes apktool to decode it, then calls `PermissionBasedDebloating.main()` with the `evaluate` flag set to `true`, which triggers the CSV output path in `appendCsvRow()`. Results are appended to `evaluation_results.csv`, with a header row written automatically on first creation. Each row records the APK filename, package name, total declared permissions, used count, over-declared count, not-in-map count, the list of removed permissions as a semicolon-separated string, and the original and debloated APK sizes in kilobytes. If an APK fails at either the decoding or debloating step, the error is printed and execution continues with the next APK.

`BatchRQ3.java` addresses RQ3 by running both debloating approaches on each app in sequence within a single pass, enabling a direct side-by-side comparison. For each APK it first runs `MonitorDebloat.main()` on the original APK and records the resulting debloated APK size from `out/DebloatedTest/<packageName>/` before any further processing takes place. It then runs `PermissionBasedDebloating.main()` to produce the permission-debloated APK, followed by a second call to `MonitorDebloat.main()` on that debloated APK, recording the new output size from the same directory. Each row in `rq3_results.csv` contains the APK filename, package name, original APK size, size after MiniMon alone, and size after the full Phase 1 plus MiniMon pipeline. If MiniMon has no execution logs for a given package name or fails at any step, the error is logged and the app is skipped. To address a FlowDroid infinite loop encountered during batch execution, call graph construction was bounded by capping callback depth at 5, limiting callbacks per component to 100, and applying a 60-second timeout, after which FlowDroid aborts and the app is skipped.

5.6 Tools and Dependencies

This section summarises the external tools and libraries used throughout the implementation, along with the specific role each one plays in the pipeline.

Soot [6] is a Java bytecode analysis and transformation framework used exclusively in Phase 2. It is responsible for initialising the analysis environment, constructing the application's call graph in combination with FlowDroid [5], and performing the

method stubbing step through the `MethodTransformer` class, which clears method bodies and replaces return values to neutralise unused functionality.

FlowDroid [5] is an Android-specific static analysis framework built on top of Soot. It is used in Phase 2 inside `CGBuilder.java` to construct a precise call graph of the input APK. FlowDroid models Android-specific behaviour including activity lifecycle methods and UI callbacks as entry points, which is necessary to produce a call graph that accurately reflects how an Android application executes.

apktool [12] is a reverse engineering tool for Android APKs. It is invoked via `ProcessBuilder` in both `Main.java` and `BatchEval.java` to decode APKs into their smali bytecode and resource representations, and again during Phase 1 to repackage the modified decoded directory back into a valid APK after manifest modification.

jose4j [15] is a Java library primarily designed for JSON Web Token handling. Its bundled json-simple parser is used in `PermissionBasedDebloating.java` to load and parse `permissions_map.json` at runtime, converting the JSON structure into the permission-to-API mapping used during smali scanning.

MiniMon [7] is the monitor-based debloating framework developed by Liu et al. and used without modification in Phase 2. It provides the `MonitorDebloat`, `CGBuilder`, `KeepClusteringSpecification`, `MethodTransformer`, and `ApkUtils` components that collectively handle call graph construction, method similarity analysis, bytecode transformation, and APK signing in the second phase of the pipeline.

zipalign and apksigner [16] are Android SDK build tools used internally by MiniMon's `ApkUtils.sign()` method. zipalign ensures that the entries in the final APK are aligned on four-byte boundaries as required by the Android package manager, and apksigner applies a valid digital signature to the APK so that it can be installed on a device or emulator.

6

Evaluation

This chapter presents the evaluation of the proposed two-phase debloating pipeline against the research questions defined in Chapter 4. The evaluation is conducted on a dataset of 38 real-world Android applications sourced from the MiniMon research artifact [7]. Section 6.1 addresses RQ1 by analysing the prevalence and distribution of over-declared permissions across the dataset. Section 6.2 addresses RQ2 by measuring the impact of Phase 1 on APK size and examining whether manifest-level permission removal can be applied without breaking application functionality. Section 6.3 addresses RQ3 by examining the combined effect of both phases and discussing whether permission-based debloating serves as an effective preprocessing step for monitor-based debloating.

6.1 RQ1: Prevalence of Over-Declared Permissions

To what extent do real-world Android applications declare permissions that are never invoked in their bytecode?

Across the 38 applications in the dataset, a total of 282 permissions were declared, averaging 7.4 permissions per application, as summarised in Table 6.1. Of these, 158 were classified as used, 33 as over-declared, and 91 as not in map. The not-in-map category covers permissions for which no API mapping exists in `permissions_map.json`, such as broadcast-only and third-party permissions, which are conservatively retained without analysis. Figure 6.1 visualises this breakdown.

Table 6.1: Summary of permission classifications across the 38 applications in the dataset.

Classification	**Count**
Total permissions declared	282
Used	158
Over-declared	33
Not in map	91
Applications with at least one over-declared permission	17 (44.7%)
Applications with no over-declared permissions	21 (55.3%)
Average permissions declared per application	7.4
Average over-declared per affected application	1.9

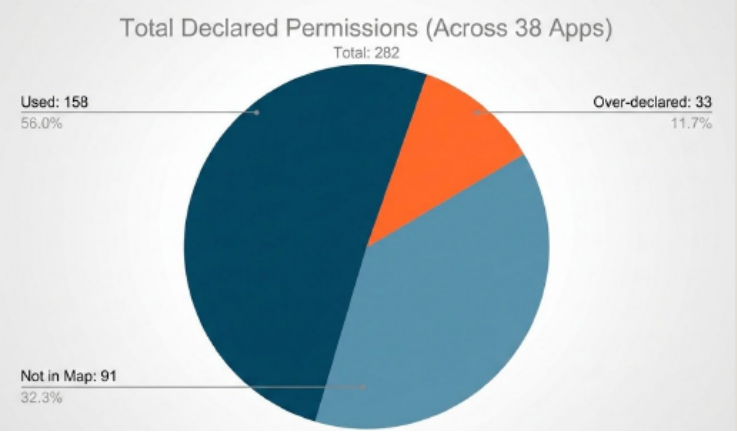

Figure 6.1: Proportional breakdown of the 282 declared permissions across the dataset: used, over-declared, and not in map.

Over-declared permissions were found in 17 out of 38 applications, representing 44.7% of the dataset. This indicates that nearly half of the applications in the dataset declare at least one permission whose corresponding API is never invoked in the bytecode, confirming that over-declaration is a widespread phenomenon rather than an isolated occurrence. Among the 17 affected applications, an average of 1.9 over-declared permissions were found per app, with the highest count being 5 permissions in `kr.cmonster.chat`. Figure 6.2 illustrates this split across the dataset.

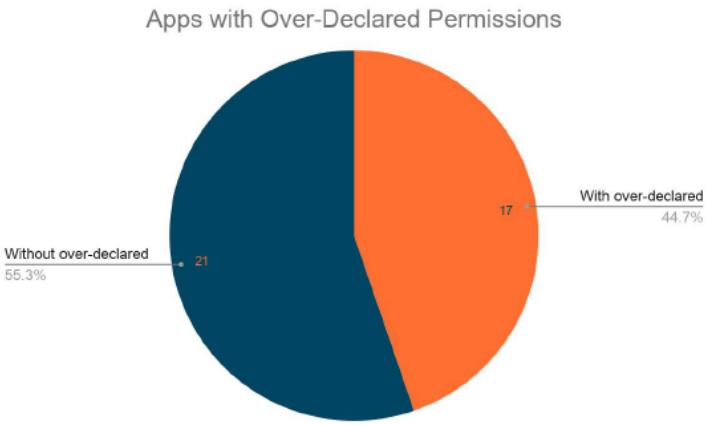

Figure 6.2: Proportion of applications in the dataset with at least one over-declared permission.

Figure 6.3 shows the full list of over-declared permissions found across the dataset. The most frequently over-declared permissions across the dataset were `VIBRATE`, found in 7 applications, and `ACCESS_WIFI_STATE`, found in 6 applications. These are both normal-level permissions that are commonly included in third-party SDKs or template manifests and may persist in the manifest even when the associated functionality is never used. Other permissions appearing in multiple applications include `READ_PHONE_STATE` and `CALL_PHONE`, both in 3 and 2 applications respectively, which are dangerous-level permissions that carry more significant privacy implications when over-declared.

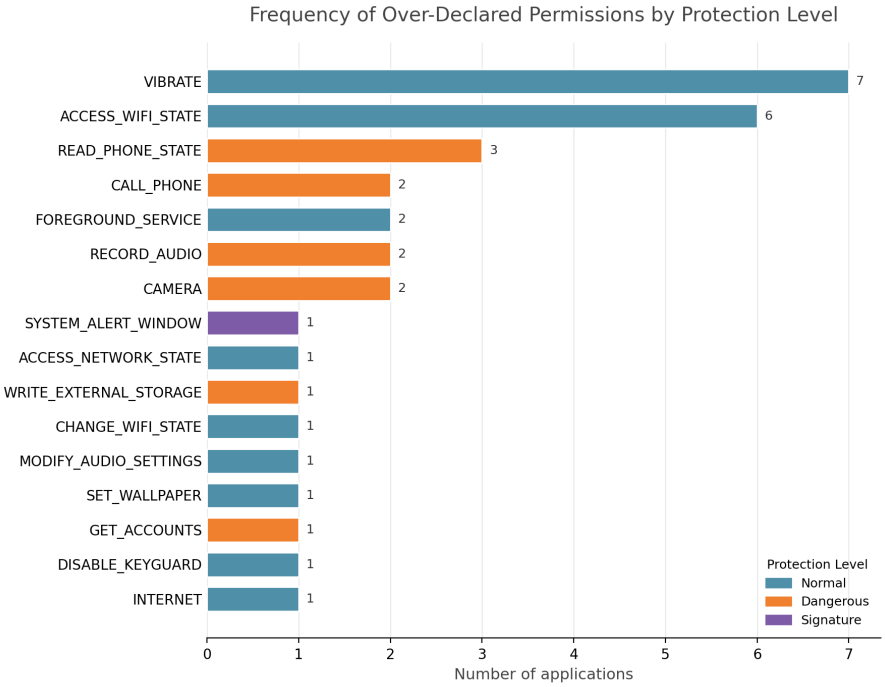

Figure 6.3: Frequency of over-declared permissions across the dataset, sorted and color-coded by protection level.

These results provide a clear answer to RQ1: over-declaration of Android permissions is common in real-world applications, affecting nearly half the dataset, and the permissions most frequently over-declared include both privacy-sensitive dangerous permissions and commonly bundled normal permissions.

6.2 RQ2: Effectiveness and Safety of Phase 1

Can manifest-level permission debloating reduce APK size and attack surface without breaking application functionality?

Phase 1 successfully processed all 38 applications without any crashes or failures during repackaging. Among the 17 applications where over-declared permissions were found and removed, 16 resulted in a smaller APK after repackaging, as shown in Table 6.2 and Figure 6.4. The average size reduction across those 16 applications was 122.14 KB, with a maximum reduction of 234.43 KB achieved for `bigprofilephoto.bigprofilepicture`. The minimum reduction observed among apps where the size changed was 57.49 KB. The remaining 21 applications, where no over-declared permissions were found, were returned unchanged, with their original APK path passed directly to Phase 2.

Table 6.2: APK size before and after Phase 1 debloating for applications where over-declared permissions were found.

Package Name	Original (KB)	After Phase 1 (KB)	Reduction (KB)	Reduction (%)
com.naouimi.henna	4180.12	4013.60	166.52	4.0
com.androidlab.videoroad	1671.42	1569.93	101.49	6.1
ru.waptaxi.driver	2456.39	2358.97	97.42	4.0
com.callrecorder.acr	4944.56	4719.11	225.45	4.6
zepeto.walkthrough.zepetotricks	2509.72	2409.08	100.64	4.0
com.darkhorse.digital	3591.72	3414.75	176.97	4.9
com.antiporn.pornoblock.safebrowser	2664.64	2606.85	57.79	2.2
com.nextpick.makeupcontouring	3707.26	3566.29	140.97	3.8
org.androworks.meteorgram	1771.95	1677.07	94.88	5.4
com.masrio.losefats2018	3929.49	3793.76	135.73	3.5
org.androworks.meteor	1884.15	1788.06	96.09	5.1
kr.cmonster.chat	3166.80	3179.28	−12.48	−0.4
bigprofilephoto.bigprofilepicture	5377.17	5142.74	234.43	4.4
com.x15tech.tarifadinamica	3816.69	3651.01	165.68	4.3
egw.estate	3417.85	3247.53	170.32	5.0
com.amaze.filemanager	7796.42	7738.93	57.49	0.7
com.tomer.alwayson	5062.66	4995.74	66.92	1.3
Average	**3717.59**	**3627.82**	**122.14**	**3.8**

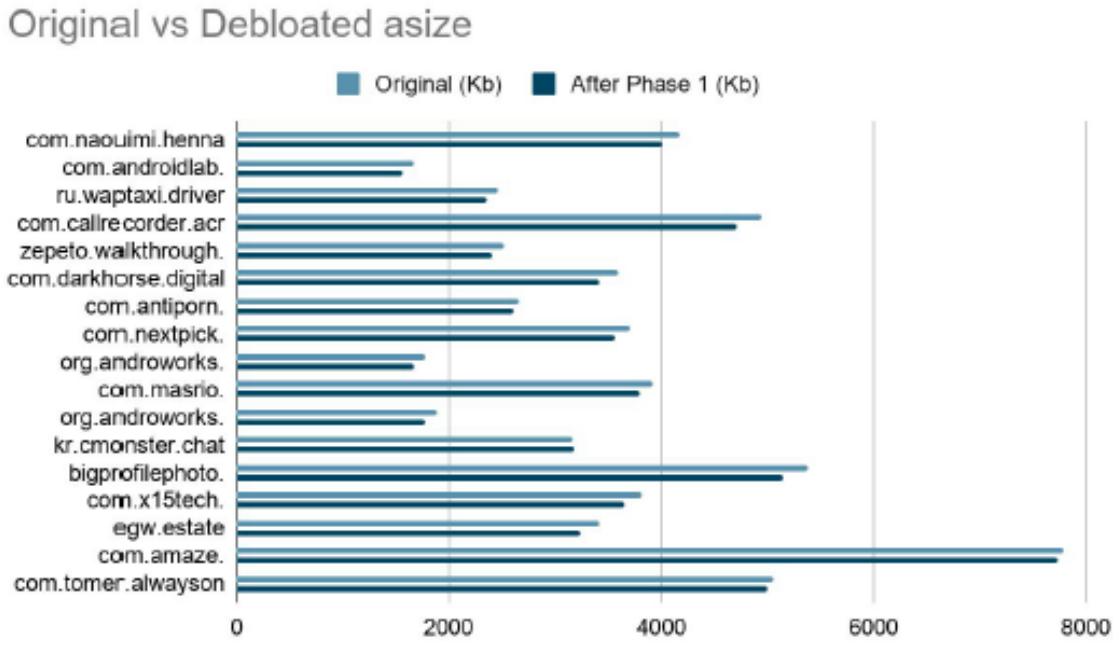

Figure 6.4: Original versus Phase 1-debloated APK size for the 17 applications with at least one over-declared permission.

One application, `kr.cmonster.chat`, showed a marginal size increase of 12.48 KB after repackaging despite having 5 permissions removed. This is attributable to minor variations in how apktool compresses resources during repackaging rather than to any issue with the permission removal itself. The manifest modification was applied correctly and the 5 over-declared permissions were successfully removed from the manifest. This case highlights that APK size is an imperfect metric for evaluating manifest-level changes, since the size of the manifest is small relative to the overall APK and repackaging overhead can occasionally outweigh the savings.

In terms of attack surface reduction, the removal of 33 over-declared permissions across the dataset represents a meaningful reduction in the set of privileges these applications unnecessarily claim. Permissions such as `CAMERA`, `RECORD_AUDIO`, and `READ_PHONE_STATE` are particularly significant in this regard, as their presence in the manifest signals to the Android system and to users that the application may access sensitive device resources, even when it never does so in practice.

Regarding functional correctness, Phase 1 is designed to be safe by construction. Since only over-declared permissions are removed, those whose mapped API signatures are absent from the smali bytecode, no executable code path is affected by the manifest modification. The debloated APKs are structurally identical to the originals in terms of bytecode, differing only in their manifest declarations. This design ensures that the debloated application behaves identically to the original for all functionality that was present and reachable before debloating.

6.3 RQ3: Combined Effect of Phase 1 and Phase 2

Does applying permission-based debloating as a preprocessing step improve the results of monitor-based debloating compared to applying MiniMon on the original APK directly?

To answer RQ3, `BatchRQ3.java` was used to run both debloating strategies on each of the 28 applications for which MiniMon execution logs were available. For each application, MiniMon was first run on the original APK to establish a baseline, and then run again on the permission-debloated APK produced by Phase 1. The remaining 10 applications were skipped because no execution logs were available in the MiniMon replication package for those package names. Additionally, FlowDroid [5] call graph construction was bounded during the batch run by capping callback depth at 5, limiting callbacks per component to 100, and applying a 60-second timeout, to prevent infinite loops encountered in some applications during analysis. Table 6.3 summarises these outcomes at a glance.

Table 6.3: Summary of RQ3 outcomes across the 28 applications with available execution logs.

Outcome	**MiniMon Only**	**Phase 1 + MiniMon**	**Extra from Phase 1**
Reduced	18/28	21/28	11/28
Increased	6/28	5/28	1/28
Unchanged	4/28	2/28	

Table 6.4 summarises the results for all 28 applications. MiniMon alone reduced APK size in 18 out of 28 applications, left 4 unchanged, and increased size in 6. The average size change across all 28 applications was a reduction of 72.03 KB, though this average is influenced by several cases where repackaging overhead caused the output APK to be larger than the input. The combined pipeline reduced size in 21 applications, left 2 unchanged, and increased size in 5, representing an improvement of 3 additional applications reduced compared to MiniMon alone.

Phase 1 produced additional size reduction on top of MiniMon in 11 out of 28 applications, with extra reductions ranging from 0.18 KB to 36 KB. The most notable cases were `org.androworks.meteorgram` with 36 KB of additional reduction, `egw.estate` with 28 KB, `com.androidlab.videoroad` with 16 KB, `com.masrio.losefats2018` with 16 KB, and `com.tomer.alwayson` with 16 KB. In 16 applications the combined pipeline produced the same result as MiniMon alone, indicating that Phase 1 found no over-declared permissions in those apps or that the size difference was negligible. In one application, `kr.cmonster.chat`, the combined pipeline produced a marginally larger output than MiniMon alone by 61.22 KB, which is consistent with the repackaging overhead observed for that application in RQ2.

Table 6.4: RQ3 results: APK sizes (KB) under MiniMon alone versus Phase 1 + MiniMon combined.

Package Name	Original (KB)	MiniMon Only (KB)	Phase 1 + MiniMon (KB)
com.iroshni.urduquran16lines	2859.21	2843.21	2843.21
com.naouimi.henna	4180.12	4184.12	4179.18
com.androidlab.videoroad	1671.42	1663.42	1647.42
ru.alexko.regionalcodesua	686.01	682.01	682.01
com.storiestime	3111.75	3107.75	3107.75
com.pocket.aptitude	4176.95	4587.47	4587.47
com.tju.android.radar	915.74	911.74	911.74
ru.waptaxi.driver	2456.39	2448.39	2443.61
world.easysolution.russiangrammar	3232.88	3776.35	3776.35
com.ml.soompi	4932.50	4824.50	4824.50
zepeto.walkthrough.zepetotricks	2509.72	2509.72	2509.54
com.darkhorse.digital	3591.72	3571.72	3571.37
org.pocketworkstation.pckeyboard	2830.50	2738.50	2738.50
ru.alexko.regionalcodes	111.63	107.63	107.63
com.nextpick.makeupcontouring	3707.26	3699.26	3687.26
com.guca.whatssemcontato	4169.46	4968.24	4968.24
org.androworks.meteorgram	1771.95	1743.95	1707.95
com.devclev.forebet	3333.17	3333.17	3333.17
com.masrio.losefats2018	3929.49	3929.49	3913.49
kr.cmonster.chat	3166.80	3559.33	3620.55
net.aviascanner.aviascanner	5076.89	5032.89	5032.89
com.borisov.strelok	1008.40	992.40	992.40
com.technoskill.cricketscore	3242.31	3242.31	3242.31
mobi.infolife.installer	4033.98	4025.98	4025.98
egw.estate	3417.85	3365.85	3337.85
com.ldroid.stopwatch.simple	4127.96	4123.96	4123.96
com.amaze.filemanager	7796.42	8552.16	8546.88
com.tomer.alwayson	5062.66	4602.41	4586.41
Average	**3316.59**	**3388.62**	**3385.82**

Overall, the results suggest that permission-based debloating as a preprocessing step provides a modest but measurable additional reduction in APK size for a subset of applications. The improvement is most pronounced in applications where Phase 1 detected and removed multiple over-declared permissions, as the manifest reduction carries through to a smaller input for Phase 2. However, the additional reduction attributable to Phase 1 is small in absolute terms compared to the reduction achieved by MiniMon alone, and in the majority of applications it makes no difference to the final output size. The primary value of Phase 1 in the combined pipeline therefore lies not in dramatically improving MiniMon's size reduction, but in reducing the permission footprint of the application before usage-based debloating is applied, contributing to a cleaner and more accurate manifest in the final output.

7

Discussion

This chapter interprets the findings of the evaluation presented in Chapter 6, discusses the limitations of the proposed approach, identifies threats to the validity of the results, and positions the work relative to existing debloating techniques in the literature.

7.1 Interpretation of Results

The evaluation results reveal a mixed picture of the proposed two-phase debloating pipeline. The findings for each research question are interpreted below.

RQ1 established that over-declaration of Android permissions is a genuine and widespread phenomenon, affecting 44.7% of the applications in the dataset. This is a meaningful finding because it confirms the premise of Phase 1, that a non-trivial proportion of real-world applications carry permission declarations in their manifest that serve no purpose in terms of actual API usage. The fact that `VIBRATE` and `ACCESS_WIFI_STATE` are the most commonly over-declared permissions is consistent with the known behaviour of third-party advertising and analytics SDKs, which often declare permissions in their own manifests that are merged into the host application's manifest during the build process, regardless of whether the SDK's permission-dependent features are actually used. The presence of dangerous permissions such as `CAMERA`, `RECORD_AUDIO`, and `READ_PHONE_STATE` among the over-declared set is particularly noteworthy, as these permissions are visible to users during installation and contribute to a false impression of what the application can access. It is also worth noting that the 91 permissions classified as not in map represent the largest single category in the dataset, meaning the 44.7% figure is a lower bound on the true rate of over-declaration rather than a complete picture.

RQ2 demonstrated that Phase 1 can reduce APK size in the majority of affected applications, with an average reduction of 122.14 KB and a maximum of 234.43 KB. While these reductions are modest in relative terms, averaging 3.8% of the original APK size, they represent a consistent and safe improvement that comes at very low computational cost compared to call-graph-based approaches. The single case where size increased after repackaging, `kr.cmonster.chat`, is not a failure of the permission removal logic but rather a side effect of apktool's resource re-compilation, which is an acknowledged limitation of any approach that relies on

apktool for repackaging. More importantly, Phase 1 is safe by construction under the assumption that no permission-protected APIs are invoked via reflection or dynamic loading, since only over-declared permissions are removed and no bytecode is modified.

RQ3 produced the least straightforward result. The combined pipeline was better than or equal to MiniMon alone in 27 out of 28 applications, with Phase 1 producing additional size reduction in 11 of those, ranging from 0.18 KB to 36 KB. In the 16 applications where no additional reduction was observed, Phase 1 simply found no over-declared permissions to remove. The one case where the combined pipeline was marginally worse, kr.cmonster.chat, is attributable to the same apktool repackaging overhead discussed in RQ2. The improvement from adding Phase 1 is modest but consistent, and the primary value it brings to the combined pipeline is not dramatic size reduction on top of MiniMon, but a cleaner and more accurate manifest in the final output, one that reflects only the permissions the application actually uses, which has independent security and privacy value that the size metric alone does not capture.

7.2 Limitations

Several limitations of the proposed approach are worth acknowledging explicitly.

The permission-to-API mapping in `permissions_map.json` is finite and manually constructed, covering 47 permissions. The Android permission system contains a larger number of permissions than this, and any permission not present in the map is conservatively retained without analysis. This means Phase 1 cannot detect over-declaration for unmapped permissions, and the true rate of over-declaration in the dataset is likely higher than what the evaluation reports. Extending the map to cover more permissions would improve recall at the cost of additional manual effort to verify the correctness of each mapping.

Phase 1 relies on substring matching of Smali API signatures to detect permission usage. This approach cannot detect API calls made through Java reflection, dynamic class loading, or native code. If an application invokes a permission-protected API through any of these indirect mechanisms, Phase 1 will incorrectly classify the permission as over-declared and remove it from the manifest, potentially breaking functionality. While this scenario is uncommon in the applications examined, it represents a correctness risk that a production-grade implementation would need to address through additional analysis.

The dataset used in this evaluation is sourced from the MiniMon replication package and consists of 38 applications, of which only 28 had execution logs available for Phase 2. This is a relatively small dataset, and the applications were not selected to be representative of the broader Android ecosystem. Results may not generalise to larger or more complex applications, particularly commercial apps that use advanced obfuscation or anti-tampering mechanisms.

Phase 2 relies on pre-collected execution logs from the MiniMon replication package rather than logs collected from fresh usage sessions. These logs represent a specific and controlled set of user interactions that may not reflect how the applications are typically used. If a feature is used in practice but was not exercised during the monitored session, it will be treated as unused and stubbed out, which could affect the correctness of the debloated application.

7.3 Threats to Validity

Internal validity concerns whether the measurements and comparisons in the evaluation are accurate and unbiased. The primary internal threat is the use of APK size as the main metric for evaluating debloating effectiveness. As discussed in Section 6.2, APK size is influenced by repackaging overhead from apktool and does not directly reflect the amount of code removed. A more precise metric would be the number of methods stubbed out by MiniMon, though as discussed in Chapter 6, this metric would be identical for both pipelines in RQ3 since Phase 1 does not modify bytecode. The FlowDroid [5] timeout and callback limits applied during the RQ3 batch run may also have affected the quality of call graphs for some applications, potentially influencing which methods MiniMon identified for removal.

External validity concerns whether the results generalise beyond the specific dataset and conditions of this study. The dataset consists of relatively small open-source or independently developed applications sourced from a single research artifact. Commercial applications from major publishers, which tend to be larger, more obfuscated, and more likely to use third-party SDKs, may exhibit different permission over-declaration patterns and different responses to the debloating pipeline. The repackaging step in Phase 1 is also known to fail for a non-trivial proportion of commercial apps, as noted in the 3DNDroid study [3], which limits the applicability of the approach to apps that can be successfully decoded and repackaged by apktool.

Construct validity concerns whether the evaluation measures what it claims to measure. The use of APK size as a proxy for debloating effectiveness is an approximation. A debloated application that is only marginally smaller than the original may still have had significant unnecessary functionality removed at the bytecode level, while an application whose size did not change may still have a meaningfully reduced manifest footprint. Future work could supplement size-based measurement with dynamic analysis to verify that removed permissions are genuinely not exercised at runtime.

7.4 Comparison to Related Work

The proposed approach occupies a distinct position relative to the existing debloating tools surveyed in Chapter 3.

Compared to MiniAppPerm [2], the most closely related work, the key difference

lies in the trigger for permission removal. MiniAppPerm removes code associated with permissions that the user explicitly wishes to disallow, even if those permissions are actively used in the bytecode. The approach proposed in this thesis targets a different problem, permissions that the developer accidentally over-declared and that are never used in code. The two approaches are therefore complementary rather than competing. MiniAppPerm is appropriate when a user wants to disable a feature they do not need; the approach in this thesis is appropriate for cleaning up manifest declarations that should never have been there. Both independently arrived at bytecode scanning as the mechanism for permission-API detection, which provides mutual validation of the core technique.

Compared to MiniMon [7], which forms Phase 2 of the pipeline, the contribution of this thesis is to demonstrate that a lightweight static preprocessing step can reduce the permission footprint of the application before the more expensive dynamic phase is applied. MiniMon alone achieves significant code reduction based on usage data, but does not address the manifest-level over-declaration problem. The combined pipeline addresses both dimensions.

Compared to activity-based approaches such as AutoDebloater [9] and feature-oriented approaches such as XDebloat [10], the permission-based approach in this thesis operates at a finer and more targeted granularity, focusing specifically on the relationship between manifest declarations and bytecode usage rather than on user-visible features or activities. This makes it less flexible in terms of what can be removed, but also safer and more fully automatic, requiring no user input or feature annotation.

Compared to 3DNDroid [3], which achieves dynamic method-level debloating without APK modification, the approach in this thesis is simpler and more accessible but requires repackaging and resigning, which limits applicability to apps that can be processed by apktool. 3DNDroid's dynamic approach is more powerful but requires flashing a custom Android OS, which is impractical for most users.

8

Conclusion and Future work

8.1 Conclusion

This thesis proposed and evaluated a two-phase hybrid debloating pipeline for Android applications, combining static permission-based debloating with dynamic monitor-based debloating. The first phase is implemented as a custom tool that automatically identifies permissions declared in the `AndroidManifest.xml` whose corresponding Android API methods are never invoked in the application's smali bytecode, and removes them from the manifest without modifying any executable code. The second phase feeds the output of the first phase into MiniMon [7], a monitor-based debloating framework, which performs method-level debloating guided by pre-collected runtime execution logs.

The evaluation was conducted on a dataset of 38 real-world Android applications sourced from the MiniMon research artifact. The results demonstrate that over-declaration of permissions is a genuine and widespread phenomenon, affecting 44.7% of the applications in the dataset. Phase 1 successfully reduced APK size in 16 out of 17 affected applications, with an average reduction of 122.14 KB and a maximum of 234.43 KB, while safe by construction under the assumption that no permission-protected APIs are invoked via reflection or dynamic loading. The combined pipeline produced additional size reduction over MiniMon alone in 11 out of 28 applications where execution logs were available, confirming that permission-based debloating can serve as a meaningful preprocessing step that reduces the permission footprint of an application before usage-based debloating is applied.

The primary contribution of Phase 1 is not a dramatic improvement in code size reduction but rather a targeted and automatic cleanup of manifest declarations that should not have been present in the first place. Over-declared permissions, particularly dangerous ones such as `CAMERA`, `RECORD_AUDIO`, and `READ_PHONE_STATE`, represent an unnecessary expansion of an application's declared privilege set that misleads users and increases the attack surface without providing any functional benefit. Removing them automatically, safely, and without user input addresses a real problem in the Android ecosystem that existing tools do not directly target.

The work also demonstrates that the two debloating approaches are complementary. Permission-based debloating addresses manifest-level over-declaration that is invisi-

ble to usage-based approaches, while monitor-based debloating addresses code-level bloat that is invisible to manifest-level analysis. Combining them produces a more complete debloating outcome than either approach applied independently.

8.2 Future work

Several directions for future work emerge from the limitations and findings of this thesis.

The permission-to-API mapping in `permissions_map.json` currently covers 47 permissions. Extending the map to cover the full Android permission system, including permissions introduced in recent API levels, would improve the recall of Phase 1 and allow it to detect over-declaration in a broader set of applications. This could be partially automated by parsing AOSP [17] source code annotations programmatically rather than manually, reducing the maintenance burden as the Android API evolves.

The smali scanning approach used in Phase 1 cannot detect permission API calls made through Java reflection, dynamic class loading, or native code. Integrating a lightweight reflection analysis or taint-tracking step into the scanning phase would reduce the risk of false positives and make the tool safer for use on applications that make heavy use of these mechanisms.

The evaluation in this thesis relied on APK size as the primary metric for measuring debloating effectiveness. Future work could complement this with runtime measurements such as memory consumption, CPU usage, and startup time on real devices, providing a more complete picture of the practical benefits of debloating for end users.

The dataset used in this evaluation is relatively small and consists of independently developed applications. Evaluating the pipeline on a larger and more diverse dataset, including commercial applications from major publishers, would provide stronger evidence for the generalisability of the findings. However, this would require addressing the repackaging limitations of apktool for commercial apps, which may use anti-tampering mechanisms that prevent successful decoding and repackaging.

Finally, the current tool operates as a post-build analysis step that requires decoding a compiled APK using apktool. An interesting direction for future work would be to integrate the permission detection logic directly into the Android development workflow, for example as a Gradle plugin or Android Studio inspection, allowing developers to identify and remove over-declared permissions during development rather than after the fact. This would eliminate the need for reverse engineering the final APK entirely and make the tool accessible to a much wider audience without requiring any knowledge of smali bytecode or APK internals.

Bibliography

[1] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software Bloat Analysis: Finding, Removing, and Preventing Performance Problems in Modern Large-Scale Object-Oriented Applications," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER)*, ACM, 2010, pp. 421–426. DOI: 10.1145/1882362.1882448. [Online]. Available: https://dl.acm.org/doi/10.1145/1882362.1882448.

[2] F. Thung et al., "Towards Speedy Permission-Based Debloating for Android Apps," in *Proceedings of the IEEE/ACM 11th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, ACM, 2024, pp. 84–87.

[3] Z. Zhang et al., *Shelving it rather than Ditching it: Dynamically Debloating DEX and Native Methods of Android Applications without APK Modification*, 2025. arXiv: 2501.04963. [Online]. Available: https://arxiv.org/abs/2501.04963.

[4] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, ACM, 2011, pp. 627–638. DOI: 10.1145/2046707.2046779. [Online]. Available: https://dl.acm.org/doi/10.1145/2046707.2046779.

[5] S. Arzt et al., "FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM, 2014, pp. 259–269.

[6] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java Bytecode Optimization Framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, IBM Press, 1999, pp. 125–135.

[7] J. Liu et al., "MiniMon: Minimizing Android Applications with Intelligent Monitoring-Based Debloating," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, ACM, 2024, pp. 1–13.

[8] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, ACM, 2012, pp. 217–228.

[9] J. Liu et al., "AutoDebloater: Automated Android App Debloating," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2023, pp. 2090–2093.

[10] Y. Tang et al., "XDebloat: Towards Automated Feature-Oriented App Debloating," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4501–4520, 2022.

[11] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast Unfolding of Communities in Large Networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, P10008, 2008. DOI: 10.1088/1742-5468/2008/10/P10008. [Online]. Available: https://arxiv.org/abs/0803.0476.

[12] "Apktool: A Tool for Reverse Engineering Android APK Files. "[Online]. Available: https://apktool.org/.

[13] Android Developer Documentation. "Android Manifest Permission Reference. "[Online]. Available: https://developer.android.com/reference/android/Manifest.permission.

[14] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Rasthofer, "On Demystifying the Android Application Framework: The Role of Utility Apps, Daemon Apps, and Hidden Communication," in *Proceedings of the 25th USENIX Security Symposium*, USENIX Association, 2016, pp. 1–18.

[15] B. Campbell. "jose4j: A Java Implementation of the JOSE Specification Suite (JWS, JWE, JWK). "[Online]. Available: https://bitbucket.org/b_c/jose4j/wiki/Home.

[16] Android Developer Documentation. "Android SDK Build Tools — zipalign and apksigner. "[Online]. Available: https://developer.android.com/tools.

[17] Android Open Source Project. "frameworks/base/data/etc/platform.xml. "[Online]. Available: https://source.android.com.