MASTER IN COMPUTER SCIENCE

Towards Enhanced Android Third-Party Library Detection through Runtime-Assisted Static Analysis

Master's Thesis

Dario Kaufmann

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

January 2026

Prof. Dr. Timo Kehrer
Thomas Sutter

Software Engineering Group
Universität Bern

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

The widespread use of third-party libraries in Android applications facilitates rapid development but introduces significant security risks. Current third-party library detection tools primarily rely on static analysis, which is hindered by obfuscation and dynamic code loading. This work addresses these challenges by exploring a hybrid approach that combines runtime extraction with enhanced static analysis.

We performed an in-depth analysis to identify the specific limitations of LibPecker, a state-of-the-art detection tool. LibPecker was selected as our baseline because it achieved the highest F1 score in previous large-scale benchmarks compared to other similarity-based approaches. Our analysis revealed that its accuracy is limited by strong assumptions regarding package hierarchy, which are often violated by common obfuscation techniques such as package flattening and package repacking.

To address the visibility gap caused by dynamic code loading, we developed a prototype using the Frida instrumentation toolkit to hook into an application's code-loading mechanisms and extract dynamically loaded DEX files directly from memory. We validated this approach using an official Android demo application and the real-world application Adobe Acrobat Reader. These experiments successfully demonstrated the feasibility of capturing code modules absent from the static APK, providing a more complete view of an application's library dependencies.

Furthermore, we evaluated different improvements for LibPecker. We investigated prefiltering mechanisms to reduce computational workload and found that the existing prefiltering method can be significantly optimized through parameter fine-tuning. However, a newly proposed prefiltering mechanism based on Android API call comparisons did not outperform the optimized version of the existing method, and combining the two prefilters did not yield significantly better results.

By implementing targeted improvements to the core detection algorithm, we improved the F1 score of LibPecker by 10 percentage points and increased its processing speed by approximately 20$\times$ through implementation-level optimizations and by an additional 4$\times$ through adaptations to the prefilter and algorithm, resulting in an overall speedup of approximately 90$\times$. Despite these advancements, performance evaluations on complex real-world applications such as WhatsApp, Signal, and Firefox indicate that execution speed remains a bottleneck for large-scale deployments. We conclude that current third-party library detection has not yet reached a level of maturity suitable for seamless integration into real-world production scenarios. Furthermore, current approaches largely ignore cross-platform applications. Future work should move beyond structural pattern comparison and investigate semantic comparison to better handle the challenges posed by advanced obfuscation techniques.

Contents

List of Abbreviations **4**

1 Introduction **5**
1.1 Research Questions 6
1.2 Contribution 7
1.3 Thesis Organization 7

2 Related Work **9**
2.1 Clustering Approaches 9
2.2 Similarity Approaches 10
2.3 Code Extraction 12

3 Background **14**
3.1 Existing Datasets and Benchmark 14
3.2 Obfuscation Tools 16
3.3 Package Renaming 20

4 Approach **23**
4.1 Goal and Scope 23
4.2 Existing Algorithms and Tools 24
4.3 Findings from LibPecker Analysis 32
4.4 System Design for Runtime Detection 34
4.5 Proposed Improvements 36

5 Implementation **39**
5.1 Dynamic DEX Extractor 39
5.2 LibPecker Improvements 42

6 Evaluation **45**
6.1 Evaluation Setup 45
6.2 Results 50

7 Discussion **60**
7.1 LibPecker Improvements 60
7.2 Runtime Detection 62
7.3 Threats to Validity 63

8 Future Work **64**

9 Conclusion **65**

List of Figures **66**

List of Listings **67**

List of Tables **69**

Bibliography **74**

A Class Merging **76**
A.1 Vertical Class Merging 76
A.2 Horizontal Class Merging 78

B Setup Device **91**

C Code References **93**
C.1 Vertical Merging Policies 93
C.2 Horizontal Merging Policies 95
C.3 Other R8 Classes 100
C.4 Android Runtime 100
C.5 Libcore 100

List of Abbreviations

AAB	Android App Bundle
API	Application Programming Interface
APK	Android Package Kit
ART	Android Runtime
CDG	Class Dependency Graph
CVE	Common Vulnerabilities and Exposures
DCL	Dynamic Code Loading
DEX	Dalvik Executable
JVM	Java Virtual Machine
SDK	Software Development Kit
TPL	Third-Party Library

1 Introduction

Smartphones have become ubiquitous: almost everyone owns a smartphone and regularly downloads and uses mobile applications. As a result, the smartphone ecosystem has become a central part of everyday life. Smartphones are used to organize personal activities, communicate with friends and business partners, and perform financial transactions such as purchasing goods and making payments. Securing this platform is therefore of critical importance, not only for private users, but also for companies that provide mobile devices to employees and connect them to internal corporate systems.

Protecting a smartphone involves addressing multiple aspects, one of which is the security of installed applications. In the case of Android, the world's most widely used mobile operating system [1], applications can be obtained from official app stores such as Google Play as well as from third-party sources like F-Droid. Applications do not consist solely of code written by the app developers themselves; they also include third-party libraries (TPL), many of which are open source. These libraries facilitate faster software development and can improve overall security, as they are widely used and tested by the community [2].

When vulnerabilities in such libraries are discovered, they are typically reported in public vulnerability databases, such as the Common Vulnerabilities and Exposures (CVE) catalog [3]. This information, however, is only useful if it is known which libraries are included in a given application. For many apps, this information is not readily available. In the case of closed-source applications, users must rely on developers to regularly update the libraries they use. In the context of security, reliance on trust alone is undesirable. Consequently, approaches for detecting TPLs within applications are required.

Identifying the exact libraries and their versions included in an application enables systematic comparison against known vulnerabilities and allows for the detection of potential security issues, thereby improving the overall security of Android devices [2, 4, 5].

Beyond security considerations, knowledge of included TPLs provides additional benefits. For instance, it enables the verification of license compliance and supports software quality analysis by identifying unnecessary or suboptimal library dependencies. Additionally, it helps address privacy concerns by enabling the detection of tracking and advertising libraries embedded in applications [6, 7].

State-of-the-art TPL detection approaches typically analyze the DEX files contained in an APK and compare them against a database of known libraries. However, many applications are obfuscated using tools such as R8, Allatori, or DashO. These tools intentionally modify the structure of an application, including its embedded TPLs to conceal code internals and obscure the original intent of the program.

Such obfuscation is commonly employed to protect intellectual property or in some cases to hide malicious behavior [8]. As a result, direct comparisons between library code and application code become highly challenging.

To mitigate this issue, existing TPL detection tools attempt to identify obfuscation-resilient features. However, defining features that remain robust under diverse and aggressive obfuscation strategies is inherently difficult. A prior study has shown that the resulting detection accuracies of existing tools fail to meet the required levels of accuracy and scalability [5]. Moreover, current state-of-the-art approaches operate exclusively on static APK files and completely ignore libraries that are loaded at runtime. Dynamic Code Loading (DCL) allows an app to expand its functionality at runtime. Code that is downloaded and executed during application execution is not part of the static APK and therefore remains invisible to static analysis-based techniques [9]. Different reasons exist to load code dynamically, ranging from legitimate business requirements to security-related attacks. Depending on the intended goal, the source of the dynamically loaded code can either be a remote server or files that are already included in the application, typically in encrypted form.

One reason is to prevent static reverse engineering aimed at protecting intellectual property or circumventing static malware analysis [10]. The corresponding code is delivered with the app in encrypted form. At application startup, code is executed to decrypt the bytecode at runtime. Often, the decryption procedure is implemented using native code [11]. When analysing the application statically, only a small portion of the code, namely the part that initializes the decryption process, is available. The application logic and TPLs are encrypted and therefore cannot be analyzed [12].

As another reason, a developer might choose DCL to enable the addition of new features to an already installed app, for example, to add paid premium features in a basic demo version after a user upgrades their subscription, or to maintain different versions of an application as part of a software product line without distributing a separate app for each version [9]. We limit our analysis to dynamic code loading in which new code is incorporated into the application at runtime and does not actively hide its behavior.

Because this code is added while the application is running, it can introduce potential security issues if the dynamically loaded code is modified and the loading app does not verify its integrity [13]. Moreover, code that is only loaded at runtime may not be fully visible to static analysis tools or pre-publication checks performed by the Google Play Store, allowing malicious behavior to evade detection [9, 14].

In this work, we aim to address these gaps in current research. We conduct a feasibility study to investigate whether useful runtime artifacts can be extracted and subsequently analyzed using static TPL detection tools. Furthermore, we extend and improve an existing state-of-the-art tool to better understand its limitations and to enhance both detection accuracy and runtime performance.

1.1 Research Questions

In this work, we aim to address these limitations by investigating complementary approaches to TPL detection. To this end, we formulate the following research questions:

- **R1** Is it feasible to extract dynamically loaded DEX files during runtime execution of Android applications?
- **R2** How can the runtime performance and detection accuracy of a state-of-the-art static TPL detection tool be improved under common obfuscation techniques such as package renaming?

Regarding the research question **R1**, we address the following subquestions:

- **R1.1** Is the set of DEX files initially dumped at runtime identical to the DEX files contained in the statically analyzed APK?

- **R1.2** Can dynamic code loading at runtime be detected using the proposed hooking-based method?
- **R1.3** Can the proposed method detect loaded TPLs that are not present or observable during static analysis?

Research question **R1.1** serves to determine whether the complete application code can be recognized when no dynamic code loading occurs. This establishes a baseline by assessing whether the DEX files extracted at runtime correspond to those contained in the statically analyzed APK. The second subquestion, **R1.2**, addresses the identification of an appropriate point in time for extracting dynamically loaded code. While such a point could be selected arbitrarily, for example using a fixed time interval, detecting the actual loading events enables the grouping of code that is loaded together during execution. Finally, **R1.3** ensures that the extracted code indeed corresponds to dynamically loaded components. Distinguishing between code originally present in the APK and code added at runtime is essential, as only newly introduced code provides additional value beyond existing static analysis techniques.

To answer Research Question **R2**, we formulate the following subquestions:

- **R2.1** How does package renaming obfuscation affect the detection accuracy of LibPecker?
- **R2.2** To what extent can targeted optimizations improve both runtime performance and detection accuracy without altering the fundamental detection approach?

1.2 Contribution

This work proposes a hybrid approach for enhanced TPL detection to identify both statically included and dynamically loaded code, and makes the following key contributions to TPL detection in Android applications:

- Runtime detection mechanism: We implement a prototype capable of detecting and extracting dynamically loaded code during application execution, demonstrating feasibility on both a demo and a real-world application.
- Improved accuracy and performance: By enhancing a state-of-the-art static detection tool with runtime insights, we achieve a 10 percentage point increase in F1 score and a 90× improvement in detection speed.
- We provide an in-depth study of obfuscation, with particular focus on R8's class merging, package flattening, and package repacking, and demonstrate how these techniques affect an existing detection tool.
- Supporting technical analyzes: We examine DEX file structures and the functionality of memory dumping tools (e.g., frida-dexdump) to enable accurate runtime code extraction.

Together, these contributions address limitations of existing static TPL detection approaches, improve detection coverage and efficiency, and provide insights into the challenges posed by code obfuscation and dynamic loading in Android applications.

1.3 Thesis Organization

This thesis is organized into nine chapters and several appendices that detail the research, evaluation, and findings regarding enhanced TPL detection in Android applications. The structure is as follows:

- **Chapter 1: Introduction** Provides an overview of the research problem, specifically the challenges posed by obfuscation and dynamic code loading in Android apps. It defines the research questions and outlines the primary contributions of this work.
- **Chapter 2: Related Work** Reviews existing literature on static TPL detection, categorizing prior tools into clustering and similarity-based approaches. It also discusses existing methods for code extraction from applications.
- **Chapter 3: Background** Presents the foundational technical knowledge, including an overview of existing datasets and benchmarks. It details various obfuscation tools and focuses on package renaming techniques such as flattening and repacking.
- **Chapter 4: Approach** Defines the goal and scope of this work. This chapter includes an in-depth analysis of the LibPecker tool to identify its limitations and describes the system design for the proposed runtime detection and static analysis improvements.
- **Chapter 5: Implementation** Details the technical realization of the Dynamic DEX Extractor prototype and the specific enhancements implemented for the LibPecker tool to improve performance and accuracy.
- **Chapter 6: Evaluation** Describes the experimental setup and presents the results. This includes testing runtime detection on a demo application and Adobe Acrobat Reader, as well as benchmarking the improved LibPecker against an existing dataset.
- **Chapter 7: Discussion** Interprets the evaluation results and explores the limitations of the proposed approach and discusses threats to the validity of the findings.
- **Chapter 8: Future Work** Outlines potential directions for future research, such as semantic code comparison and extending detection to cross-platform mobile applications.
- **Chapter 9: Conclusion** Summarizes the key findings and the extent to which the initial research questions were answered.

Additionally, the **Appendices** provide supplementary information, including detailed analyzes of R8 class merging policies (Appendix A) and a guide for the technical setup of the evaluation device (Appendix B). The linked code is referenced in Appendix C.

2 Related Work

Research related to this thesis spans two distinct domains: Third-Party Library (TPL) detection and runtime code extraction. While TPL detection focuses on identifying and versioning software components within an application, code extraction deals with the technical challenge of retrieving bytecode that is hidden or loaded dynamically at runtime.

In the area of static TPL detection for Android applications, several studies have been published in recent years. These approaches have evolved from relatively simple, whitelist-based methods [15] to more sophisticated techniques that account for code obfuscation and aim to detect libraries at the version level. However, the majority of recent studies can be broadly categorized into two groups [7, 15, 16]. The first approach decomposes the applications into components. These components are considered library candidates. Using hashing or similarity metrics, the library candidates are clustered. If multiple candidates from different applications belong to the same cluster, they are regarded as identical and assumed to be the same unknown library. As a consequence, only frequently used libraries can be detected [6, 7, 17, 18]. This approach does not require prior knowledge of existing libraries. Instead, it requires a dataset of applications that include several shared libraries [16].

The second approach utilizes a predefined set of known libraries for detection. To extract libraries from applications, various studies have proposed different features to measure the similarity between application components and libraries. Only libraries included in the reference set can be identified using this method [7, 15, 16, 19].

In parallel, research in the area of code extraction focuses on overcoming the visibility gaps caused by packers and dynamic code loading. These techniques are not detection methods themselves, but rather provide the means to reconstruct a complete codebase from memory.

2.1 Clustering Approaches

LibRadar [17] counts the number of distinct Android API calls and uses the resulting frequencies as features. By applying a hash function, a unique fingerprint can be generated for each library package. The authors argue that this approach is resilient to obfuscation, as Android API call frequencies are not affected by obfuscators. However, the package structure itself may still be altered through obfuscation and is therefore not resilient against this obfuscation technique. The previously generated feature hashes are then used to cluster packages into groups, with each group consisting of packages that share exactly the same features. The authors used a dataset of one million apps to construct these clusters. To detect libraries in a given application, features of this application are extracted, hashed, and compared with the preconstructed clusters.

LibD [20] leverages the internal dependency structure of applications to detect and cluster libraries. To extract potential libraries from an app, the relationships among packages, classes, and methods are analyzed. In this way, instances of potential libraries are created, each consisting of one or more packages. Features are then generated for these candidates: first, a feature is created for each method by hashing its opcodes and combining them in a specific order. For each class, the corresponding method features are collected, sorted, and hashed to produce a class-level feature. Finally, the same process is applied at the library level: class features are sorted and hashed to generate a single feature representing the entire library. Libraries can then be clustered into groups, with each instance in a group sharing identical features. The authors show that this approach is resilient against renaming. But package flattening might reduce the general obfuscation resilience.

In contrast to the two previously described approaches, which are based on package structure or package relationships respectively, **LibHawkeye** [18] builds intra-app dependency graphs to detect library boundaries. In these graphs, components that are weakly connected are considered potential library instances. Features of these candidates are then calculated and clustered. To evaluate the approach, the authors used a dataset of 1,000 unobfuscated apps.

Cluster-based approaches have been shown to face challenges in distinguishing between library versions, making version-level detection difficult in many cases or even not feasible at all [15].

2.2 Similarity Approaches

To determine whether an application contains a specific library, class matching can be applied. In this approach, classes within the application are compared to those in known libraries. If the classes are sufficiently similar, they are considered part of a library. **LibScout** [6] employs two main metrics for this matching process: the package structure and fuzzy representations of method signatures. The fuzzy method signatures consist of the return type and argument types of each method. If a type does not belong to the Android SDK, it is replaced with a placeholder to account for potential renaming through obfuscation. This results in an approach that is resilient to package, class, and identifier renaming, as well as control flow modifications. However, it is not resilient to package flattening or dead code removal.

ORLIS [7] uses fuzzy method signatures as features, similar to those used in LibScout. However, instead of relying solely on syntactic information like LibScout, ORLIS constructs a fuzzy call graph and uses a string-based representation as a feature. This graph includes all fuzzy method signatures of the methods that are reachable from the method for which the feature is generated. These string features are sorted and concatenated. A digest of the resulting string is then used to perform a coarse-grained filtering of potential library matches. After this initial filtering, the method features are used to assess class similarity for more fine-grained analysis. For each class of the given app, all method features from it and its superclasses are aggregated and compared pairwise between the libraries and the application, again using the filter digests. A comparison of different tools demonstrates that ORLIS exhibits remarkable resilience to obfuscation [15]. However, it does not support detection at the library-version level.

Rather than method implementations, **LibPecker** [4] considers dependencies in the code to calculate similarity. Three types of dependencies are considered: (1) class A inherits from class B, (2) class B is used as a field in class A, and (3) class B is used as a parameter or return type in a method of class A. Classes A and B might be obfuscated names. For each class B on which class A depends, a dependency signature is generated. This signature consists of three elements: (i) whether class B belongs to the same package, a different package, or is a system class; (ii) whether class B is an array type; and (iii) if class B is a system class and thus not obfuscated, the class name of B. The class signature of class A is defined as the combination of all its dependency signatures. Since the generated class signatures may differ due to dead code elimination, a fuzzy class matching algorithm computes a similarity score, which is used to determine whether an application package belongs to a given library package. As the approach relies on the package structure, it is not resilient to package flattening obfuscation.

Like LibPecker, **LibID** [21] also leverages class dependencies. However, LibID extends its feature set by incorporating fuzzy method signatures and call flow graphs. Two variants are provided: LibID-S, which is optimized for scalability, and LibID-A, which focuses on accuracy. LibID-S performs class and dependency matching and is resilient to repacking techniques, provided that code shrinking is not applied simultaneously. LibID-A enhances this profiling by introducing a dependency graph and extends dependency matching with additional constraints on invocations, inheritance, and interfaces. Since LibID employs more features than LibPecker, it achieves significantly better detection performance at the version level [15].

ATVHunter [16] detects TPLs in Android apps by first removing host code and isolating library candidates using a Class Dependency Graph (CDG). It then extracts control flow graphs and opcode sequences to generate coarse- and fine-grained features. The features do not rely on the package structure and are therefore resilient to package flattening. The features are compared against a large database using fuzzy hashing across sliding-window opcodes and similarity scoring to identify the library versions.

Despite the sliding window approach, ATVHunter is susceptible to code flow randomization because the insertion of redundant opcodes alters the relative offsets of the original opcodes within the sliding window. To address this limitation, **LibScan** [22] employs call-chain-opcode similarity. First, code features are extracted to create a fingerprint for each class to enable comparison. These features include keywords such as *class* and *static*, as well as primitive data types like *int* or *int[]*. Overall, these features remain unaffected by code obfuscation. For app classes sharing the same signature, method-opcode similarity analysis is performed. In this step, the similarity between the opcodes of methods in the two classes is calculated. For methods exhibiting high similarity, a call-chain-opcode similarity comparison is conducted, where the opcodes of subsequently invoked methods are taken into account and compared.

Libloom [23] compares classes from the application and the library using class signatures. To achieve this, it employs a fuzzy method signature matching technique similar to those used in previous studies. However, methods identified as constructors, which are not renamed during obfuscation, are retained. Additionally, the approach considers class fields and the hierarchical position of the class, identified through the *extends* and *implements* keywords. The final signature of a class is defined as the set comprising all these combined signatures. Due to code shrinking and method deletion, the signature set of an obfuscated library class may be a subset of the original class's signature set. This subset inclusion problem is addressed using a Bloom filter. To efficiently eliminate most candidate matches in a short time, a package-level overlap measurement is employed in the first stage. To mitigate the effects of repackaging and flattening obfuscations, Libloom uses heuristics to assess similarity. In the second stage, a class-level subset comparison is conducted. The method can output a range of possible library versions. While the approach is designed to be robust against code shrinking, obfuscation techniques that introduce additional code can reduce its effectiveness.

LibHunter[19] is the first approach to explicitly consider the impact of code optimization on TPL detection. For instance, Call Site Optimization removes method parameters that are either unused or consistently invoked with a constant value. Prior work has commonly used fuzzy method signatures under the assumption that the number of parameters remains unchanged during obfuscation. However, this assumption does not hold under optimization. Similarly, Inlining Optimization is applied when a method is invoked by only a few callers; in such cases, the entire method body is inserted at the call site, which alters the structure and reduces the expressiveness of call flow graphs.

LibHunter operates in three stages. In the signature-based class matching stage, both field-level and method-level features are extracted. While it adopts fuzzy method and field signatures like previous work, LibHunter enhances them by analyzing the original library code to anticipate possible optimization effects. For example, if analysis determines that a parameter may be removed during optimization, the corresponding symbol in the fuzzy signature is marked accordingly. In the second stage, similarity-based method matching is performed using opcode and string features. This step identifies both fully and partially matched methods. Partial matches capture scenarios where an app method incorporates code from multiple TPL methods due to inlining. The third stage is cross-inlining method matching, where the tool analyzes potential inlining in the original library and compares synthesized method features with app code. Based on the comparisons performed in these three steps, LibHunter calculates similarity scores to identify the most probable TPL version. While the approach is specifically designed to be resilient against optimization, it relies heavily on accurately modeling and simulating optimization behaviors.

SAD [2] represents the most recent approach in TPL detection. Zhou et al. identified limitations in the version-level detection capabilities of prior tools and developed a novel method to address this issue. During the pre-processing phase, SAD utilizes a CDG to narrow down the set of potential library matches. In this directed graph, nodes, representing classes, are labeled as *static*, *abstract*, *default*, or *interface*, while edges, indicating relationships between classes, are labeled as either *implements* or *extends*. These structural features are resilient against code obfuscation. For each node, a feature is computed by aggregating the labels of its neighboring nodes, sorting them, and applying locality-sensitive hashing. This process is performed for both app and library classes. The similarity between all pairs of nodes, comprising an app class and a library class, is then used to filter and construct the candidate list for further analysis. For class matching, SAD categorizes classes as stateful or stateless and compares them based on opcode overlap and field operations. To further reduce false positives and enhance semantic consistency, SAD generates class summaries using method call sequences and field operations, then compares these summaries to determine match confidence. Finally, SAD evaluates the structural similarity between the CDGs of the app and library to confirm version-level matches.

All the discussed detection tools operate on static files. Consequently, these approaches are not resilient to obfuscation techniques such as dynamic code loading and class encryption, which modify the code at runtime.

2.3 Code Extraction

All previously described TPL detection tools rely on static analysis. Code that is loaded dynamically at runtime is therefore ignored and remains invisible during analysis. However, prior research exists on DEX file extraction that operates at runtime, particularly in the context of packed applications that deliberately attempt to conceal their code.

DexHunter [24] is a tool for extracting hidden code from packed Android applications. It uses a modified Android Runtime (ART) to read OAT files, which are produced by ahead-of-time compilation and contain the corresponding DEX code. To retrieve the complete code base, DexHunter proactively initializes all classes. However, it does not capture code that is loaded dynamically during app execution. Although this limitation could be addressed by hooking methods related to dynamic class loading, the authors leave this as future work.

In addition to code extraction from packed applications, **DexLego** [12] aims to detect self-modifying code and reconstruct it into executable form. This enables static analysis tools to inspect code that would otherwise be invisible due to runtime modification. The just-in-time compiler assigns an index to each instruction, and DexLego identifies modified code by detecting different instructions that share the same index at runtime. To achieve this, DexLego employs a customized ART.

To further address the challenges of dynamic code loading and reflection-based obfuscation, the hybrid framework **DLCDroid** [25] monitors application behavior in an anti-emulation-resistant environment. By utilizing API hooking at runtime, the tool captures unresolved method signatures and arguments that typically thwart static analysis replacing reflective calls with their static equivalents to enable downstream tools to conduct accurate taint analysis on previously hidden execution paths.

PackerGrind [26] extracts DEX files from packed applications by collecting dynamically released code at runtime. It identifies suitable extraction points by defining data collection hooks within the Android Runtime.

DroidTrace [27] uses `ptrace`, a debugging mechanism, to monitor applications at runtime and analyze the behavior of dynamically loaded code. To trigger dynamic loading, DroidTrace does not rely on UI-interaction simulation. Instead, it performs a static analysis of the application by constructing a function call graph to identify execution paths that lead to dynamic code loading. For each identified path, a modified version of the application is generated in which additional trigger code is injected to ensure that the function responsible for the dynamic behavior is executed. The resulting APKs are then installed on an emulator and analyzed at runtime to observe the behavior of the dynamically loaded code.

A more robust approach for unpacking applications and extracting DEX files is proposed by the tool **BPFDex** [28]. Their technique operates at the kernel level and therefore below the layer at which application packers can exert influence. This allows their method to effectively circumvent common packing strategies.

3
Background

This chapter provides an overview of the necessary background information. We present existing datasets, discuss their limitations, and summarize the first existing benchmark for comparing TPL detection tools. Furthermore, we provide an overview of the most important obfuscation tools and their techniques.

3.1 Existing Datasets and Benchmark

To evaluate and benchmark their methodologies, researchers have developed and utilized various datasets as ground truth. For a proper evaluation of the tools, it is important to have a set of applications for which the included libraries are known. These datasets can be broadly categorized into three types: datasets derived from open-source applications, synthetic datasets, and datasets constructed from partially obfuscated real-world applications.

Open-Source-Based Datasets One of the foundational datasets in TPL detection research is the dataset by ORLIS [7]. This dataset comprises open-source Android applications from which the used libraries were extracted directly from the source code. The code was then compiled and obfuscated using ProGuard [29], DashO [30] and Allatori [31]. Subsequent studies have used this dataset or extended and refined it. Zhan et al. [16] extended the dataset by selecting 88 applications from the original dataset and generating additional obfuscated variants using DashO. Wu et al. [22] contributed an additional 51 unique applications, each subjected to three standard obfuscation levels of the R8 obfuscator [32]. During its evolution, certain modifications were applied: Wu et al. [22] filtered out applications lacking TPLs, and Zhou et al. [2] corrected several inaccuracies present in the dataset. Notably, Zhang et al. [21] critiqued the original dataset by asserting that all applications labeled as ProGuard-obfuscated were, in fact, not obfuscated.

The final dataset used by Zhou et al. [2] comprises 276 distinct applications (225 from ORLIS and 51 from LibScan), from which a total of 1,253 app versions were generated. These applications include 241 distinct libraries, with a total of 451 different library versions represented in the dataset. On average, there are two different versions per library in the dataset.

Beyond the ORLIS dataset, several studies have constructed datasets directly from open-source Android applications, typically sourced from repositories such as F-Droid [33] or GitHub. For example, Xie et al. [19] collected a dataset of 200 open-source apps from F-Droid and generated five different versions for each app, including variants that are non-obfuscated, optimized, obfuscated, or combinations thereof. They used D8 [34] for optimization and R8 for obfuscation.

The most recent and largest dataset is provided by Gu et al. [5]. They built a collection of 6,055 real-world applications obtained from GitHub and F-Droid. The App-Library-Mapping was extracted from the source code using a specialized tool that processes diverse Gradle build configurations. The dataset includes dependencies retrieved from online repositories such as Maven Central, as well as local binary files embedded within the project's directory structure. More than half of the applications are built and obfuscated using R8. The dataset stands out due to its diversity and real-world relevance, as it includes projects ranging from small to large scale. However, to evaluate the impact of different obfuscation techniques, this dataset refers to more controlled datasets, such as those introduced previously.

The dataset described by Gu et al. is used to compare ten of the most recent TPL detection tools [5]. It represents the first large-scale comparison in this field. In total, in addition to more than 6,000 applications, 5,756 unique TPLs and 15,274 library versions were considered.

The most important findings are summarized in Table 3.1. The scalability-optimized version of LibID (LibID-S) is indeed faster than the accuracy-oriented variant (LibID-A). Interestingly, LibID-S also achieves a slightly higher detection rate than LibID-A across all evaluation metrics. As expected, the clustering-based approaches yield comparatively poor results. Among these, ORLIS exhibits a notably lower detection performance than the other tools.

With respect to the F1 scores, LibPecker achieves the best performance, reaching 60% at the library level and 49% at the version level. In both cases, this corresponds to an improvement of approximately eight percentage points over the second-best approach.

Not all tools were evaluated using the same number of libraries. Since most of the tools are similarity-based, meaning that application code is compared against each library's code, the number of libraries included has a significant impact on runtime. Consequently, the reported runtime averages are only comparable within the same group.

Overall, the results indicate that current approaches are not yet highly effective. The highest F1 score at the library level is achieved by LibPecker, with a value of only 60%. However, the corresponding runtimes are high and do not support instant analysis.

As of the end of January 2026, neither the dataset nor the benchmark is publicly available.

Synthetic Dataset Zhang et al. [21] introduced a synthetic dataset. This dataset includes 69 popular libraries, covering a total of 1,444 distinct versions sourced from Maven, JCenter, GitHub and official library websites. For each library, representative code snippets implementing core functionalities were manually extracted from official documentation and example projects. These snippets, including their import statements and dependencies, were compiled into synthetic applications and obfuscated using ProGuard. Due to their synthetic nature and simplified library usage, these applications tend to retain only a small subset of library classes after code shrinking.

Partially Obfuscated Real-World Applications Another approach involves the use of partially obfuscated real-world applications, as exemplified by Zhang et al. [4]. This dataset leverages the observation that not all package names are obfuscated during the Android build process. By exploiting these residual identifiers, the authors automatically constructed a true-positive ground truth. The dataset comprises 9,834 applications and 310 distinct libraries, with an average of approximately 3.45 libraries per application. However, this dataset does not account for library versioning and does not include a false-positive analysis.

Table 3.1: Benchmark Results of Gu et al [5]: Comparison of TPL detection tools at library and version level

			Library Level [%]			Version Level [%]		
Method	Profiling	Avg. Time	Recall	Precision	F1	Recall	Precision	F1
Clustering-based								
Libradar	no	**9s**	9.19	13.58	13.58	–	–	–
LibD	no	447s	41.33	16.24	23.32	–	–	–
Similarity-based (all libraries)								
Orlis	yes	233s	1.22	14.74	2.26	0.79	1.74	1.08
LibScout	yes	38s	41.07	14.81	21.77	38.54	3.97	7.19
LibLOOM	yes	38s	57.15	18.81	28.31	50.24	7.25	12.67
Similarity-based (reduced library set)								
LibPecker	no	223s	47.11	83.15	**60.15**	42.33	58.96	**49.28**
LibScan	no	43s	13.98	**89.13**	24.18	12.87	60.06	21.20
LibID-S	yes	3679s	34.30	87.62	49.30	31.44	**60.41**	41.35
LibID-A	yes	3977s	33.61	84.32	48.06	30.45	57.27	39.76
LibHunter	no	342s	**81.79**	39.07	52.87	**76.48**	17.55	39.69

3.2 Obfuscation Tools

Several obfuscators are available for Android applications. Two commercial products are Allatori, developed by Smardec Inc. [31], and DashO, distributed by PreEmptive [30]. Their use is not limited to Android applications but applies to any Java project. Both products are actively maintained, receiving updates in September 2025 [35] and November 2025 [36], respectively. Allatori focuses on code obfuscation, APK size reduction, and performance improvement. In addition to code obfuscation, DashO provides protection against unauthorized code analysis and debugging. Allatori and DashO do not describe in detail how their obfuscation techniques operate. They merely provide high-level overviews of the techniques they employ. Therefore, it is difficult to analyze the effectiveness of their strategies.

Another tool is R8. It is the successor to ProGuard and uses the same configuration interface. R8 is open-source and actively maintained by Google. It is integrated into the Android Studio development environment and therefore requires no separate installation. Before R8, ProGuard was the default tool included with Android Studio. A recent study analyzing obfuscation tools reports that ProGuard remains the most widely used tool, followed by Allatori [37].

This section provides an overview of the obfuscation and optimization techniques used by this tools. Insights are gained from the documentations [30, 31, 38] and from analysis of the tools' output. R8's class-merging capabilities are analyzed in depth in section 4.2.1 due to their impact on TPL detection.

3.2.1 Allatori & DashO

Control Flow Randomization The goal of control flow randomization is to alter the code so that it becomes harder to understand during reverse engineering. While the semantics remain unchanged, the structural form of the code is modified. The research literature discusses numerous strategies. However, in 2018, Kumar and Kurian identified a significant gap between research and practice in commercial tools [39].

One simple strategy is to insert dead code to complicate the control flow (Listing 3). This approach, however, can often be undone automatically by modern decompilers that perform optimization. A more sophisticated example of control flow randomization, as employed by DashO, is illustrated in Listing 1, where the three simple statements are embedded within a while loop to alter the order of their definitions. Such transformations disrupt structural patterns that many static analyzers rely on.

```java
//Before obfuscation
public int compute(int a, int b) {
    int x = a + b;
    int y = x * 2;
    return y - 3;
}

// After obfuscation
public int compute(int a, int b) {
    int state = 42;
    int x = 0, y = 0, result = 0;

    while (state < 500) {
        state = switch (state) {
            case 0 -> {
                // original: return y - 3;
                result = y - 3;
                yield 743;
            }
            case 42 -> {
                // original: int x = a + b;
                x = a + b;
                yield 4;
            }
            case 4 -> {
                // original: int y = x * 2;
                y = x * 2;
                yield 0;
            }
            default ->
            // dead or confusing blocks can be added as noise
            result = 4;
        };
    }
    return result;
}
```

Listing 1: Simple Method before and after control flow randomization

Although the keyword `goto` is reserved but unused at the Java source-code level, it is fully valid at the bytecode level [40, 41]. As Allatori exemplifies, this enables the replacement of high-level loop constructs with goto-based control transfers, further altering recognizable structural patterns.

As these examples demonstrate, method bodies can be easily modified. Consequently, most static TPL detection tools rarely use method bodies as features and instead focus on other features, such as method signatures and dependency relationships.

String Encryption When reverse engineering an application, strings in the code can help identify relevant code snippets. For example, a reverse engineer can use strings displayed in the user interface or in error messages to search the code base and thereby identify the corresponding code locations. To prevent this, strings are encrypted in the delivered code and decrypted at runtime. As a result, decompiled applications contain only meaningless symbols instead of useful strings. Static TPL detection mechanisms that rely on string comparison between library code and application code become ineffective when the code is obfuscated using string encryption. However, because the decryption is performed at runtime, the decryption algorithm remains present in the code. This algorithm can be extracted and simulated, allowing the encrypted strings to be decrypted.

The encryption of strings in both Allatori and DashO relies on simple XOR operations. Each character in the encrypted string is XORed with a key, producing the corresponding decrypted character. A few subtle differences exist between the two tools. Allatori employs a fixed key in its basic decryption algorithm.DashO, in contrast, increments its key for each character, demonstrated in Listing 2. In Allatori's advanced decryption algorithm, the key is derived from the class name and method name of the caller of the decryption method. This is achieved by throwing a runtime exception and interpreting the stack trace to obtain the caller at runtime. In addition, DashO attempts to obfuscate calls to the decryption method by masking them using arithmetic operations, as illustrated in Listing 3.

```java
public static String a(int key, String s) {
char[] chars = s.toCharArray();
int len = chars.length;
int i = 0;

while (i < len) {
    char c = chars[i];
    int mask = key & 0x5f;
    c = (char) (c ^ mask);
    key = key + 1;
    chars[i] = c;
        i = i + 1;
}
return String.valueOf(chars, 0, len);
}
```

Listing 2: String decryption algorithm (Code generated by DashO)

```java
java.util.Random rnd = new java.util.Random();
int limit = 0x4b;
int v0 = rnd.nextInt(limit);    // nextInt(75) -> 0..74
v0 = v0 + 1;                    // range of the random number is 1..75

int v1 = v0 * 5;
v1 = v1 % v0;                   // v1 is always 0

String s;
if (v1 == 0) {
    s = "\u001b?40&60u\"2 -{";    // s is the encrypted string
} else {                       // else branch is never executed
    s = a.a(0x57, "bk=bke>nrx$&'iqqpqd(sx\u007fc{6c`ff0`1aj9");
}

s = a.a(-0x12, s);             // key for decryption is -0x12 (decimal: -18)

System.out.println(s);         // s can now be used as in the original code

```

Listing 3: Obfuscated call to the string decryption method (Code generated by DashO)

Optimization Both DashO and Allatori perform code optimization. Allatori uses its own implementation for identifier renaming, package flattening, and dead code removal. DashO relies on R8 for these tasks. Details regarding these optimizations are provided in 3.2.2.

Other techniques Allatori allows multiple smaller modifications. They may reduce assumptions about features that remain unchanged by obfuscation and can be used for comparing library code with application code.

- **Reordering of Members**. Fields and methods are shuffled so that the order within classes changes.
- **Adding the `final` Keyword**. Classes without subtypes can be supplemented with the final keyword. This prevents TPL detection approaches from using this keyword as an immutable feature for class comparison.
- **Synthesizing Fields and Methods**. Fields and methods can be marked as synthetic. As described in A.2.1, synthetic constructs are normally generated by the compiler. Marking members as synthetic can cause decompilers to omit such classes. Another implication is that, when comparing method headers in static TPL detection, the synthetic keyword must be filtered out.
- **Changing Access Modifiers to Public**. Access modifiers of fields and methods can be changed to public. This does not break accesses but prevents access modifiers from being used as immutable features.

DashO provides the following additional features. Other techniques exist, but they are not available in Android mode.

- **Resource Encryption**. Resource files are encrypted, and code is injected so that they are decrypted at runtime.

- **Checks**. Code is injected to verify at runtime whether the app is being debugged, running in an emulator, is hooked, or running on a rooted device. An action can be configured to occur when such an event is detected. For example, a method in the app can be triggered, or the app can exit immediately.

3.2.2 R8 Shrinker

The primary goal of R8 is code shrinking rather than code obfuscation. Nevertheless, the techniques it employs significantly affect TPL detection. R8 modifies the structure of the code, which complicates structural comparisons, although it does not alter program behavior or control flow. The resulting optimizations promise reduced application size, leading to faster download and installation times, as well as improved startup and rendering performance during execution [42].

Dead Code Removal R8 refers to this step as Tree Shaking. It removes unused code both from the application codebase and from included libraries. The objective is to reduce the overall application size. In TPL detection, this complicates the direct mapping between candidate library classes and application classes, since some library classes or class members may no longer be present in the optimized application.

Method Inlining Calls to small methods or methods invoked only a few times are replaced by the method body itself. This reduces the total number of methods and limits the usefulness of class comparisons based on method signatures.

Class Merging R8 attempts to merge classes horizontally and vertically. Siblings in class hierarchies (classes sharing the same supertype) and parent/child classes are considered candidates for merging. Based on several policies, R8 determines whether classes may be merged without altering program behavior or violating the Java specification. Details are provided in 4.2.1

Identifier Renaming R8 refers to this step as code minification. It replaces package, class, and class member names with shorter, meaningless identifiers to reduce the code footprint. This prevents direct mapping of packages, classes, and class members between library code and the optimized application. Regarding package renaming, R8 can flatten the package hierarchy and repack packages. Further details are provided in 3.3.

Resource Shrinking Finally, unused resources are removed. Although this step is less relevant for structural code comparison, since resources are not the primary focus of TPL detection, it may still influence the structure of the resulting APK.

3.3 Package Renaming

Identifier renaming is a core feature of an optimizer and obfuscator, resulting in less meaningful code and smaller applications. Renaming classes, methods, and fields requires careful handling to ensure that not only the definitions but also every usage is updated correctly. In addition, special attention is required when renaming code that uses reflection.

However, renaming classes, methods, and fields does not alter the overall code structure. In contrast, renaming packages enables an optimizer to flatten or repack them, effectively moving classes within the package hierarchy. The impact of this process and how ProGuard and R8 handle it is illustrated in the following example [38].

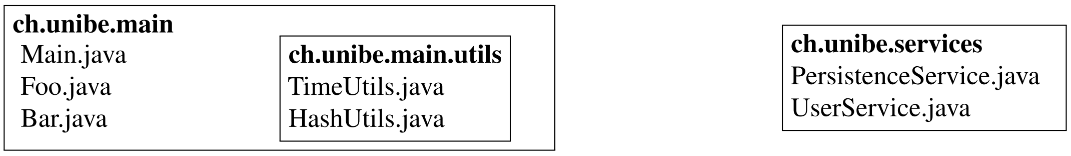


Figure 3.1: Packages and classes before renaming.

Consider a simple application structure with three packages, shown in Figure 3.1. When renaming classes and package names, classes preserved by the configuration (e.g., via the `-keep` directive) and their containing packages cannot be renamed. In our example, the class `main` located in the package `ch.unibe.main` represents such a protected class. This also affects the package `ch.unibe.main.utils`: since "*.utils" is nested in `ch.unibe.main`, only the `utils` portion can be renamed. Obfuscating the names results in the structure shown in Figure 3.2.

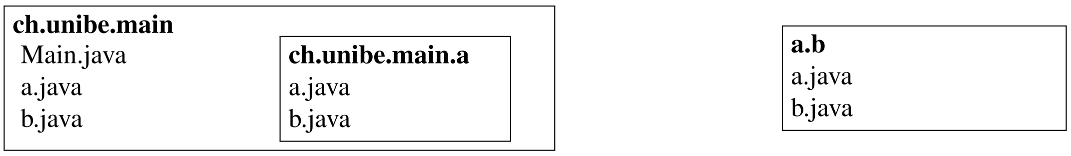


Figure 3.2: Package and class names are obfuscated. The main class and its package can't be renamed.

To further obfuscate the package structure, the option `--flattenpackagehierarchy` can be used. Every package in which all classes are allowed to be renamed is moved into the provided package. This produces a flat hierarchy while every package still including exactly the same classes, as shown in Figure 3.3.

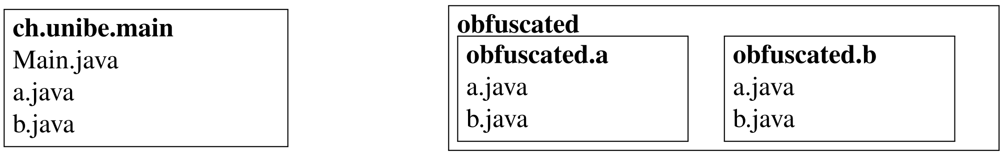


Figure 3.3: The `-flattenpackagehierarchy 'obfuscated'` option moves all packages into a common parent package, effectively flattening the package hierarchy.

Alternatively, classes can be repacked using the `-repackageclasses` option. The classes are moved into a single package, as illustrated in Figure 3.4.

Until now, classes that share a package with a class that cannot be renamed could not be included in package repacking. With the `-allowaccessmodification` option, this restriction no longer applies. However, using this option is not always desirable; for example, when obfuscating a library, non-public members would become public. Figure 3.5 illustrates the situation when package repacking is applied to its maximum.

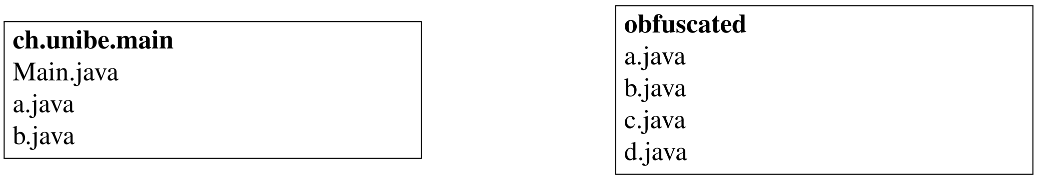


Figure 3.4: The `-repackageclasses 'obfuscated'` option moves all classes from packages that can be renamed into a single package.

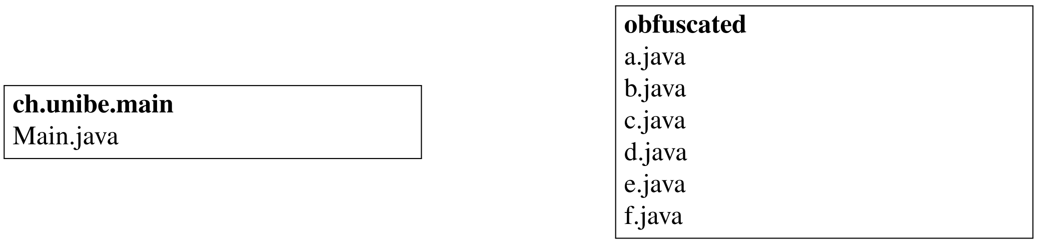


Figure 3.5: The `-repackageclasses 'obfuscated'` option can be combined with `-allowaccessmodification` to achieve the maximum possible degree of package repacking.

4 Approach

This chapter serves to support the understanding of the overall concept of the improved runtime TPL detection mechanism. We restate the research gap and the goal of this work. To understand how such a tool can be built it is essential to examine the tools involved in extracting and detecting dynamically loaded libraries. These include obfuscation tools, runtime extraction tools and existing TPL detection tools. In this chapter, we present an in depth analysis of one representative tool from each of these categories. Finally, we describe the resulting system to extract DEX files and the proposed improvements to the detection accuracy and runtime performance of the state-of-the-art TPL Detection tool.

4.1 Goal and Scope

Static analysis, as performed by current state-of-the-art approaches, uses APK files as input. The tools analyze the included DEX files to detect library code. However, some applications use dynamic feature modules to extend their functionality at runtime. When relying solely on static analysis, dynamically loaded code remains unanalyzed. When libraries are loaded dynamically, they will never be recognized.

To address this limitation, we investigate whether it is possible to extract dynamically loaded DEX files during runtime. For this purpose, we employ frida-dexdump to extract DEX files and propose a Frida-based hook for detecting dynamic code loading, enabling automatic extraction whenever new content is loaded during execution.

This feasibility study is limited to applications that do not employ protection mechanisms such as anti-debugging or anti-hooking techniques. Consequently, we do not target applications that intentionally obscure their dynamic code loading behavior. We focus on cases in which dynamic loading is implemented as intended by Android's Dynamic Feature Module capabilities.

The Android system is undergoing rapid and continuous development. Accordingly, the mechanisms for loading features, as well as the techniques for detecting such loading, are expected to evolve over time. For this reason, we invested only a limited amount of effort in analyzing robust detection metrics and instead focused on designing a configurable architecture.

Gu et al. [5] compared several state-of-the-art tools using a new benchmark and showed that existing approaches are not yet able to detect TPLs with high certainty. In addition, the evaluated tools exhibit high runtime overhead and do not provide results within a short time frame. We want to understand a state-of-the-art approach in detail in order to optimize the approach with respect to both runtime and detection performance. We selected LibPecker as the baseline, as it achieves the highest F1 score among the evaluated tools. However, the corresponding paper has not yet been peer-reviewed, and the benchmark has not been published. This prevents a direct comparison of our improvements with the reported benchmark results.

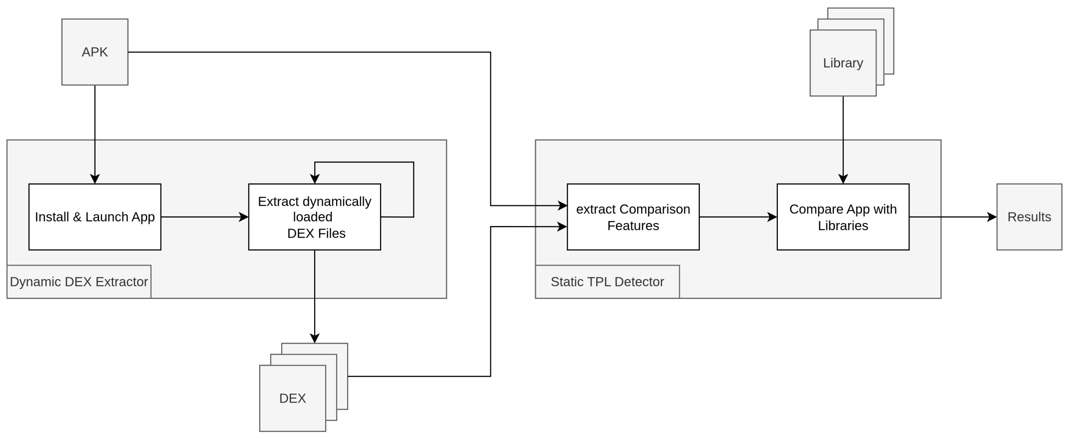

APK
Library
Install & Launch App
Extract dynamically loaded DEX Files
Dynamic DEX Extractor
extract Comparison Features
Compare App with Libraries
Static TPL Detector
Results
DEX

Figure 4.1: Overview of the hybrid third-party library detection tool architecture

The goal is to extract DEX files that are not available through static analysis and subsequently apply existing static TPL detection tools to them. This enables a more complete view of the libraries used within an application. Furthermore, we aim to use the improved understanding of obfuscation techniques and the internals of LibPecker to enhance detection accuracy by analyzing their impact and implementing appropriate adaptations. In addition, we aim to demonstrate how runtime performance can be improved to increase the scalability of the tool. For this purpose, we propose a hybrid approach, consisting of a dynamic DEX extractor component and a static TPL detection tool, as shown in Figure 4.1. The output of the DEX extractor can then be analyzed by the static TPL detection tool to identify and evaluate the libraries it contains.

4.2 Existing Algorithms and Tools

We provide code analyzes of the main components that influence or support the goal of a runtime TPL detection tool. While various obfuscation techniques, such as package renaming, are described in the Background chapter 3, we focus on the poorly documented class merging technique, which has a significant impact on the structure of an obfuscated application. We analyze the source code of R8 to understand its class merging capabilities. Next, we examine an existing tool and its approach to detecting DEX files in device memory. An appropriate detection mechanism is crucial for the complete extraction of dynamically loaded code. Finally, we analyze LibPecker to identify its limitations and potential avenues for improvement.

4.2.1 Class Merging

The shrinking tool R8 is part of Android Studio and is therefore widely used. One of the key optimization strategies employed by R8 is class merging. While most techniques of obfuscation and shrinking tools are well understood and documented, class merging remains poorly documented. Neither R8 nor its predecessor ProGuard explicitly describes in their documentation the conditions under which class merging occurs. ProGuard states in its interface description that merging is applied "whenever possible" when optimization is enabled [38]. Class merging has a significant impact on the structure of the code, as it can reduce the number of classes, increase the number of methods within a class through basic merging, or relocate classes by merging classes from different packages. In addition to the Shrinker R8, the Dexer D8 uses the same code base. D8 is responsible for generating DEX files from bytecode. Understanding D8 is crucial, as it is always executed when an Android application is built, regardless of whether code shrinking is enabled. The following analysis is based on the source code of R8 [43].

During the process of class merging, two or more classes are combined, resulting in a reduced number of classes within the application code. This reduction complicates the task of mapping library classes to application classes, as it may no longer be possible to identify an unambiguous mapping for every library class. Class merging is be performed according to two strategies, vertical and horizontal merging. *Vertical* class merging refers to the process of merging a class A into a class B, where A is the supertype of B. Unless stated otherwise, A or B (or both) may also be interfaces. To be a merge candidate, A must have exactly one subtype B. In this case, A and B form a merge group, with A as the source and B as the target. To enable parallelization, the merge analysis is performed on disjoint hierarchy structures.

The *horizontal* merging of classes proceeds in two phases. In the first phase (Single class Policies), each class is evaluated to determine whether it satisfies the required criteria. Classes that do not qualify for merging are filtered out. The second phase (Multi class policies) determines with which other classes a given class can be merged. After applying the single class policies, the remaining classes and interfaces are divided into two groups: one consisting only of interfaces and one consisting only of classes. The multi class policies take then a group of classes as input and produce a set of subgroups as output. Each policy in this second phase defines rules specifying with which other classes a given class cannot be merged. Each group is then split into subgroups such that every subgroup satisfies the rule defined by the policy. For example, the rule "Same Parent Class" ensures that every class within a subgroup shares the same superclass. Subgroups that contain only a single class (trivial groups) are removed. Consequently, in this second phase, classes and interfaces for which no compatible merging partners exist are eliminated. Finally, the classes in each remaining group are merged.

The Dexer D8 has limited class-merging capabilities. Its strategy is more restrictive and therefore requires handling fewer complex edge cases. The policy `Only Classes With Static Definitions And No Class Initializer` limits D8 to classes that contain only static members and have no class initializer. This restriction significantly reduces the number of classes eligible for merging. Additional policies that apply only when using D8 are `Same Partial Sub Compilation`, `No Api Outline With Non Api Outline`, and `Same Package For Non Global Merge Synthetic`. R8 does not impose this limitation and therefore includes additional policies to handle cases that could lead to changes in code behavior, which cannot occur in the merges allowed by D8.

When the optimization or shrinking options are disabled, class merging is always limited to synthetic classes. Only R8 provides those configuration options. Consequently, D8's merging behavior is always restricted to synthetic items. As a result, the `CheckSyntheticClasses` policy filters out all non-synthetic classes, ensuring that only synthetic classes are merged.

A complete overview of every policy, including a description of how it filters classes, is provided in the appendix A. To summarize, Table A.1 in the appendix shows the policies used by R8. For each policy, the required conditions for activation are also documented. Classes that are used only in R8 and not in D8 are marked accordingly. Similarly, Table A.2 shows the policies used by D8. To highlight the differences between R8 and D8 in terms of class merging capabilities, the policies that are used exclusively by one of the tools are presented in Table A.3.

4.2.2 DEX File Extraction

This section investigates the assumptions made by the open source tool `frida-dexdump` [44] when extracting DEX files from memory at runtime by analyzing its source code. The tool scans memory regions for the DEX file header and, upon identifying a valid header, reconstructs and dumps the corresponding DEX image to disk. To support this analysis, we also examine the DEX file structure as defined in the Android runtime source code. First, we describe the relevant properties of DEX files. Then, we explain the assumptions that `frida-dexdump` uses to locate DEX files in memory.

A DEX file is a binary file whose definition is implemented in the Android Open Source Project as a C++ class in a header file[Code]. The DEX file representation consists of a header and a data section [45] as shown in figure 4.2. The header usually begins with the magic number `0x64 0x65 0x78 0x0a 0x30 ?? ?? 0x00`, representing the string `dex\n` followed by the version number and terminated by a null byte. The current version is 040. However, Android 16, the most recent Android version, includes experimental support for version 041. This newer version introduces two additional fields in the header (Container Size and Header Offset), making the header 8 bytes larger.

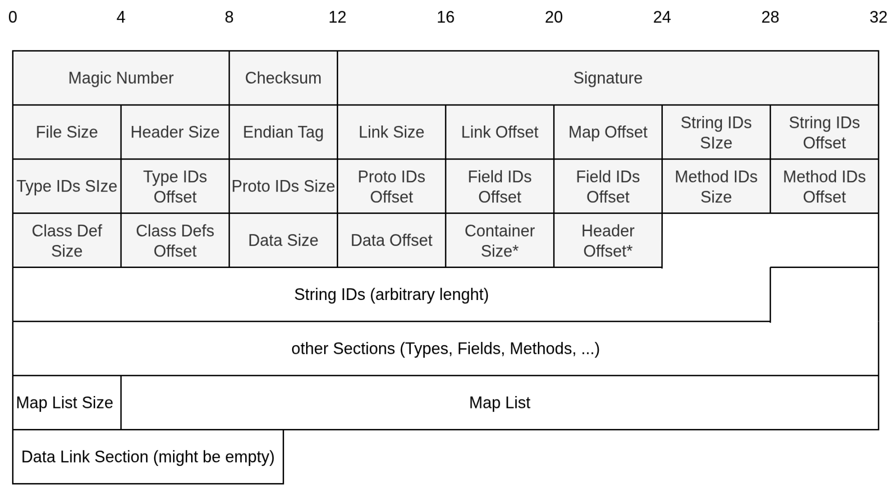


Figure 4.2: DEX File Header and Data Section. Fields in the header (shaded in grey) have fixed lengths, while most fields in the data section are of variable length. The header fields `container_size` and `header_offset` are present only in DEX file version 41.

To detect a DEX file in memory, the simplest approach is to search for the magic number. Frida-Dexdump enumerates every readable memory range and searches for this sequence. When the sequence is found, it performs a simple validation to avoid incorrectly interpreting memory sections as DEX files that contain the magic number only by coincidence. The validation ensures two conditions: First, the header must fit entirely within the memory range. If the match occurs too close to the range's end such that the header would extend beyond it, the range is ignored. Second, it checks whether the header has the expected size of 112 Bytes (Version 040). One method would be to check whether the value 0x70 (112) is stored in the `header_size` field at offset 0x24 (36). However, this field can be easily altered by an obfuscator to conceal a DEX file. To increase robustness, Frida-Dexdump instead checks the field `string_ids_off` at offset 0x3C (60). This field contains the offset to the first section after the header and, since offsets are relative to the file start, this value corresponds to the header size.

There are two additional strategies for finding DEX files in memory in cases where the magic number is obfuscated. The simpler approach applies only when the DEX file starts at the beginning of the memory range. The idea is to compare two redundant pieces of information in the DEX file format. When these values match, there is a high probability that the structure corresponds to a DEX file. The data section includes a `map list`, a datastructure describing the entire contents of the file. Each item in this list contains a type ID and a pointer to the corresponding section within the DEX file (`type` and `offset` in Figure 4.3). The type ID of the map list section is 0x1000 (4096), and this entry is always present. Frida-Dexdump iterates through this list until it finds the corresponding item. When the value in the `offset` field in the map-list item matches the `map_offset` field in the header, the structure is likely a DEX file.

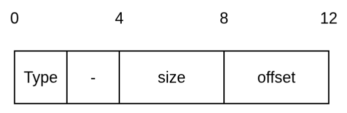


Figure 4.3: Structure of a map list item. The type identifier of the map list entry is 0x1000 (4096).

The other strategy can be enabled by setting the option `deepsearch` when running Frida-Dexdump. Instead of searching for the magic number, it searches memory for the bytes `0x70 0x00 0x00 0x00`. The header fields are stored in little-endian format; therefore, this value corresponds to 0x70 (112), which is the length of the header. This value is expected in the `string_ids_off` field. Since this field is located at offset 0x3C (60) from the beginning of the header, the start of the DEX file is assumed to be 0x3C bytes before the match. Based on this assumed start address, the tool performs the same map-list verification used in the simplified detection method when deepsearch mode is not enabled. When this verification passes, the tool examines the various offset fields in the header. For the string, type, proto, field, and method offset fields, it checks whether they point to addresses located after the header and before the end of the DEX file. When this check also succeeds, the structure is considered a DEX file.

Since the detection of DEX files relies heavily on the header length, the introduction of DEX file version 41 [Code] may have a significant impact. To ensure reliable detection, the increased header size must be taken into account in future implementations.

4.2.3 LibPecker Internals

We describe the overall principle of LibPecker and its main feature for comparing application code with library code. The analysis is divided into subsections, roughly corresponding to the phases in the code. For the analysis, we used both the corresponding paper and the source code.

Given an Android app in the form of an APK and a library, LibPecker calculates a score between 0 and 1 indicating whether the library might be included in the app. In addition to the app and the library, LibPecker uses the Android SDK to distinguish between application code and the Android framework. Given these prerequisites, LibPecker analyzes the app and the library by extracting features and comparing them. The following in-depth description focuses on aspects that are central to understanding the algorithm, areas with potential for improvement, and limitations of LibPecker.

Prefiltering. First, LibPecker attempts to terminate early if the library does not match the application at all. For this purpose, a simple class signature (similar to figure 4.5) is constructed for each class A. This signature consists of the access flags of class A and information about its superclass B: the class name of B if B is a system class, and an indicator of whether B belongs to the same package as A. The array dimension of a superclass is always zero; therefore, this field is not relevant in this context .

LibPecker checks, for every library class, whether a corresponding application class with an identical basic class signature exists. If fewer than 50% of the library classes are matched, the library is discarded, and the ratio of matched library classes is reported as the upper bound similarity score.

Package Comparison. For subsequent comparisons, LibPecker applies weights. Larger classes, defined as classes with more methods and more dependencies, are assigned higher weights. The rationale is that small classes can be matched more easily, whereas larger classes contribute more functionality and are more difficult to match.

LibPecker compares each library package with each application package. To calculate the similarity between two packages, the following process is performed. The similarity of two packages depends on the similarity of their classes. This similarity is computed by first mapping library classes to application classes. Starting with the library class with the highest weight, LibPecker searches for the application class with the highest similarity. The matched classes are then removed from consideration, and the process continues with the remaining library class with the highest weight. The resulting mappings and their associated similarity scores are stored and used for the calculation of the package similarity. For each mapping, the similarity score is multiplied by the weight of the corresponding library class. These weighted similarities are summed and then divided by the sum of all weights, resulting in the final similarity score between the two packages. When a library package and an application package achieve a similarity above a defined threshold, the application package is stored as a candidate for the given library package.

Class Comparison. We now describe how classes are compared during the package comparison step. Classes are reduced to signatures for comparison. The selected features are designed to be resilient to obfuscation and to depend on class dependencies. We divide the class signature into subcomponents to illustrate its construction.

The **Class Dependency Signature** (Figure 4.4) reflects dependency relationships. It encodes whether a dependency is a system class or belongs to the same package as the class under investigation. In addition, the array dimension is recorded, as well as the package name in the case of a system class.

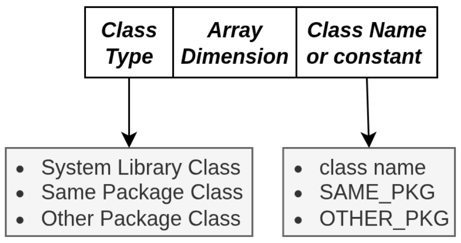


Figure 4.4: Class Dependency Signature (CDS)

The **Basic Class Signature** (Figure 4.5) consists of the access flags of the class and the Class Dependency Signature. The dependency used in this context is the superclass of the class. In addition, the number of implemented interfaces, as well as the Dependency Signatures of these interfaces, are included. Since some access flags are subject to obfuscation, only a subset of access flags is considered.

A **Field Signature** (Figure 4.6) contains the access flags of the field, the Dependency Signature of the field type, and the field value. When the field value is uninitialized, a corresponding constant value is used to indicate this; otherwise, the actual value is used.

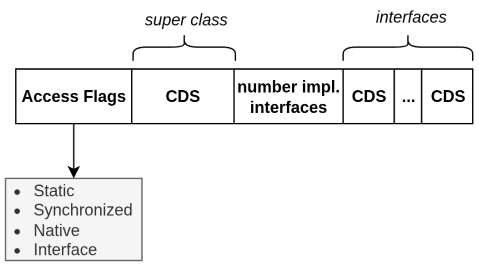


Figure 4.5: Basic Class Signature. For prefiltering, a simplified version of this basic class signature exists, which does not include the interfaces.

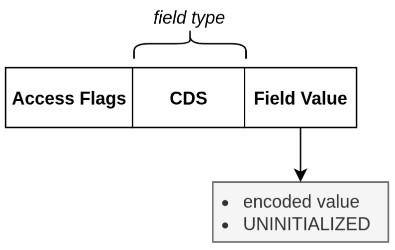


Figure 4.6: Field Signature

The **Method Signatures** (Figure 4.7) are constructed similarly. In addition to the access flags, they include the Dependency Signatures of the return type, the parameter types, and the exception types.

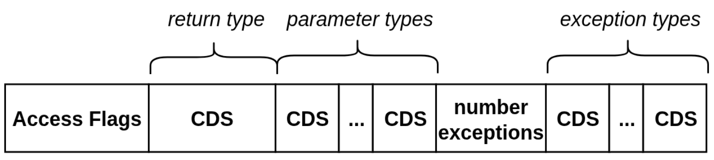


Figure 4.7: Method Signature

Finally, the **Class Signature** (Figure 4.8) is composed of these components. It consists of the Basic Class Signature, followed by the Field Signatures and the Method Signatures.

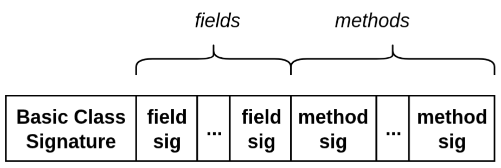


Figure 4.8: Class Signature

When comparing a library class A and an application class B, the class signatures are compared. If the signatures are identical, the classes are considered equal, with a similarity score of 1. Conversely, if the Basic Class Signatures of the two classes are not equal, they are considered non-equal, with a similarity score of 0. In all other cases, the Jaccard similarity is used to calculate the class similarity. This is done by counting the number of methods and fields in class A that share a method or field signature with class B, and dividing this count by the total number of methods and fields in the library class (Eq. 4.2).

$$\text{rate} = \min\left(1, \frac{|\text{fields and methods in library class}|}{|\text{fields and methods in app class}|}\right) \tag{4.1}$$

$$\text{classSim} = \text{rate} \times \frac{|\text{identical signatures}|}{|\text{methods and fields in app class}|} \tag{4.2}$$

To normalize comparisons when a library class is much smaller than the corresponding application class, the rate is multiplied with the similarity score (Eq. 4.1). This prevents biased or unfair comparisons. For example, consider an application class A with four methods and a library class B with only two methods. If both method signatures match, the resulting similarity score is 0.5. However, when comparing the same application class A with a library class C that also contains four methods, two matching method signatures likewise result in a similarity score of 0.5. When applying the rate, the comparison between A and B yields a similarity score of $\frac{2}{4} \times 0.5$, whereas the comparison between A and C yields a similarity score of $\frac{4}{4} \times 0.5$. This rate penalizes therefore matches in smaller classes.

Package Linking. After the package comparison step, we obtain, for each library package, a set of application package candidates, represented as `<libraryPackage, Set<AppPackage>>`. To reduce the number of package candidates, two simple rules are applied, both based on package structure.

The first rule checks whether, for a library package, there exists an application package candidate with exactly the same package name. If such a candidate exists, all other candidates for this library package are removed.

In the second rule, the reduced set of package candidates is examined to extract the package root. Consider a scenario in which, based on the previous rule, a library package perfectly matches an application package with the package name `ch.unibe.tpld`. When mapping another library class with the same package root `ch.unibe`, only application packages with the corresponding root `ch.unibe` remain valid candidates. Consequently, the library package `ch.unibe.utils` may be matched to `ch.unibe.a`, but not to `ch.unifr.a`. In other words, this rule ensures that only application packages sharing the same package-name root are retained as candidates.

To prevent LibPecker from spending excessive time comparing an application with a library that is not included, LibPecker evaluates the maximum possible similarity score that can be achieved with the given candidates. To calculate this, for each library package, the application package candidate with the highest similarity score is selected. This score is multiplied by the weight W_a of the corresponding library package a. The sum of these weighted maximum similarity scores represents the best theoretically possible matching across all library packages, while ignoring effects such as duplicated matches (Eq. 4.3).

$$\text{maxMatchedWeight} = \sum_{a \in \text{libPkgs}} W_a \times \max_{b \in \text{appPkgs}} \text{sim}(a, b) \tag{4.3}$$

$$\text{totalWeight} = \sum_{a \in \text{libPkgs}} W_a \tag{4.4}$$

The maximum total weight is defined as the sum of all weights and corresponds to a hypothetical scenario in which, for each library package, an application package with a similarity score of 1 exists (Eq. 4.4). To obtain the maximal possible similarity, the calculated maximum matched weight is divided by the total weight (Eq. 4.5).

$$\text{maxSimilarity} = \frac{\text{maxMatchedWeight}}{\text{totalWeight}} \tag{4.5}$$

If this upper bound similarity falls below a predefined threshold, the library is considered not to be included, and the upper bound similarity is reported as the final similarity score. If the estimated similarity exceeds the threshold, LibPecker proceeds with the enumeration of the partitions.

Partition Enumeration. To determine the maximum overall library similarity score, LibPecker selects one application package from the set of package candidates for each library package. To this end, all valid combinations of such selections are enumerated. In the case of library packages that have only a single candidate application package, this selection is trivial. However, the number of possible combinations can be large.

To avoid evaluating every possible combination, heuristics are applied to discard mappings that are implausible based on already established package mappings. For example, if a library package `ch.unibe.tpld` has already been mapped to an application package `a.b.c`, then mapping another library package `ch.unibe.*` to an application package `a.z.*` is inconsistent and therefore disallowed.

Consider two library packages lp_1 and lp_2 and their corresponding mapped application packages ap_1 and ap_2. LibPecker considers a mapping invalid in each of the following cases:

- lp_1 is in a parent package of lp_2, but ap_1 is not in a parent package of ap_2
- lp_1 is in a child package of lp_2, but ap_1 is not in a child package of ap_2
- lp_1 and lp_2 are in the same package, but ap_1 and ap_2 are not

If the packages pass these checks, the longest common parent package lp_{common} of lp_1 and lp_2, as well as ap_{common} of ap_1 and ap_2, is extracted. Subsequently, the distance Δ to this common parent package is calculated for each package. The distance is defined as the number of package hierarchy levels between two packages. For example, the distance between the packages `ch.unibe` and `ch.unibe.tpld.services` is two.

The condition in equation 4.6 must hold:

$$\Delta(lp_{\text{common}}, lp_1) = \Delta(ap_{\text{common}}, ap_1) \ \wedge \ \Delta(lp_{\text{common}}, lp_2) = \Delta(ap_{\text{common}}, ap_2) \tag{4.6}$$

After evaluating all partitions, the one achieving the highest overall similarity score is selected. This score is returned as the final similarity score of the library and indicates how likely the library is included in the application.

4.3 Findings from LibPecker Analysis

The analysis of LibPecker helped to identify the limitations of the tool. First, we present the findings regarding the impact of package renaming, as LibPecker depends heavily on the package structure. Next, the impact of other obfuscation techniques is summarized. Finally, the observed runtime bottlenecks are discussed.

Several components of LibPecker's detection depend on the package structure. Based on the analysis of obfuscation techniques in Section 3.3, we distinguish two levels of package renaming: *package flattening* and *package repacking*. Using the analysis of LibPecker's mechanisms, we assess the theoretical impact of these obfuscation techniques on its accuracy.

Package Comparison. LibPecker attempts to match packages by comparing the classes of a library package A_{lib} with the classes of each application package B_{app} to compute a similarity score between A_{lib} and B_{app}. In the case of package flattening, the similarity remains unchanged, because the classes remain grouped within the same package. However, when *package repacking* occurs, the original package structure is lost. The application then typically consists of several small packages and one large package containing most of the classes. Consequently, no corresponding application packages exist for the library packages anymore, and the original package-level similarities therefore no longer hold. Moreover, the class type field in the Class Dependency Signature (see Fig. 4.4) changes when classes are moved to a different package after obfuscation. For example, in the unobfuscated version, the superclass B of class A resides in a different package, resulting in a Class Dependency Signature with an `Other Package` flag. After package repacking, classes A and B may be placed in the same package, producing a dependency signature with a `Same Package` flag. This change can lead to a false negative.

Package Linking. During package linking, package candidates are reduced using heuristics based on the package structure. Renaming packages significantly affects this step. If all packages are renamed such that no library and application packages share the same name, the reduction rules cannot be applied, and no candidates are removed. This negatively impacts performance, because reducing candidate packages accelerates the enumeration of partitions. However, the accuracy of the result is not affected.

We now consider a scenario in Table 4.1 where some packages remain unobfuscated (e.g., due to `-keep` rules), while others are renamed and flattened. Packages listed in the same row correspond to each other. Applying the first reduction rule, the candidate packages for `ch.unibe.tpld` are reduced to the application package with the same name. For other library packages sharing the root `ch.unibe`, the second rule removes candidate packages with a different root. In our scenario, the application packages renamed to `a` and `b` are removed from the candidate list because they do not share the same package-root name, even though they would constitute the correct matches.

Library Package	**Application Package**	**Notes**
`ch.unibe.tpld`	`ch.unibe.tpld`	package not obfuscated
`ch.unibe.utils`	`a`	renamed/flattened, contains same classes as before
`ch.unibe.api`	`b`	renamed/flattened, contains same classes as before

Table 4.1: Example scenario: package flattening

When *package repacking* is applied, the scenario described in Table 4.2 is possible. Again, the candidate packages for `ch.unibe.tpld` are reduced to the application package with the same name. The application package `a`, which contains most of the classes, is no longer a candidate for any library package. However, it is debatable whether matching one of the unobfuscated library packages with `a` would yield a meaningful result.

Library Package	**Application Package**	**Notes**
`ch.unibe.tpld`	`ch.unibe.tpld`	package not obfuscated
`ch.unibe.utils`	–	package repacked into `a`
`ch.unibe.api`	–	package repacked into `a`
–	`a`	package contains most classes from multiple packages

Table 4.2: Example scenario: package repacking.

Partition Enumeration. The rules that exclude infeasible matchings strongly depend on the package hierarchy. At this step, LibPecker is aware of renaming but assumes that the hierarchy itself is not altered. LibPecker enforces that the relative distance between packages in the library and the application is identical.

Consider again the example described in Table 4.1. Suppose the library package `ch.unibe.tpld` (lp_1) and the corresponding application package `ch.unibe.tpld` (ap_1) are already matched. Now, we want to match the library package `ch.unibe.utils` (lp_2) with the application package `a` (ap_2).

However, this matching is prohibited by the rules defined in LibPecker. While the library packages lp_1 and lp_2 are siblings in the package hierarchy, the application packages ap_1 and ap_2 are not. Consequently, in no enumeration will the matching

$$\texttt{ch.unibe.tpld} \rightarrow \texttt{ch.unibe.tpld} \quad \text{and} \quad \texttt{ch.unibe.utils} \rightarrow \texttt{a}$$

exist. Therefore, package flattening hinders LibPecker's ability to match packages accurately.

Besides package renaming, LibPecker must also handle other obfuscation techniques. Since LibPecker focuses on dependency relationships and fuzzy signature headers, method bodies are not considered during comparison. Consequently, obfuscation techniques that modify the code itself, such as control-flow randomization, do not affect the accuracy of the approach.

LibPecker uses a limited set of access flags as obfuscation-resilient features; however, this resilience is not absolute. Notably, LibPecker excludes the access modifiers `public`, `protected`, and `private`, as these flags may be altered by R8 when the `allowaccessmodification` option is enabled. Consequently, the only access modifiers considered by LibPecker are `static`, `synchronized`, `native`, and `interface`. Nevertheless, Allatori allows methods and fields to be marked as synthetic, thereby effectively modifying a feature that is intended to be obfuscation resilient.

LibPecker further assumes that classes remain within their original packages. Package renaming obfuscation has been discussed previously. Another obfuscation technique that violates this assumption is class merging. Under certain conditions, R8 allows classes from different packages to be merged horizontally (see the Respect Package Boundaries Policy in Section A.2.2). This process may change the package in which a class was originally defined and, as a result, alter the Class Type Field in the Class Dependency Signature (Figure 4.4).

Another violation of LibPecker's assumptions is that the number of interfaces implemented by a class can not change. This property is used in the Basic Class Signatures. When a class A implements two interfaces that are merged, or when the contents of an implemented interface are inlined into A, the number of interfaces implemented by A may be smaller in the obfuscated code.

These limitations may occur in practice; however, in contrast to package structure issues, they are less relevant and should be considered edge cases.

Besides detection accuracy, the scalability of an approach is an important factor for real-world use. We present the observed bottlenecks related to the runtime performance of LibPecker.

Caching App Analysis LibPecker is executed separately for each library under test. That is, for each execution of LibPecker, only the similarity between a single library and an application is calculated. For an application compared against a database of n libraries, LibPecker must be called n times. Since the code of the evaluated application has to be analyzed and the corresponding signatures must be calculated, the application code is analyzed n times. As a result, the application's signatures are recalculated repeatedly, leading to substantial and unnecessary runtime overhead.

Preprocessing Libraries In typical use cases, multiple applications are analyzed against the same library database. Recomputing library signatures for each application is therefore inefficient. However, LibPecker does not provide a mode to use preprocessed libraries. Consequently, a library must be processed each time an application is compared against it.

Slow Signature Comparison Beyond code analysis, the use of the IntelliJ Profiler helped to identify computational bottlenecks. The analysis revealed that class comparison is dominated by repeated calls to `ArrayList::contains`. In LibPecker's Class Comparison step, method signatures of an application class are stored in a mutable ArrayList sig_{app}. For each method signature of the corresponding library class, the algorithm searches for a matching entry in sig_{app} using repeated calls to `ArrayList::contains`. When a match is found, the signature is removed from the list. The same procedure is applied to field signatures. Since `ArrayList::contains` has a worst-case complexity of $\mathcal{O}(n)$, this approach incurs significant overhead compared to other data structures that are optimized for fast access, such as hash-based structures.

Effectiveness of Prefiltering The number of required comparisons is a key factor influencing the runtime of the tool. To address this, the prefiltering step reduces the set of libraries that are subjected to detailed analysis by excluding candidates that are unlikely to match. The threshold applied during prefiltering is a critical parameter, as it determines how many libraries are filtered out. Choosing this threshold involves a trade-off: more aggressive filtering reduces the number of comparisons but increases the risk of excluding libraries that are actually included, whereas less aggressive filtering preserves more candidates at the cost of additional comparisons. Fine-tuning this parameter is non-trivial, as adjustments may favor specific datasets rather than yielding generally applicable improvements. Therefore, the prefiltering step is essential for balancing computational efficiency with the reliable identification of relevant libraries.

4.4 System Design for Runtime Detection

We now describe the proposed system for detecting and extracting DEX files. Frida [46] serves as the foundation of the system since it is easy to integrate and provides powerful capabilities. In particular, it offers extensive support for method hooking and direct memory access. An alternative to Frida for analyzing the behavior of applications is to modify the Android Runtime (ART), as demonstrated by Ning and Zhang [12]. Another approach is to attach a debugger directly to the target application; for example, using ptrace allows attaching to a running process and inspecting or modifying its memory, as shown by Zheng et al. [27]. While both approaches are powerful, they are also considerably more complex to implement.

The process of analyzing an application involves initializing the setup and executing the target application as shown in Figure 4.9. When our tool is started, it first checks whether Frida is properly initialized on the device. If the Frida binary is not already present, it is downloaded and pushed onto the device. The tool then ensures that the target application is installed. Once the application is launched, an initial dump of the in-memory DEX files is performed. These DEX files are then disassembled into Smali files using the tool Baksmali [47]. Typically, the initial dumped DEX files correspond to those that can be extracted statically from the APK.

Frida is used to hook events that are triggered when dynamic code is loaded at runtime. During application usage, either through manual interaction or by using an automated tool such as Android Monkey, such events are eventually triggered and detected by our prototype. Upon detection, the tool performs another dump of the in-memory DEX files. The collected DEX files therefore exceed those obtainable through static analysis of the APK alone. These DEX files can subsequently be analyzed using a static TPL detection tool to identify the libraries they contain.

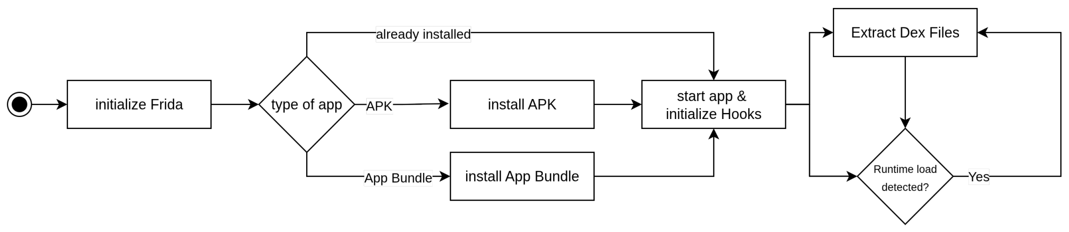


Figure 4.9: First, Frida is initialized and the application is started. Subsequently, the DEX files are extracted. Whenever the hook is triggered, the extraction process is executed again.

Instead of implementing a dynamic code loading detection mechanism that is difficult to extend to new requirements, we focus on an architecture that can be adapted to changes in how dynamic code may be loaded in the future or in how the Android system handles such behavior.

In this work, we describe a limited set of hooked methods that enable the collection of dynamically loaded code. However, we explicitly make no assumptions regarding the completeness or robustness of the selected hooks and leave their extension to future work. Furthermore, advanced techniques exist that allow Android applications to prevent analysis using Frida.

The mechanism used to detect dynamic code loading is based on class loaders. Class loaders operate at the Java level and are implemented within the Android libcore library. The `DexClassLoader`[Code] enables the loading of code that is not part of the base application. It is particularly useful for loading DEX files contained in `.jar` or `.apk` archives. Such situations occur, for example, in Android App Bundles, where each feature module is packaged as a separate APK.

Rather than directly hooking the usage of the class loader itself, we hook a side effect of its initialization. When a `DexClassLoader` instance is created, its superclass `BaseDexClassLoader` is invoked, which in turn creates a `DexPathList`[Code] object. This object contains references to the loaded DEX files. The method `makeDexElements` in the class `DexPathList` is responsible for loading the corresponding DEX files from the specified path. This path may refer to either an APK archive or a raw DEX file. Consequently, by hooking `makeDexElements`, we are therefore triggered whenever DEX files are loaded and can additionally observe the files from which they are loaded, which may provide further insights

A secondary method we hook is `makeInMemoryDexElements`. This method represents legacy code and is no longer used by `InMemoryDexClassLoader`. Nevertheless, it remains part of the code base because many applications rely on it and may still invoke it.

Other possibilities for detecting such events include hooking into the Android Runtime. This approach would make the detection more independent of the Java-level implementation of the application. Possible starting points are methods that directly interact with the DEX file representation or with OAT files, which contain ahead-of-time compiled assembly code as well as the original DEX file [48].

4.5 Proposed Improvements

Based on the findings regarding runtime bottlenecks, we propose improvements to increase the execution speed of LibPecker.

Caching App Analysis In the improved version, LibPecker is executed once per app. The locations of all libraries to be tested are provided as input arguments. During the initialization phase, the improved LibPecker analyzes the application a single time and reuses the resulting signatures for all library comparisons. Instead of terminating after each library comparison, LibPecker iterates over the provided libraries and keeps the calculated application signatures in memory. Consequently, the app does not need to be reanalyzed for each library.

Preprocessing Libraries We propose a preprocessing step in which all libraries are analyzed once in advance. The resulting signatures are stored in a separate binary file and reused during subsequent application library comparisons. This prevents libraries from being reanalyzed when they are compared against another application. Since the signatures are stored in a separate file, they persist across LibPecker restarts.

Hash Based Signature Comparison Since ArrayLists are not well suited for frequent read operations, we propose replacing them with a hash-based data structure. However, as the creation of hash-based structures is more expensive than that of lists, effective caching of this data structure must be considered so that it can be reused when comparing a corresponding class against another class. Without reuse across multiple comparisons, the advantage of faster read operations may be negated.

In the original implementation, the list is modified during the comparison step by removing already matched elements. Removing elements from the hash-based replacement is not an option, as this would conflict with the goal of caching the data structure. Instead, we propose a data structure in which each stored element is associated with a counter indicating how many times the element appears in the structure. When the original implementation would remove an element, the improved implementation decrements the corresponding counter instead. To enable reuse of the structure for a new comparison, the counter is reset. Therefore, the internal hash table does not need to be recalculated.

Prefiltering As prefiltering has a substantial impact on performance, we propose an alternative prefiltering mechanism. Depending on its effectiveness, it can be combined with the existing prefiltering approach. Ideally, this combination could enable the use of a stricter threshold for the existing prefiltering. The goal is to achieve faster execution by reducing the number of required comparisons.

The proposed heuristic prefiltering is based on Android API calls. While most method calls may be renamed during obfuscation, Android API calls cannot be renamed. We first collect the set of all API calls contained in the APK and the set of all API calls contained in the library. Extracting API calls from the code can be time consuming, as every line of code in the entire APK and all libraries must be analyzed. However, when applying the improvements proposed above, most of this processing is performed during the preprocessing phase or, in the case of the APK, only once per app comparison. After collecting the API calls, we verify how many API calls used in the library are also present in the APK. We cannot simply assume that every API call in the library appears in the APK, because only a portion of the library may be used, and dead code removal may eliminate the unused API calls. This reduces recall. Therefore, we use a threshold to require only a smaller matching ratio of API calls.

To improve the effectiveness of the prefilter, we want to combine the signature-based and API-call-based prefilters, leveraging the strengths of each. Our proposed approach for combining the prefilters is to adjust the threshold of one prefilter based on the result of the other. Our evaluation of the two

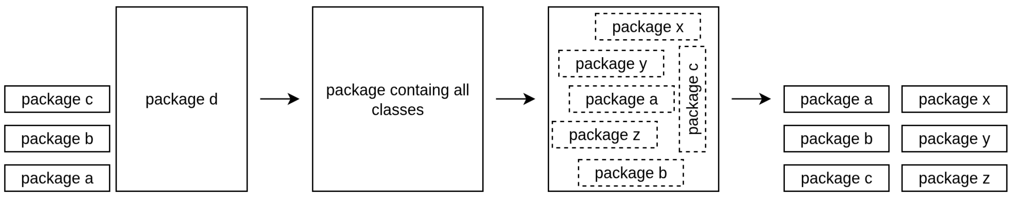


Figure 4.10: First, the application contains repackaged packages. Next, all classes are merged into a single large group. Using similarity measures, we then regroup classes that are likely to belong to the same original package. Finally, we obtain a package structure without a package hierarchy.

prefilters shows that the signature-based prefilter achieves very high recall but limited specificity, while the API-call-based prefilter achieves higher specificity at the cost of lower recall. To combine their strengths, we measure both the signature similarity and the ratio of API-call matches for each library. Libraries with high signature similarity are considered highly confident, so we allow a lower API-call-match rate in the second prefilter. Conversely, libraries with low signature similarity, indicating higher uncertainty, are subjected to a stricter API-call requirement, filtering out more candidates.

We discovered that LibPecker makes strong assumptions about the package structure. Our hypothesis is that these assumptions limit its accuracy when detecting libraries in obfuscated applications. To improve accuracy, we aim to relax the assumptions made by LibPecker. The main idea is to construct artificial packages in order to restore package structures. Specifically, we attempt to regroup classes that have been repackaged into a single class into packages resembling the original structure.

Primarily, we aim to address the scenario in which classes are repackaged into a single large package. In practice, a small number of residual packages may remain, containing classes that could not be repackaged. The main modification affects the Package Comparison step. In the following, we describe the core idea.

Artificial Package Structure We ignore the original application package structure entirely and instead form a single application package that contains all classes. Rather than comparing each library package with each application package, we compare each library package with this single, aggregated application package. During this comparison, we reconstruct a package structure as follows. In an arbitrary order, we select a library package and compare its classes with the remaining application classes. For each library class of this package, we select the application class with the highest similarity and remove this class from further consideration. The selected classes are grouped into a newly constructed package. This process is illustrated in Figure 4.10. If the correct classes were selected consistently, the resulting scenario would correspond to a flattened application in which the package hierarchy is lost, but classes belonging to the same original package are still grouped together. In practice, however, the outcome represents only an approximation of the original package structure.

An important observation in our setting is that very high similarity often behaves approximately transitively in practice. That is, if a class A exhibits very high similarity to a class B, and class B exhibits very high similarity to a class C, then class A will typically also show high similarity to class C. We attribute this behavior to the fact that pairs of highly similar classes share a large fraction of their structural elements, such as method or field signatures.

To illustrate this intuition, consider two classes A and B, each containing ten elements, that share nine elements. Similarly, assume that classes B and C also share nine elements. Under these conditions, classes A and C are guaranteed to share at least eight elements, resulting in a high similarity score as well. While this tendency is commonly observed when similarity values are close to the maximum, it does not generally hold for intermediate similarity levels and is not guaranteed for all similarity measures.

When comparing a library class A_{lib} and identifying multiple application classes B_{app} and C_{app} that both exhibit very high similarity, we find that the specific choice between these classes is not critical for subsequent comparisons. In practice, such classes tend to be structurally similar to each other and can be treated as interchangeable representatives. As a result, the order in which highly similar classes are selected has limited impact on the outcome, provided that a sufficiently high similarity threshold is enforced.

We deliberately merge all application packages into a single package, even when parts of the original package structure are still preserved. This is because we cannot assume that the remaining packages are complete. In particular, we must assume that many classes that originally belonged to one of the still existing packages have been relocated into the large repackaged package.

After the package comparison step, we obtain application packages that contain classes similar to those of the corresponding library packages, even in scenarios where the original application consisted of a single large repackaged package and only a few small residual packages.

We do not claim that the transitivity argument holds for every similarity measure, nor that the resulting matches are optimal. Rather, we treat it as an approximation that avoids computationally expensive exhaustive matching. We assume that this modification to the application processing improves the detection rate for applications that employ package repacking. Empirical evaluation is required to determine whether this approach has a significant negative impact on applications without package repacking and whether, consequently, an explicit package-repacking detection mechanism is necessary to enable the described modifications only when they are beneficial.

We introduce two variants to assess the impact of small, targeted modifications to our approach.

(I) Do not remove matched classes In the default configuration, when application classes are matched to classes from library packages, the corresponding application classes are removed from the single large package. In this variant, we construct the synthetic packages but retain the matched classes in the large package, allowing them to participate in subsequent matching steps. The intent is that incorrectly matched classes from earlier steps do not affect subsequent matching, since they are retained in the large package.

(II) Adapted CDS As described earlier, the CDS includes a flag indicating whether a class is a system class (`SYS_LIB_CLASS`), originates from the same package (`SAME_PCK_CLASS`), or belongs to a different package (`OTHER_PCK_CLASS`). This information depends on the package structure. When evaluating the above changes, we additionally consider a relaxed variant of this flag. In this variant, only system and non-system classes are distinguished by marking all non-system classes as `OTHER_PCK_CLASS`.

5 Implementation

This chapter presents the implementation details of our approach, focusing on both the runtime extraction of DEX files and the enhancements made to LibPecker. We first describe how applications are handled and how dynamically loaded DEX files are detected and extracted. Subsequently, we provide a detailed explanation of the improvements that were introduced to enhance performance and detection accuracy.

5.1 Dynamic DEX Extractor

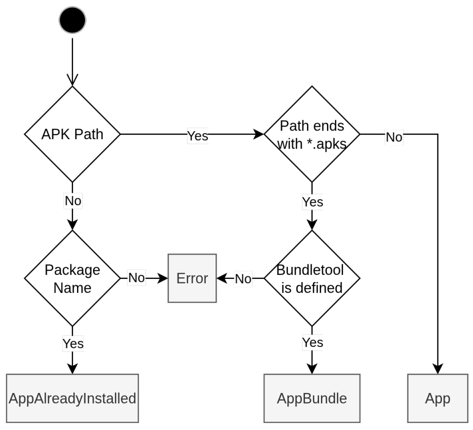
To control and analyze applications on a connected device, interactions are performed through the Android Debug Bridge. To avoid direct handling of low level ADB commands, a small abstraction library named `appHandler` is implemented. This library provides a unified interface for installing, starting, stopping, and uninstalling applications. In addition, it manages the initialization of a Frida instance on the device. Building on this abstraction, a generic mechanism is provided to execute different analyzes in a structured and repeatable manner. The entire codebase is implemented using TypeScript and Node.js.

App Factory Each application is represented by an instance of the `App` class. Three categories of applications are supported. The first category includes applications installed from APK files. The second category represents App Bundles installed from APKS files. The third category covers applications that are already present on the device and cannot be installed from the host system. This is particularly useful for analyzing applications obtained from official marketplaces such as the Google Play Store. Since each category requires different input parameters, a factory class is used to create the correct `App`-instance based on the provided configuration. The decision logic is illustrated in Figure 5.1a. Once the instance is created, it can be used to control the application lifecycle on the device.

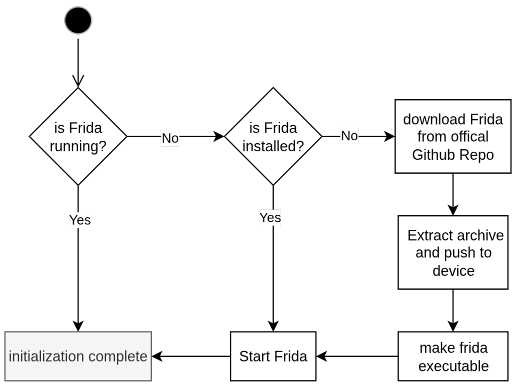
Interacting with Applications Application interaction is handled through a unified interface. Calling `App::start` initiates the application, while the instance itself determines the necessary preparation steps. This includes checking whether the application is already installed and performing installation if required. The application is then launched using an ADB command. Additional methods are provided to stop, uninstall, and query the runtime state of the application.

Frida Initialization The most common way to use Frida is in injected mode, where Frida attaches to a running process and takes control of its execution. In this setup, the instrumentation logic runs inside the target application, while a Frida instance on the host computer communicates with the Frida server running on the Android device. Because the Frida server requires root privileges, this mode relies on an emulator or a rooted physical device. If root access is not available, Frida can also operate in embedded mode, where the Frida Gadget is directly integrated into the application by repacking the APK [46]. However, for our use case where multiple applications must be analyzed, this approach would require substantial manual effort for each target app and is therefore unsuitable.

The setup of the Frida server on the device is fully automated. By invoking the `initializeFrida` function, scripts can ensure that the correct Frida server binary is available on the device and running. If the server is not present, it is downloaded, transferred to the device, and started automatically, as shown in Figure 5.1b.



(a) Decision rules of the App Factory for instantiating different application types based on the provided input parameters.



(b) Automated initialization process of the Frida server on the device, including deployment and startup.

Figure 5.1: Overview of application handling and Frida initialization mechanisms.

Experiment Orchestrator To manage analysis workflows, an experiment orchestrator is implemented. This component is responsible for coordinating the execution of individual experiments, such as crash detection or DEX dumping. Each experiment is implemented as a subclass of the abstract `Experiment` class and defines its behavior in the `run` method. Experiments also specify execution requirements, for example whether the application must be restarted before running. The orchestrator enforces these requirements and executes all experiments sequentially, thereby reducing boilerplate code and minimizing errors in experiment scripts.

The `CrashDetectionExperiment` is used to verify that an application can be started and does not crash immediately after launch. It was employed to ensure that the artificially constructed dummy applications are runnable. Before starting the application, the Android log is cleared. The application is then started, and after a waiting period, the log is analyzed for fatal exception messages. If no crash is detected, the orchestrator proceeds to execute the next experiment in the pipeline.

The `DexDumpExperiment` class is responsible for extracting DEX files from a running application on a device for subsequent analysis. The experiment first ensures that all required directories for storing the extracted files are created. Next, it invokes the `frida-dexdump` tool, which scans the device memory for DEX files and saves them to the specified locations. After extraction, the DEX files are converted into smali code using the `baksmali` tool, enabling further analysis of the application's runtime behavior.

An example of how to use the orchestrator is shown in Listing 4.

```
//get App Instance
const adobeAcrobatReader: AppProperties = {
    appName: 'adobeAcrobatReader',
    packageName: 'com.adobe.reader'
}
const app = getAppInstance(adobeAcrobatReader)

//run experiments
const experiment: Experiment = new DexDumpExperiment()
const orchestrator = new ExperimentOrchestrator([experiment], app)
await orchestrator.execAsync()
if (experiment.hasResult()) {
    const results = dexDumpExperiment.getResult()
}
```

Listing 4: Usage of the Experiment Orchestrator

Using these utilities, a script can be constructed to extract DEX files at runtime. First, Frida is initialized on the device. Then, an `App` instance is used to install and start the application on the device. After a short delay of one second, as used in our implementation, the `DexDumpExperiment` is executed to collect the DEX files immediately after startup. Subsequently, a Frida hook is deployed to receive notifications when dynamic code loading is detected. The Frida script is implemented in a dedicated JavaScript file named `fridaScripts.js`, in which various Frida hooks can be defined. This file is the only component that needs to be modified to adapt the dynamic code loading detection. Our implementation relies on class loaders to detect code loading events. Listing 5 shows the implementation of such a hook. When one of the hooks is triggered, the implementation is notified, and the DEX dumping procedure is executed again.

```
const DexPathList = Java.use("dalvik.system.DexPathList");

DexPathList.makeDexElements?.overloads?.forEach((ovl, idx) => {
    ovl.implementation = function(files, optimizedDir, suppressed) {
        console.log("Loading detected!");
        return ovl.apply(this, arguments);
    };
});
```

Listing 5: Frida Hook

We extended `frida-dexdump` to prevent extracting DEX files that have already been observed. The goal is to avoid redundant extraction of DEX files and the subsequent disassembly of files belonging to the base application or feature modules that have already been analyzed, thereby reducing computational overhead. For this purpose, we store a hash of each previously extracted DEX file. When DEX files are extracted again, for example after registering the loading of dynamic code, the hash of the newly found DEX file is compared with the stored hashes. A DEX file is stored and disassembled only if it has not been seen before. This approach does not completely eliminate the extraction of duplicate classes. In scenarios where a largely identical DEX file appears that differs only by a single additional class, the hash will differ from previously seen DEX files and the file will be extracted again.

5.2 LibPecker Improvements

We describe the changes made to LibPecker. First, we present the modifications introduced to improve runtime performance. These include caching the signatures of libraries and APKs for reuse, replacing the data structure used during class comparison to enable faster lookup, and extending the prefiltering mechanism to filter out additional libraries, thereby reducing the number of required comparisons. Then, we describe in detail the improvements aimed at increasing the detection rate, focusing on reducing the limitations imposed by package renaming.

To accelerate the detection of libraries in an application, we added two new modes. In the `Preprocess` mode, the signatures created by `LibPecker` are stored in a file to enable reuse of the preanalyzed library artifacts in subsequent detection executions. In contrast to the original code, which allowed only a single library to be provided, a directory containing multiple libraries can now be specified as a runtime argument.

Our added code then iterates through each library in the directory. Existing `LibPecker` functionality is used to load the libraries and create the signatures of classes, methods, and fields. This process produces a `LibProfile` object for each analyzed library. To store these objects in a file, it is necessary to ensure that every class used by `LibProfile` that contains relevant data implements the `Serializable` interface. Most of the relevant classes were already serializable; only `LibProfile` itself required modification. We added `writeObject` and `readObject` methods, utilizing Java's `ObjectOutputStream` and `ObjectInputStream`. Additionally, an instance field was introduced to store the name of the library within the `LibPecker` profile.

We also implemented an `ObjectStore` class, which provides utilities to persist data in a binary format and restore it to its original representation. The data to be cached, in our case the `LibProfile` objects, is first serialized into byte arrays. These byte arrays are then stored in a single binary file with a well-defined layout, as illustrated in Figure 5.2. The file begins with a header containing a four-byte integer that specifies the total number of stored elements. The file body consists of a sequence of tuples, each comprising a four-byte integer indicating the length of the subsequent element, followed by the binary data representing that element.

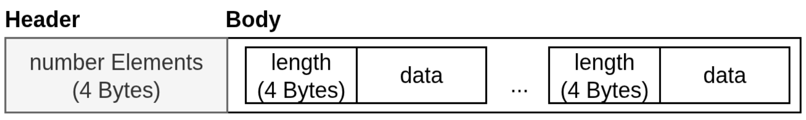


Figure 5.2: File Structure of Library Profile Cache

When restoring the cached `LibProfile` objects in the *cache* mode, the `ObjectStore` is used to read the data. The elements are first read as byte arrays according to the corresponding size recorded in the file. The byte arrays are then deserialized back into the original objects. The provided APK path is used to load and analyze the application. The signatures created by `LibPecker` are stored in an `ApkProfile` object. For every library loaded from the cache file, the comparison is executed. The `ApkProfile` is retained in memory until all libraries have been compared.

In the class comparison step, we replace the ArrayLists that store the signatures of the classes being compared with a hash based data structure in order to improve lookup performance. The new data structure is a wrapped HashMap.

The HashMap uses the signature string as its key. The corresponding value is a custom `Pair` class that stores two integer values. The first value represents the number of occurrences of a given signature in the class and is immutable. The second value is initialized to the same value and is used as a mutable counter during the comparison process. This mutable value can be decremented when a signature match is found and reset to its original value after the comparison.

For each class, such a HashMap is constructed from the signature list and stored in the corresponding `ClassProfile` object. This construction is performed for both method signatures and field signatures. During the class comparison, the cached HashMap is loaded from the `ClassProfile`.

As in the original implementation, the signatures of the library class are compared with the signatures of the APK class. The goal is to determine how many signatures are shared between the two classes. To achieve this, we iterate through the signature list Sig_{lib} of the library class. For each signature $s \in Sig_{lib}$, we check whether it is contained in the signature set Sig_{apk}.

If a match is found, the corresponding counter in the wrapper HashMap is decremented. This approach conceptually replaces the removal of a signature from Sig_{apk} used in the list based implementation, while avoiding costly structural modifications. After the comparison is completed, the wrapper data structure can be reset efficiently by restoring the mutable counters of all `Pair` objects to their original immutable values.

We introduce a second prefiltering technique which, in combination with the original approach, aims to filter out additional libraries without reducing recall. The proposed prefiltering method is based on the use of Android API calls. We extract the API calls for each method of both the library and the APK. The extraction code is executed when LibPecker's `MethodProfile` is created, ensuring that the extraction is performed for every method. First, class initialization methods are ignored, as they are generated by the compiler. Next, we iterate through each instruction of the method. We classify a method invocation as an API call if it belongs to an Android API package. The Android API packages are defined in the official Android Developer documentation and include packages such as `java`, `javax`, `android`, as well as specific `org` packages, for example `org.json` [49]. If the class of a called method belongs to one of these packages, the method invocation is added to the list of API calls. The API calls are stored as string based signatures using the following format:

$$className\#methodName(ParameterType, ParameterType, \dots) : ReturnType$$

During the prefiltering phase, we check whether the API calls contained in the library set are also present in the APK set. When the API call based prefilter is used in isolation, a library is excluded from further comparison if the fraction of its API calls observed in the APK is below a specified threshold.

In addition to the standalone API-call-based prefilter, we implement a combined approach. In this variant, the original signature-based prefilter is applied together with the API-call-based prefilter. The threshold of the API-call-based prefilter is adjusted dynamically based on the outcome of the signature-based similarity. Specifically, we apply the following rules:

$$\begin{array}{rcl}
\textbf{Similarity by signature-based prefilter} & \Rightarrow & \textbf{Threshold of API-call-based prefilter} \\
< 0.85 & \Rightarrow & \text{ignored} \\
\geq 0.8 \text{ and } < 0.9 & \Rightarrow & 0.95 \\
\geq 0.9 \text{ and } < 0.95 & \Rightarrow & 0.8 \\
\geq 0.95 & \Rightarrow & 0.4
\end{array} \tag{5.1}$$

For high similarity values, we apply relaxed thresholds for the API-call-based prefilter, whereas lower similarity values are subject to stricter thresholds. When the signature-based prefilter indicates a similarity below 0.85, the library is filtered out. The same applies if the library does not meet the dynamically determined API-call-based prefilter threshold.

Since we observed package repacking to be a potential source of reduced detection accuracy, we implemented a strategy to mitigate its effects. We treat each application uniformly, regardless of whether its packages are repacked. To achieve this, when building the LibPecker model, we assign every class to the same default package. This is implemented by modifying the creation of the `PackageNodes` in the `TreeAnalysis` class. Instead of extracting the package name from each class to determine its original package and creating a corresponding class instance, all classes are placed in a single default package. As a result, the original package structure and hierarchy are effectively collapsed into one large package.

In the main detection flow, this single package is used to approximate the original packages. In the original implementation of the `PackagePairCandidate` class, for a library package A, similar APK packages are identified and the most similar packages are kept as candidates for that library package. The comparison of package A with each APK package is performed by comparing each class in A with the classes of the APK packages.

In our implementation, there are no longer multiple APK packages. Instead, each class in A is compared with every class in the single, large package. The total number of comparisons remains the same, so this approach does not introduce additional runtime overhead. Similarity is calculated as in the original implementation: for each library class A_{lib}, the APK class B_{apk} with the highest similarity is stored. The library classes are evaluated in order of their weight. This produces a mapping where, for each library class A_{lib}, the corresponding APK class B_{apk} with the highest similarity is reported.

Unlike the original implementation, we do not maintain separate lists per library package and APK package, but only per library package, since all APK packages have been collapsed into a single package. After this mapping, the matched classes are removed from the large package, and an `ApkPackageProfile` is created from them. This procedure is repeated for each library package, thereby bundling APK classes corresponding to each library package into separate profiles. The remaining APK classes remain in the large package, as they are not relevant for subsequent comparison steps.

6
Evaluation

We evaluate the implementation in this chapter. First, we describe the evaluation setup, introducing the applications used to test the detection and extraction of DEX files at runtime, as well as the methodology for assessing the improvements made to LibPecker. In the subsequent Results section, we present the outcomes of these experiments, including the extracted components from the test applications and the observed improvements in precision, recall, and processing speed.

6.1 Evaluation Setup

In this section, we describe the setup used to evaluate our implementation. We first introduce the test applications, including a Demo App provided by Google to demonstrate dynamic code loading capabilities, and a real-world application, Adobe Acrobat Reader. We then explain how we verify whether the extracted code is novel. Next, we present the dataset used to evaluate the performance of LibPecker. Finally, we structure the LibPecker improvement evaluation into three aspects: performance, prefilter effectiveness, and detection accuracy.

6.1.1 Runtime Detection

To test the extraction of DEX files, we used a rooted Pixel 6 device. A brief guide on rooting this device is provided in the Appendix B. Furthermore, an ADB connection is required to enable USB debugging, which is also described in the Appendix.

To observe dynamic code loading, we used two applications. The first is a demo application from Google that demonstrates the capabilities of dynamic code loading using Android App Bundles (AAB) [50]. It illustrates how classes from dynamic feature modules can be utilized. The application implements a simple counter, where the module responsible for persisting the state is organized as a separate feature module. This module is loaded as soon as the user clicks the ”save” button in the app’s UI.

Technically, there is a base APK that implements the core logic of the application and is capable of loading the storage feature module, which is packaged as a separate APK. Together, they form an AAB, stored as a `.apks` file. In the case of the demo App, the bundle contains the `base.apk` and `storage.apk`.

In practice, the AAB is uploaded to Google Play. During installation, only the base application is delivered. When requested, the corresponding feature module is downloaded from Google Play Services and added to the app at runtime. For testing purposes, we relied on the tool `bundletool` [51], which can simulate dynamic feature loading without requiring the app to be uploaded to Google Play. In this scenario, the entire App Bundle is stored on the device. When the app attempts to load the feature module, the missing part is not downloaded from the network but instead retrieved from local storage.

The demo application provides complete insight into what is loaded at each point in time, enabling an assessment of the completeness of the dynamically loaded code detected by our method. However, since this is a demo application, it may lack the complexity of real-world apps.

For the second application, we selected Adobe Acrobat Reader, a real-world app that is known to use dynamic code loading. Google Play published a case study describing how Adobe implemented this feature in their application [52]. The blog article does not specify exactly what is loaded at runtime. In addition, since the app is closed source, we cannot verify whether our detected code is complete.

Both applications were analyzed using our runtime detection implementation. The demo app was installed by our prototype using `bundletool`. After the initial extraction of the DEX files, the button in the UI was clicked manually to trigger the loading of the dynamic code. The extracted code was then disassembled into smali code using the tool `baksmali`. Adobe Acrobat Reader was installed via Google Play. Since it was not clear which actions in the app trigger dynamic loading, several functionalities of the app were tested manually in a non-systematic manner until dynamic code loading occurred.

In addition to the demo app and the real-world app, a further category of applications was used. We built a collection of dummy applications that consist solely of an empty main activity and randomly chosen libraries. Each generated Dummy App contains 2 to 3 randomly selected libraries from a collection of over 30,000 libraries. The purpose of these applications is not to provide functionality, but to create a large dataset of apps with known content. The motivation for building this dataset is not part of this work; however, it was used to verify R1.1. We implemented an automated pipeline to deploy the applications to an emulator, launch them, dump the DEX files immediately after startup, and compare the dumped DEX files with those contained in the APK.

To answer research question R1.1, we compare the initially dumped DEX files with the DEX files included in the APK. The goal is to demonstrate that both sets are equivalent and, consequently, to show that the initial runtime dumping of DEX files is complete.

We ran the Demo App, Adobe Acrobat Reader, and 100 Dummy Apps, and performed DEX dumping with a one-second delay to ensure that each application had fully started. To avoid differences caused by version mismatches, we use the same APK that is executed on the device for the comparison. In the case of Adobe Acrobat Reader, which was installed from Google Play, the APK was extracted directly from the device using ADB. Subsequently, the DEX files of the APK were disassembled into Smali code in the same manner as the dumped DEX files.

When comparing the resulting Smali files, obfuscation does not need to be considered. For obfuscated applications, the dumped DEX files contain the same obfuscation artifacts as the original APK, which makes the comparison straightforward.

We compare classes by examining their fields and method headers. First, we attempt to match classes that are exactly identical, i.e., having the same fields and method headers. If an exact match is not found, we compute a similarity score using the Jaccard index.

We do not perform a direct comparison of the DEX files, because the number of DEX files in the APK differs from the number of DEX files dumped from memory. Consequently, we cannot assume that the dumped files correspond exactly to the original DEX files in the APK. Similarly, the ordering of methods and fields in the disassembled smali files cannot be assumed to remain unchanged. For these reasons, we restrict our analysis to signature-based comparisons, rather than performing full-text comparisons of the smali files.

To reduce the number of comparisons, we ignore packages present in the dumped DEX files but not in the APK; the filtered packages are listed in the Results section. Furthermore, we prioritize matching classes that reside in the same package and share the same (obfuscated) class name, under the assumption that the package structure is preserved.

We then investigate whether the dynamically loaded and dumped DEX files include new code, to answer Research Questions R1.2 and R1.3. To this end, we compare the dumped code with the APK to identify whether the files are already part of the statically bundled APK. We expect to observe very low similarity, with few or no identical classes. Subsequently, we manually analyze the newly dumped Smali code. Several aspects can help us understand what new code might have been loaded:

- **Package Name**. When not obfuscated, the package hierarchy can provide useful insights into which library the code belongs to.
- **Source Directive**. This directive in the smali file provides a hint about the original source file, although it may be obfuscated.
- **APK Name**. When the loaded DEX file is delivered within an APK, the filename itself may offer useful information.

We aim to identify the downloaded components and determine their origin. To this end, we run Adobe Acrobat Reader both with and without an internet connection to assess whether the artifacts are downloaded or loaded from local storage. Furthermore, we disable Google Play Services to evaluate whether the dynamic loading depends on this service.

6.1.2 Improvements to LibPecker

To empirically investigate the changes made to LibPecker, we use the open-source LibScan dataset [22], which contains 225 distinct Android applications. These applications are obfuscated using different tools and configurations, resulting in a total of 939 application variants. A summary of the dataset composition is provided in Table 6.1. In total, the dataset includes 453 distinct libraries. We selected the LibScan dataset because, as described in Section 3.1, it is currently the most up-to-date and largest publicly available dataset. However, once datasets from Zhou et al. [2] or Gu et al. [5] become available, more comprehensive alternatives may be used.

Table 6.1: Composition of the LibScan dataset with respect to obfuscation tools and techniques.

Obfuscator	**Technique**	**Number of Apps**
DashO	Control Flow Randomization (ctrl)	79
DashO	Package Flattening and Identifier Renaming (fltn-rnm)	79
DashO	Dead Code Removal (rmv)	79
DashO	ctrl, fltn-rnm, rmv	159
ProGuard	*not specified*	152
Allatori	*not specified*	188
Not obfuscated	–	203
Total		939

The evaluation of improvements in speed, recall, and precision is divided into multiple measurements. First, we assess the impact of technical changes aimed at improving the runtime performance of the tool. These include library preprocessing, caching of app analyzes, and the use of improved data structures for faster comparisons.

Next, we evaluate the effectiveness of the prefiltering step. We begin by assessing each prefilter independently: the original signature-based prefilter and the newly introduced API-call-based prefilter. We then examine the performance of the combined prefilter variant.

Finally, we measure the effectiveness of the detection improvements. This evaluation is conducted once using the original prefilter and once using the combined prefilter, allowing us to assess the impact of the improvements under both configurations.

All experiments were conducted on a system with 16 CPU cores and 32 GiB of memory, using an AMD EPYC (3rd Gen) Milan B1 processor based on the Zen 3 microarchitecture and manufactured on a 7 nm process node.

Performance Improvement To evaluate the effect of the performance improvement changes, we recorded the execution time while analyzing 939 applications, each compared against 453 libraries. Both the applications and libraries were taken from the LibScan dataset. Preprocessing time was measured separately from the main detection process, as its benefit depends strongly on the number of analyzed applications. For each improved version of the tool, we measure the total time required to preprocess all libraries, the average preprocessing time per library, the average time to process an APK, and the average processing time per APK per library and report the speed-up associated with each improvement step. Every improvement step builds upon the improvements applied in the preceding step.

In addition to the LibScan applications, we selected three well-known real-world applications (WhatsApp, Signal, and Firefox) and measured the analysis time using the same set of 453 libraries. This evaluation illustrates how representative the dataset-based measurements are when applied to widely used and complex real-world applications.

Since the dataset used by Gu et al. [5] has not yet been published, a direct comparison is not possible. However, we computed the average time per library from their reported results and used this value to compare against our baseline.

Prefiltering We evaluate the effectiveness of the existing signature-based prefilter, the new API-call-based prefilter, and a combination of both prefilters. The goal is to determine how many libraries are incorrectly filtered out, resulting in false negatives, and how many additional libraries could be filtered out to improve performance. We use the same dataset and ground truth as previously presented.

We define false negatives (FN) as libraries that are included in the APK but are incorrectly filtered out by the prefiltering step. True negatives (TN) are libraries that are correctly filtered out because they are not present in the APK. True positives (TP) are libraries that are included in the APK and are not filtered out, while false positives (FP) are libraries that are not included in the APK but are retained by the filter.

To quantify the negative impact of the filter on detection, meaning the removal of libraries that are actually in the APK, we use the recall metric (Eq. 6.1). Achieving a high recall is necessary to ensure that LibPecker can detect the correct libraries in the main detection phases. To evaluate the filter's contribution to improving performance, we use specificity (Eq. 6.1), which measures the fraction of non-included libraries that are correctly filtered out. High specificity indicates that the filter removes as many unnecessary libraries as possible, thereby increasing the speed of the overall detection process. In addition, we report the total number of incorrectly filtered libraries, corresponding to false negatives, aggregated over all APKs, as well as the maximum number of libraries filtered out in a single APK.

$$Specificity = \frac{TN}{TN + FP} \qquad Recall = \frac{TP}{TP + FN} \tag{6.1}$$

First, we evaluate the original signature-based prefilter. To assess how the threshold affects its performance metrics, we consider values other than the default 50% threshold. Specifically, we test thresholds from 50% to 95% in 5% steps, and to explore the limit, we also include 99%. To evaluate our new API-call-based prefilter, we report results for threshold values from 10% to 100% in 10% steps. The threshold defines the proportion of API calls in a library that must be present in the APK. For example, a 100% threshold requires that every API call found in the library is also included in the APK. While running this evaluation, the original signature-based prefilter is deactivated and does not affect the results. The combined prefilter is evaluated using different dynamic thresholds. The following summarizes the rules used to set these thresholds. The value on the left of the arrow represents the similarity obtained from the default signature-based prefilter, while the value on the right indicates the threshold applied to the API-call-based prefilter, chosen based on this similarity.

Variant A

Sim. by Sig-based Pr.	$\Rightarrow$	**Resulting Thresh.**
< 0.8	$\Rightarrow$	ignored
≥ 0.8 and < 0.9	$\Rightarrow$	0.8
≥ 0.9 and < 0.95	$\Rightarrow$	0.5
≥ 0.95	$\Rightarrow$	0.3

Variant B

Sim. by Sig-based Pr.	$\Rightarrow$	**Resulting Thresh.**
< 0.7	$\Rightarrow$	ignored
≥ 0.7 and < 0.8	$\Rightarrow$	0.8
≥ 0.8 and < 0.9	$\Rightarrow$	0.5
≥ 0.9	$\Rightarrow$	0.3

Variant C

Sim. by Sig-based Pr.	$\Rightarrow$	**Resulting Thresh.**
< 0.8	$\Rightarrow$	ignored
≥ 0.8 and < 0.9	$\Rightarrow$	0.95
≥ 0.9 and < 0.95	$\Rightarrow$	0.8
≥ 0.95	$\Rightarrow$	0.3

Variant D

Sim. by Sig-based Pr.	$\Rightarrow$	**Resulting Thresh.**
< 0.7	$\Rightarrow$	ignored
≥ 0.7 and < 0.8	$\Rightarrow$	0.95
≥ 0.8 and < 0.9	$\Rightarrow$	0.8
≥ 0.9	$\Rightarrow$	0.3

Variant E

Sim. by Sig-based Pr.	$\Rightarrow$	**Resulting Thresh.**
< 0.85	$\Rightarrow$	ignored
≥ 0.8 and < 0.9	$\Rightarrow$	0.95
≥ 0.9 and < 0.95	$\Rightarrow$	0.8
≥ 0.95	$\Rightarrow$	0.4

Detection Accuracy Improvement We report the effectiveness of our improved LibPecker variant in detecting the included libraries. Detection performance is evaluated using precision, recall, the resulting F1 score, and runtime. These metrics are reported both for the overall results and separately for each obfuscation category.

As a baseline, we first measure the results using only the implementation-level runtime improvements. This variant is chosen as the baseline because, otherwise, the runtime would be prohibitively long, and none of these modifications affect detection accuracy. Subsequently, we evaluate a version that does not include detection accuracy improvements but incorporates the enhanced prefiltering mechanism. We then evaluate the version with improved detection accuracy, considering both the cases with and without the enhanced prefiltering mechanism. For a clearer overview, we summarize the evaluated variants in the matrix shown in Table 6.2.

Table 6.2: Overview of the evaluated variants combining prefilter and improvement strategies.

	Default Prefilter	**Combined Prefilter (E)**
Only Impl.-Level improvements	Variant 1	Variant 2
Impl.-Level improvements & Algorithm changes	Variant 3	Variant 4

We further evaluate the impact of different variants of the detection accuracy improvements, namely the variant that does not remove matched classes (Variant 3.I) and the variant that uses adapted signatures (Varian 3.II).

6.2 Results

This section presents the results of our experiments, organized into three main parts. First, we evaluate Dynamic Code Loading, examining the completeness of the initial DEX dump and analyzing the dumped files to determine which code is observable and whether it differs from the original APK. Second, we report the results of LibPecker's prefiltering step, focusing on the accuracy of the filter in retaining libraries included in the APK while removing unnecessary ones. Finally, we assess the improvements made to LibPecker, considering both runtime performance and detection effectiveness. Together, these results provide a comprehensive view of the tool's accuracy and efficiency.

6.2.1 Runtime Detection

As described in the evaluation setup, we first examine whether the initial DEX dump contains the same classes as the APK. Next, we analyze which code can be successfully extracted. Finally, we determine whether the extracted code is indeed new and not already present in the APK. This analysis demonstrates whether our approach is capable of extracting useful classes that extend the capabilities of static APK analysis.

We present the results of comparing the APKs with the dumped DEX files. Overall, every class in the APKs could be mapped to a corresponding class in the dumped data. The following sections provide more details for each application, highlighting classes that exist only in the dumped DEX files and those that could not be mapped.

Adobe Acrobat Reader We compared the DEX files in the Adobe Acrobat Reader APK with the dumped DEX files. In total, we collected 85,964 dumped classes and 63,842 APK classes. For each APK class, we were able to map it to an equivalent dumped class, corresponding to 74.3% of dumped classes. We detected 19,850 dumped classes (23.1%) that reside in packages not present in the APK's DEX files. These include packages such as `dalvik`, `java`, `javax`, `sun`, `jdk`, `libcore`, `android.adservices`, `android.nfc`, `android.system`, and others. The remaining 2,272 dumped smali files (2.6%) could not be mapped to any APK class, even though they are in packages that exist in the APK's DEX files, such as `android.app`, `android.media`, `android.os`, `android.provider`, `kotlinx.coroutines`, `m2`, and `m7`.

Demo App In our Demo App, we extracted 27,118 dumped classes and 6,950 APK classes. Every APK class could be mapped to an equivalent class in the dumped data, corresponding to approximately 25.6% of dumped classes. We detected 20,168 classes (74.4%) that reside in packages present only in the dumped DEX files. These packages are similar to those found in the Adobe Acrobat Reader app.

Dummy applications Besides the classes mentioned earlier, which appear at runtime and are not part of the APK, every class in the artificially generated apps could be mapped with an identical APK class.

These results show that the initial dump contains the classes packaged in the APK and that the complete DEX files can be successfully collected. In the following, we present the findings from the dumped classes of our two test applications, the Demo App and Adobe Acrobat Reader. In both cases, we detected dynamically loaded code that was not present in the original APK. The analysis allowed us to identify runtime loading events and extract additional code artifacts that are not visible through static analysis. We then show that the loaded code is exclusively part of the dynamically loaded module and not present in the initial dump.

Demo App When interacting with the corresponding UI element, the Dynamic Feature Module is loaded. This behavior also occurs without an internet connection, as the application was installed from a locally stored app bundle. The loading event was successfully detected, and the package `com.google.android.samples.storage` was extracted. This package contains exactly the classes of the separately packaged feature module.

Adobe Acrobat Reader After the initial dump, we observed multiple events in which code was added to the base application. Two types of events can be distinguished: code loading from sources already installed on the device, and code downloaded from a remote source, triggered by a user event and accompanied by a UI hint indicating that a feature is being downloaded. The Table 6.3 shows the loaded modules.

Table 6.3: Dynamically Loaded Modules

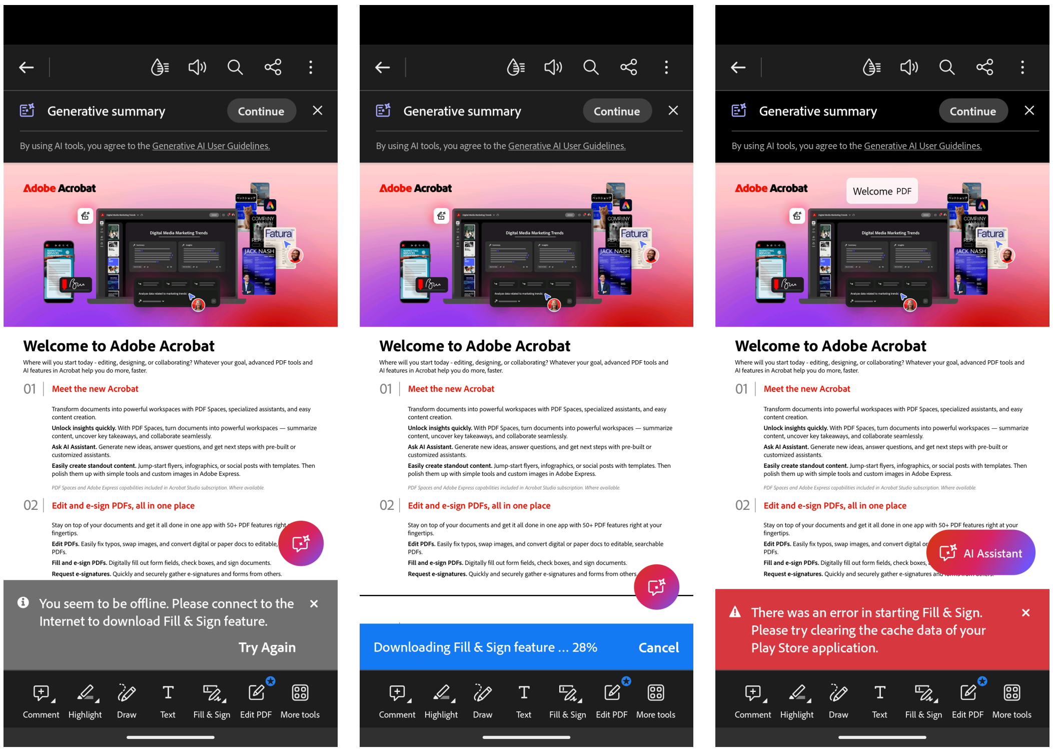
Module Name	**is local Source**	**Source Name**
Dynamite Loader	✓	`com.google.android.gms`
Dynamite Measurement	✓	`com.google.android.gms`
Dynamite TFlite	✓	`com.google.android.gms`
Google Certificates	✓	`com.google.android.gms`
Vision Barcode	✓	`com.google.android.gms`
WebView	✓	`com.google.android.trichromelibrary`
WebView	✓	`com.google.android.webview`
OpenCV	×	
FastFieldDetection	×	
obfuscated	×	

The package name `com.google.android.gms` refers to Google Play Services. This application is preinstalled on most android devices and provides various services to other applications that include the corresponding client library [53]. These services include, for example, the Machine Learning Kit, which offers an API for barcode scanning tasks. When Google Play Services was disabled on the device, the corresponding modules could no longer be loaded. With Google Play Services disabled, the three remote modules were not downloaded (see 6.1c). The WebView modules are also loaded from preinstalled Android system components and are independent of Google Play Services.

We identified two distinct runtime downloads, each corresponding to a different application feature. When attempting to use the **Fill & Sign** functionality from the menu bar in the Adobe Acrobat Reader app, the user was informed that an additional feature was being downloaded via a corresponding UI element (see Figure 6.1). Three packages were successfully detected and extracted by our implementation.

- The package `org.opencv.*` likely belongs to the open-source Computer Vision library [54]. Since the library is open source and the package and class names are not obfuscated, we compared the downloaded artifacts with the original library and confirmed that it is a subset of the OpenCV library.

- The package `com.adobe.libs.fastfielddetection` integrates into the structure of the base application. The `com.adobe.libs` package already exists and contains several subpackages; however, this specific `fastfielddetection` package was not previously part of it.



You seem to be offline. Please connect to the Internet to download Fill & Sign feature.

Try Again

Downloading Fill & Sign feature ... 28% Cancel

There was an error in starting Fill & Sign. Please try clearing the cache data of your Play Store application.

(a) No internet connection is available (b) The download is in progress (c) Google Play Services are disabled

Figure 6.1: To download the Fill & Sign feature, an internet connection and Google Play Services are required.

- The third package, `Dr`, is completely obfuscated and contains four classes. Its origin could not be determined from the dump.

The second download was triggered by using the **Edit PDF** button. The event of loading an APK named `split_PDFEditFontsDF.apk` was detected; however, no DEX files were extracted. Manual inspection of this APK in the device storage using sudo rights revealed that it contained only additional fonts and no DEX files.

We further analyzed whether the detected and extracted DEX files are indeed absent from the static APK, in order to address research question R1.3. This analysis provides evidence regarding whether this dynamic extraction process offers additional value compared to static analysis alone.

Dynamically downloaded Features The comparison between the initially dumped files and the dynamically downloaded DEX files (including the modules *OpenCV*, *FastFieldDetection*, and an obfuscated package, see Table 6.3) resulted in an overall similarity of 0.47. Eight classes were matched directly, and one additional class exhibited a similarity of 0.7. The remaining classes showed lower similarity values (0.44 or below).

We manually analyzed the matches obtained through signature-based comparison, focusing on those with a similarity above the average (> 0.47). As described in Table 6.4 all of these higher-similarity matches were found to be either false positives or not relevant. Consequently, the classes do not correspond to existing classes in the initial dump and represent newly added content.

Table 6.4: Matching results of classes from dynamically loaded modules and the base application, including the reasons for high similarity.

Class from loaded module	**Class from initial dump**	**Sim**	**Reason**
com/adobe/libs/fastfielddetection	**D1**		
/a	/b	1	All classes implement the `java.util.Comparator` interface, thereby exposing the same method signatures; the classes, however, are not equal.
/b	/d		
/c	/f		
/d	/h		
/e	/j		
com/adobe/libs/fasfielddetection		1	These classes are identical; however, they do not implement any logic, containing only an empty default constructor and static fields.
/R	pl/u		
/BuildConfig	com/adobe/ftpdf/BuildConfig		
org/opencv/android	**Ol**	1	The classes share the same signature, but invoke different methods within their bodies.
/a	/b		
Dr	**Gc**	0.7	The classes share many static fields, which primarily drive the high similarity. The base application defines only fixed values, whereas the dynamically loaded class defines logic and initializes values by invoking static functions.
/a	/d		

Webview Module The comparison of the 82,847 classes from the initial dump with the 4,785 classes from the loaded WebView module shows that 321 classes had exact matches. This indicates that the majority of classes are likely new. Due to the large number of classes, a full manual investigation was not performed for the exact matches or for matches with high similarity. Spot checks, however, revealed a pattern similar to that observed for the newly added features in the dynamically downloaded code: The investigated classes were not identical.

Dynamite Modules While many classes in the dumped packages `com/google/android/gms` and `com/google/protobuf` could be directly mapped to corresponding classes in the APK, no such mappings were found for classes in the dumped packages `com/google/mlkit`, `com/google/barhopper`, `com/google/photos`, and `com/google/protos`. This suggests that these packages were newly introduced by the dump process.

In summary, the results show that, in addition to system-provided components, dynamically loaded modules initiated by explicit user interactions were observed. These included feature modules that introduced additional resources, such as fonts, as well as modules that provided executable code at runtime. In particular, the dynamic loading of the OpenCV library, which provides computer vision functionality, was detected. Beyond identifying dynamically loaded modules, the extracted code was verified to be newly introduced at runtime rather than duplicated from the APK.

6.2.2 LibPeckers Prefiltering

Table 6.5 shows the results of the original signature-based prefilter. The recall remains just below 100% for thresholds up to 90%. For thresholds above 90%, the recall begins to decrease, though it stays relatively high even at a threshold of 99%. However, the specificity remains very low for low thresholds. Figure 6.2a illustrates the increasing specificity across different thresholds, while recall decreases only slightly.

The effectiveness of the new API-call-based prefilter is shown in Table 6.6. For high thresholds, which require that many API calls be present in the APK, high specificity can be achieved; however, recall decreases to around 75%. The results of the API-based prefilter are illustrated in Figure 6.2b.

The resulting values for the different variants of the combined prefilter are illustrated in Figure 6.2c, while Table 6.7 reports the exact numerical values. Variant E achieves the highest specificity, whereas variants B and D achieve the highest recall. The results of variant C are similar to those obtained with the signature-based prefilter using a threshold of 95%.

Table 6.5: Results of the original signature-based prefilter, showing maximum false negatives per APK, total false negatives, average specificity, and average recall across different similarity thresholds.

Threshold	**Max FNs per APK**	**Total FNs**	**Avg Specificity (%)**	**Avg Recall (%)**
50%	1	5	2.66	99.97
55%	1	5	3.40	99.97
60%	1	5	3.84	99.97
65%	1	5	5.80	99.97
70%	1	5	7.11	99.97
75%	1	5	9.60	99.97
80%	1	7	12.36	99.90
85%	1	7	17.43	99.90
90%	1	11	27.91	99.85
95%	3	53	44.78	98.81
99%	3	169	64.60	96.36

Table 6.6: Results of the API-call-based prefilter, showing maximum false negatives per APK, total false negatives, average specificity, and average recall across different API-call matching thresholds.

Threshold	**Max FNs per APK**	**Total FNs**	**Avg Specificity (%)**	**Avg Recall (%)**
10%	1	5	0.63	99.97
20%	1	5	2.13	99.97
30%	2	57	6.13	99.33
40%	2	132	13.07	98.47
50%	6	631	20.51	89.72
60%	10	1084	32.72	83.37
70%	10	1115	53.71	83.01
80%	10	1134	76.62	82.74
90%	11	1472	86.79	76.85
100%	17	1614	93.92	75.31

Table 6.7: Results of the combined prefilter variants, showing maximum false negatives per APK, total false negatives, average specificity, and average recall for each variant.

Variant	Max FNs per APK	Total FNs	Avg Specificity (%)	Avg Recall (%)
A	2	186	31.02	98.42
B	2	118	17.77	99.27
C	2	198	44.33	98.34
D	2	118	29.41	99.27
E	2	173	48.28	97.49

Table 6.8: Performance comparison of LibPecker optimizations using 453 libraries

		Preprocessing		Matching	
Configuration	#Libs	Time (s)	Time/Lib (s)	Time (s)	Time/Lib (s)
Result Gu et al. [5]	230	–	–	223.13	0.97
Baseline (no modifications)	453	–	–	631.76	1.39
Cache app analysis	453	–	–	62.23	0.14
Preprocessing libraries	453	10.45	0.02	36.31	0.08
HashMap-based comparison	453	11.90	0.03	29.86	0.07

6.2.3 Improvement to LibPecker

In this section, we first present the measured speed improvements and comparisons with large, well-known real-world applications. We then report on the impact of the prefiltering mechanism and the improvements in detection accuracy on the overall detection capabilities.

Table 6.8 summarizes the performance measurements of the improved LibPecker, including the execution time reported by Gu et al. as a reference. The preprocessing step requires only a short amount of time and thus effectively reduces the overall app analysis time.

The relative performance improvements are summarized in Table 6.17. Each successive optimization yields an additional speedup, resulting in an approximately 21× performance improvement from the implementation level optimizations compared to the baseline implementation, and a 4× improvement due to changes to the prefilter and algorithm. Overall, a total speedup of 91× was achieved.

A comparison with real-world applications is presented in Table 6.9. This measurement was conducted using the improved version of LibPecker (Variant 3). The results indicate that apps from the LibScan dataset are processed significantly faster. Notably, even the slowest APK in the dataset is analyzed approximately seven times faster than Firefox.

Table 6.10 (Variant 1) shows the results without detection accuracy enhancements and with the default prefilter, serving as a baseline for comparison. Tables 6.11 (Variant 2), 6.12 (Variant 3), and 6.15 (Variant 4) present the results of the detection accuracy enhancements and the impact of the prefilter across different configurations. Tables 6.13 (Variant 3.I) and 6.14 (Variant 3.II) show the results of adaptations of Variant 3. Values that outperform the baseline (Variant 1) are shown in bold.

Comparing Variant 1, which does not include detection accuracy enhancements and uses the default prefilter, to Variant 4, which incorporates both detection accuracy enhancements and the improved prefilter, we observe improvements in both detection accuracy and processing speed. For Variants 3 and 4, precision improves across all categories, while recall decreases. The only exception is DashO, for which both precision and recall increase. Nevertheless, the gain in precision is sufficiently large to compensate for the reduction in recall, resulting in a higher F1-score overall.

Table 6.9: Average time per library for real-world applications compared to the LibScan dataset

Application	**Time per Library (s)**
WhatsApp	~ 7.7
Signal	~ 5.6
Firefox	~ 2.2
Slowest APK (LibScan Dataset)	0.29
Average APK (LibScan Dataset)	0.07

Table 6.10: **Variant 1 (Baseline):** Results without detection accuracy improvements using the default prefilter

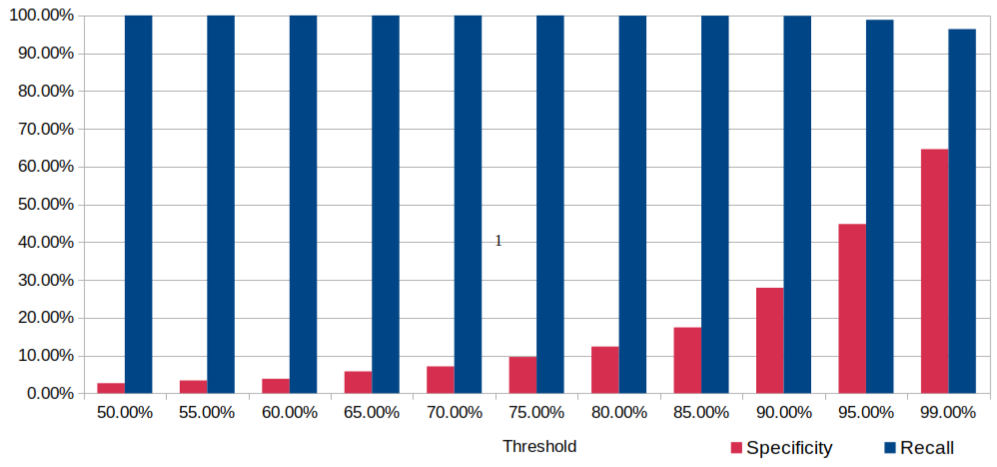
Category	**Precision**	**Recall**	**F1-score**	**Avg Time/App (s)**	**# APKs**
Overall Average	0.54	0.88	0.67	29.86	939
Not obfuscated	0.56	1.00	0.72	29.62	203
Allatori	0.59	0.98	0.74	29.91	188
Proguard	0.53	1.00	0.70	25.81	152
DashO	0.42	0.41	0.42	40.20	159
fltn-rnm	0.55	0.91	0.69	25.96	79
ctrl	0.59	1.00	0.74	26.30	79
rmv	0.55	0.91	0.69	24.85	79

Table 6.11: **Variant 2:** Results of the improved LibPecker implementation without detection enhancements using the combined prefilter

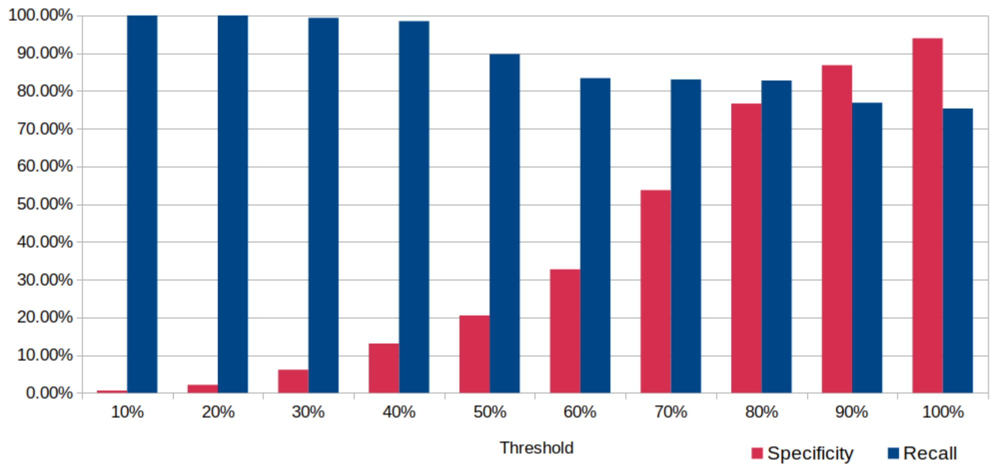
Category	**Precision**	**Recall**	**F1-score**	**Avg Time/App (s)**	**# APKs**
Overall Average	0.54	0.87	0.67	**15.29**	939
Not obfuscated	0.56	1.00	0.72	**17.55**	203
Allatori	0.58	0.94	0.72	**16.87**	188
Proguard	0.53	1.00	0.70	**14.02**	152
DashO	0.42	0.41	0.41	**14.48**	159
fltn-rnm	0.55	0.91	0.69	**13.37**	79
ctrl	0.59	1.00	0.74	**13.80**	79
rmv	0.55	0.91	0.69	**13.25**	79

Table 6.12: **Variant 3:** Results of the improved LibPecker implementation with detection enhancements using the default prefilter

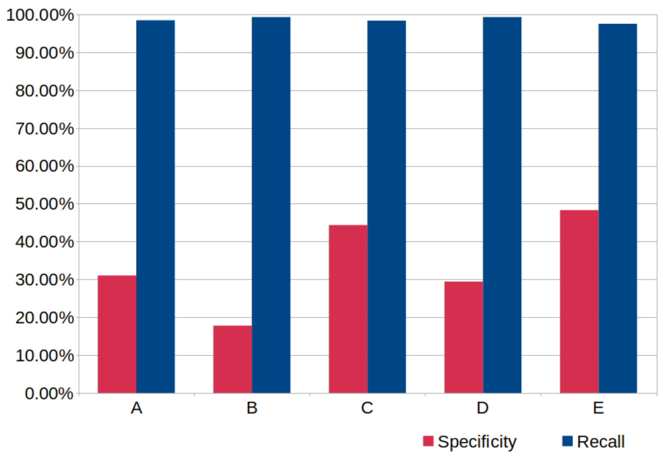
Category	**Precision**	**Recall**	**F1-score**	**Avg Time/App (s)**	**# APKs**
Overall Average	**0.80**	0.75	**0.77**	**17.30**	939
Not obfuscated	**0.87**	0.75	**0.81**	**6.44**	203
Allatori	**0.70**	0.87	**0.78**	**29.07**	188
Proguard	**0.87**	0.77	**0.82**	**4.28**	152
DashO	**0.56**	**0.74**	**0.64**	**34.26**	159
fltn-rnm	**0.94**	0.64	**0.76**	**14.49**	79
ctrl	**0.94**	0.65	**0.77**	**14.49**	79
rmv	**0.94**	0.62	**0.75**	**13.70**	79



(a) Signature-based Prefilter



(b) API-call-based Prefilter



(c) Combined Prefilter

Figure 6.2: Recall and specificity for different threshold values.

Table 6.13: **Variant 3.I:** Results of the improved LibPecker implementation with detection enhancements using the default prefilter. Mapped classes are retained and not removed during the processing.

Category	Precision	Recall	F1-score	Avg Time/App (s)	# APKs
Overall Average	0.55	0.85	0.67	30.82	939
Not obfuscated	**0.59**	0.91	0.71	31.75	203
Allatori	0.58	0.89	0.71	31.79	188
Proguard	0.56	0.93	0.70	27.49	152
DashO	**0.54**	**0.77**	**0.64**	37.60	159
fltn-rnm	0.50	0.75	0.60	26.86	79
ctrl	0.51	0.77	0.61	27.52	79
rmv	0.50	0.73	0.59	26.18	79

Table 6.14: **Variant 3.II:** Results of the improved LibPecker implementation with detection enhancements using the default prefilter. Adapted CDS are used.

Category	Precision	Recall	F1-score	Avg Time/App (s)	# APKs
Overall Average	**0.80**	0.75	**0.77**	**17.11**	939
Not obfuscated	**0.87**	0.76	**0.81**	**6.38**	203
Allatori	**0.70**	0.86	**0.77**	**29.07**	188
Proguard	**0.87**	0.77	**0.82**	**4.16**	152
DashO	**0.56**	**0.74**	**0.64**	**33.60**	159
fltn-rnm	**0.94**	0.66	**0.77**	**14.16**	79
ctrl	**0.94**	0.65	**0.77**	**14.35**	79
rmv	**0.94**	0.62	**0.75**	**13.59**	79

Table 6.15: **Variant 4:** Results of the improved LibPecker implementation with detection enhancements using the combined prefilter

Category	Precision	Recall	F1-score	Avg Time/App (s)	# APKs
Overall Average	**0.81**	0.75	**0.78**	**6.95**	939
Not obfuscated	**0.91**	0.80	**0.85**	**2.80**	203
Allatori	**0.70**	0.85	**0.77**	**15.05**	188
Proguard	**0.92**	0.83	**0.87**	**1.68**	152
DashO	**0.51**	**0.59**	**0.55**	**11.99**	159
fltn-rnm	**0.94**	0.70	**0.80**	**4.49**	79
ctrl	**0.94**	0.71	**0.81**	**4.08**	79
rmv	**0.95**	0.69	**0.80**	**3.66**	79

Table 6.16: Comparison of LibPecker Variants across overall performance metrics. Best values are highlighted in bold.

Variant	Precision	Recall	F1-score	Δ F1	Avg Time/App (s)
Variant 1 (Baseline)	0.54	**0.88**	0.67	-	29.86
Variant 2	0.54	0.87	0.67	+0.00	15.29
Variant 3	0.80	0.75	0.77	+0.10	17.30
Variant 3.I	0.55	0.85	0.67	+0.00	30.82
Variant 3.II	0.80	0.75	0.77	+0.10	17.11
Variant 4	**0.81**	0.75	**0.78**	**+0.11**	**6.95**

Table 6.17: Relative performance improvements of LibPecker optimizations

Comparison	**Speedup**
Implementation level optimizations	*21.16*$\times$
Baseline $\rightarrow$ Cache app analysis	10.15$\times$
Cache app analysis $\rightarrow$ Preprocessing libraries	1.71$\times$
Preprocessing $\rightarrow$ HashMap optimization (Variant 1)	1.22$\times$
Prefilter & Algorithm changes	*4.3*$\times$
HashMap optimization (Variant 1) $\rightarrow$ Variant 3	1.73$\times$
Variant 3 $\rightarrow$ Variant 4	2.49$\times$
Overall	**90.90**$\times$

7 Discussion

We discuss the results from the evaluation chapter, focusing on improvements to LibPecker and the dynamic extraction of DEX files. Afterwards, we summarize the main threats to validity.

7.1 LibPecker Improvements

Across all optimized variants, substantial reductions in runtime are observed, with Variant 4 exhibiting the strongest improvement. The runtime evaluation in this work is based on wall-clock time measurements rather than CPU time. Consequently, individual measurements are subject to minor fluctuations of approximately one second between repeated runs per application. While this limits the precision of individual measurements, these speedups are sufficiently large to indicate that the applied optimizations have a meaningful and systematic impact on overall performance rather than reflecting measurement artifacts.

Preprocessing and prefiltering play a critical role in reducing computational overhead and limiting the search space prior to detection. Runtime is strongly influenced by the number of candidate libraries contained in the database. Variants that exclude a larger number of irrelevant libraries at an early stage require fewer comparisons and consequently achieve lower analysis times per application. At the same time, this dependency highlights a trade-off between runtime efficiency and the risk of prematurely excluding relevant library candidates.

The effectiveness of the applied prefilter thresholds may depend on the characteristics of the evaluated dataset, including app complexity and obfuscation techniques. While the experiments demonstrate that prefiltering is essential for achieving acceptable runtime, it remains unclear whether the selected thresholds generalize well to other datasets or real-world scenarios. Further evaluation on larger and more diverse datasets would be required to assess the robustness of these parameter choices.

Nevertheless, the newly introduced API-call-based prefilter neither outperforms the existing prefilter nor does the combination of both prefilters yield better results than the threshold-optimized signature-based prefilter. This suggests that filtering libraries based on used API calls is not particularly effective. A comparison at a finer granularity, such as examining the Android API calls used at the class level, could potentially be more meaningful and help filter out irrelevant class candidates. However, experiments in this direction were also not very promising.

The use of synthetically created packages further contributes to the observed runtime improvements. By generating only one synthetic APK package per library package, the proposed approach avoids enumerating all package combinations observed in real applications. This substantially reduces the number of partitions considered during the computation of the optimal mapping between application packages and library components. Since the complexity of this mapping step increases rapidly with the number of candidate packages, limiting the search space in this manner leads to significant performance gains.

In addition to algorithmic changes, execution speed was further improved through implementation-level optimizations. These optimizations do not alter the underlying detection logic and therefore do not affect detection accuracy. This observation underscores that empirical performance comparisons between tools must be interpreted with caution, as reported runtimes may reflect differences in implementation quality and engineering effort rather than fundamental differences in the underlying detection approaches.

A comparison between applications from the LibScan dataset and real-world applications reveals a substantial disparity in analysis time, with real-world applications requiring significantly longer processing. This suggests that evaluations based solely on curated or simplified datasets may underestimate the runtime costs encountered in realistic deployment scenarios. In practical settings involving library databases with tens of thousands of libraries and multiple versions, analysis may still require hours or days, indicating that real-time detection remains challenging despite the achieved improvements.

With respect to detection performance, Variants 3 and 4 achieve a clear improvement over the baseline. Across most obfuscation categories, precision increases consistently, while recall decreases noticeably, indicating a shift toward a more conservative detection strategy. This trade-off reduces the number of false positives at the cost of missing some true library instances. An exception is observed for applications obfuscated with DashO, where both precision and recall improve relative to the baseline. Given that DashO aggressively flattens and repackages library code, this result indicates that the proposed enhancements remain effective even when traditional package-based heuristics are disrupted. Overall, the improved variants favor correctness over completeness.

Variants 3.I and 3.II do not provide further improvements over the base Variant 3. While Variant 3.II yields detection results identical to the base configuration, Variant 3.I leads to a noticeable degradation in detection performance. A key difference between these variants lies in the handling of already matched classes during the construction of synthetic packages. In Variant 3, such classes are removed prior to synthetic package generation, whereas Variant 3.I omits this step. The loss of performance in Variant 3.I indicates that the removal of already matched classes is a critical design aspect of the synthetic package construction process.

Although achieving an F1-score just below 80% represents a clear improvement over the baseline, the results are not yet suitable for production use. The evaluated dataset comprises fewer than 1,000 application variants and primarily includes relatively simple applications, while the exact configuration of the applied obfuscation techniques remains unknown. To draw stronger conclusions regarding practical effectiveness, evaluation on larger and more realistic datasets, as well as on more extensive library collections, would be required. The inability to validate the approach against the benchmark dataset of Gu et al. [5], which has not yet been released, further limits the comparability of the results.

Finally, detection accuracy in this work is evaluated at the library level rather than at the library version level. For many security-related use cases, identifying the exact library version is essential, as vulnerabilities often affect only specific releases. The limited reliability of library-level detection therefore represents a fundamental obstacle to dependable version-level identification, highlighting an important direction for future research.

Overall, the presented results demonstrate that both runtime performance and detection accuracy can be improved, thereby addressing research question R2. Implementation-level optimizations alone yield an approximate twentyfold speedup, while changes to the detection approach, including adapted prefiltering and the use of synthetic packages, further improve performance by an additional factor of four. At the same time, detection accuracy improves by more than 10 percentage points in terms of F1-score, primarily due to increased precision, albeit at the cost of reduced recall across most obfuscation categories.

7.2 Runtime Detection

The findings demonstrate that runtime extraction provides significant insights beyond what is achievable through static analysis alone. The observation that dynamically loaded code, including feature modules and system-provided components, is not present in the original APK underscores the limitations of purely static approaches. Runtime analysis therefore emerges as a necessary complement, revealing dependencies that are introduced only during application execution, which is particularly relevant for modern modular Android applications.

These observations highlight that dynamically loaded feature modules can introduce substantial third-party functionality during execution, adding new code to the application and extending its capabilities beyond what is statically packaged. For instance, the dynamic loading of the OpenCV library illustrates how runtime execution can introduce additional functionality that may escape detection by purely static analysis approaches. Furthermore, the verification that the extracted code is newly introduced at runtime confirms that runtime extraction can reveal dependencies absent from static views of the application, thereby enabling subsequent static analysis techniques to be applied to a more complete code base.

The system-provided components allow applications to reuse common functionality and behave similarly to shared libraries. From a library detection perspective, they constitute a distinct class of dependencies that is typically invisible to static analysis, yet may still influence an application's functionality and security posture. Furthermore, WebView has been the target of multiple security vulnerabilities, as reported by Hu et al. [55], which underscores its relevance for security analyzes. The ability to detect WebView at runtime therefore enhances the identification of applications that may be exposed to known WebView-related attacks, even when this dependency is not apparent from the statically packaged code.

The primary limitation of the proposed approach arises in applications that deliberately conceal dynamic code loading or employ anti-debugging or anti-hooking mechanisms. Packed applications, for example, may modify DEX files directly in memory or expose them only temporarily, preventing reliable extraction of executable code [26]. While prior work has demonstrated that such countermeasures can be addressed by adapting the Android Runtime [12] or by monitoring applications at the kernel level [28], these techniques are beyond the scope of this work.

Dynamic code loading is detected in this work by monitoring the class loader system. Other loading mechanisms that operate independently of the `DexPathList` class may exist and are not automatically captured. However, the modular design of the implementation allows additional hooks to be introduced, and periodic dumping of DEX files remains possible even without complete coverage of all loading mechanisms.

Finally, the signature-based comparison used to assess similarity between DEX files introduces inherent imprecision. Classes implementing the same interface may exhibit high similarity due to shared method signatures, and classes of different sizes are weighted equally. As a result, similarity may be overestimated in certain cases. While this limits the precision of individual similarity measurements, it does not undermine the overall conclusion that dynamically loaded code can be reliably distinguished from statically packaged code in the evaluated scenarios.

Overall, these findings demonstrate that dynamically loaded DEX files can be extracted during runtime execution and that this extraction reveals third-party libraries and system components that are not visible in the APK. This confirms that runtime analysis constitutes a necessary complement to static library detection approaches and addresses research question R1.

7.3 Threats to Validity

Several factors may threaten the validity of the results presented in this thesis. These threats are related to dataset characteristics, dynamic code detection mechanisms, evaluation metrics, and tool implementation choices.

Dataset Limitations The LibScan dataset used in this study is relatively small and consists primarily of simple applications. Consequently, the results may not fully reflect the performance or detection behavior of more complex or larger-scale real-world applications. Furthermore, the applications selected for runtime testing, such as Adobe Acrobat Reader and Google's Demo App, serve only as illustrative examples and do not cover the full diversity of possible app behaviors or obfuscation techniques. Therefore, while the observed trends provide valuable insights, they should be interpreted with caution when generalizing to other applications or larger datasets.

Unknown Obfuscation Configurations The exact obfuscation techniques applied to the APKs in the evaluated datasets are not known. In particular, the degree of package flattening, package repacking, and other transformations remains unspecified. This limits the ability to assess the effectiveness of the proposed approach for specific obfuscation strategies and may affect the generalizability of the observed detection improvements.

Detection of Dynamically Loaded Code The detection of dynamically loaded code depends on monitoring class loader behavior. Other loading mechanisms or techniques may exist that are not covered by this approach, potentially resulting in undetected code execution. For example, some dynamically loaded modules in applications such as Adobe Acrobat Reader may not be detected. As a result, dynamically loaded code may not always be fully captured.

DEX File Comparison To evaluate whether the dumped code matches the APK and whether dynamically loaded code is already present in the APK, we rely on a signature-based similarity comparison. While this approach provides an indication of similarity, it is not exact and may produce imprecise results in some cases. Nevertheless, the overall trends observed in the experiments are considered reliable.

Prefilter Threshold Selection The thresholds used for library prefiltering were fine-tuned based on the evaluated dataset. Consequently, their effectiveness may depend on the specific characteristics of this dataset, such as app complexity or obfuscation techniques. It is unclear whether the selected thresholds generalize to other datasets or real-world scenarios.

Runtime Measurement The evaluation of performance is based on wall-clock time measurements rather than CPU time. As a result, execution times may be subject to minor fluctuations caused by system load or background processes. While these variations can affect the precision of individual measurements, they are unlikely to invalidate the observed relative performance trends.

Anti-Debugging and Anti-Hooking Techniques Dynamic code detection can be hindered by anti-debugging or anti-hooking techniques employed by applications. Moreover, the proposed dynamic analysis approach is generally not applicable to heavily packed applications, where runtime behavior is deliberately obscured. These limitations restrict the applicability of the approach in adversarial settings.

8 Future Work

Our evaluation highlights several limitations of existing library detection approaches. Many methods rely on structural features of the application, which are deliberately altered by obfuscation techniques, making reliable detection at the version level challenging. Furthermore, newly developed obfuscation techniques can invalidate features previously considered resilient. Consequently, future research should explore semantic comparison methods that focus on code behavior rather than structure or syntax.

Semantic comparison techniques, such as those studied in code plagiarism detection (e.g., Type IV clones) and binary code analysis, offer promising directions. Token-based, tree-based, and graph-based approaches, as well as hybrid methods, have shown the ability to detect functionally equivalent code despite syntactic differences [56–58]. Extending these methods to library-level detection in mobile applications could improve robustness against obfuscation.

Another avenue for future work concerns cross-platform applications. Current tools are primarily designed for native Android applications in Java or Kotlin and are not applicable to frameworks such as Flutter, React Native, or Cordova [59–61]. In these frameworks, application logic is executed by the Flutter engine, a JavaScript runtime, or a WebView, making DEX-based analysis insufficient. Developing detection techniques that can handle cross-platform execution environments remains an open challenge.

A primary goal of future work should be to further increase both detection accuracy and performance. As highlighted by Gu et al. [5], current approaches still lack production readiness, demonstrating the need for methods that can scale and remain robust in realistic scenarios. Achieving this goal will require comprehensive, large-scale, and realistic datasets, as well as extendable benchmark setups that allow for systematic evaluation and comparison of different detection techniques, following the approach initially proposed by Gu et al.

Overall, advancing library detection will require approaches that are resilient to both obfuscation and platform diversity, potentially combining semantic analysis with cross-platform support and dynamic execution monitoring. By addressing these challenges and providing robust benchmarks and datasets, future research can bridge the gap between experimental methods and production-ready solutions.

9 Conclusion

We implemented a prototype for detecting and extracting dynamically loaded code to provide a more comprehensive view of third-party library usage. Using both a demo application and the real-world application Adobe Acrobat Reader, we demonstrated that dynamically loaded code can be extracted at runtime even when it is not present in the APK. While the prototype may not detect all possible dynamically loaded code, it is designed to be extensible for future improvements. Furthermore, by introducing implementation level optimizations, adapting the prefilter, and modifying the detection algorithm, we increased the F1 score of LibPecker from 0.67 to 0.78 and achieved a speedup by a factor of 90.

By implementing a working DEX detection and extraction prototype, we demonstrated the feasibility of runtime analysis for third-party library detection. Applying the approach to the real-world application Adobe Acrobat Reader further showed that feature modules are used in practice, confirming that runtime analysis extends the completeness of third-party library detection beyond static analysis alone.

The analysis of LibPecker's detection algorithm provided deeper insight into the limitations of the approach. By relaxing structural assumptions through the use of synthetic application packages, we were able to improve detection accuracy. However, our evaluation also showed that the potential for further improvements remains limited, as only a small number of features are unaffected by structural modifications introduced by current obfuscation techniques. While the accuracy of LibPecker was improved, this work does not propose an entirely new third-party library detection approach that can be generalized across the full spectrum of detection techniques.

Our contribution enriches existing third-party library detection approaches with a hybrid method that accounts for dynamically loaded code. While existing DEX extraction tools aim to reassemble the complete application code, our approach specifically focuses on identifying code that is loaded dynamically at runtime.

Although this work establishes a foundation for detecting dynamically loaded code at runtime, future work is required to further improve detection accuracy while maintaining acceptable runtime performance. In addition, cross platform applications should be considered in order to obtain a more complete view of third-party library usage.

List of Figures

3.1 Packages and classes before renaming. 21
3.2 Package and class names are obfuscated. The main class and its package can't be renamed. 21
3.3 The `-flattenpackagehierarchy 'obfuscated'` option moves all packages into a common parent package, effectively flattening the package hierarchy. 21
3.4 The `-repackageclasses 'obfuscated'` option moves all classes from packages that can be renamed into a single package. 22
3.5 The `-repackageclasses 'obfuscated'` option can be combined with `-allowaccessmodification` to achieve the maximum possible degree of package repacking. 22

4.1 Overview of the hybrid third-party library detection tool architecture 24
4.2 DEX File Header and Data Section. Fields in the header (shaded in grey) have fixed lengths, while most fields in the data section are of variable length. The header fields `container_size` and `header_offset` are present only in DEX file version 41. . . 26
4.3 Structure of a map list item. The type identifier of the map list entry is 0x1000 (4096). . 27
4.4 Class Dependency Signature (CDS) 28
4.5 Basic Class Signature. For prefiltering, a simplified version of this basic class signature exists, which does not include the interfaces. 29
4.6 Field Signature 29
4.7 Method Signature 29
4.8 Class Signature 30
4.9 First, Frida is initialized and the application is started. Subsequently, the DEX files are extracted. Whenever the hook is triggered, the extraction process is executed again. . . . 35
4.10 First, the application contains repackaged packages. Next, all classes are merged into a single large group. Using similarity measures, we then regroup classes that are likely to belong to the same original package. Finally, we obtain a package structure without a package hierarchy. 37

5.1 Overview of application handling and Frida initialization mechanisms. 40
5.2 File Structure of Library Profile Cache 42

6.1 To download the Fill & Sign feature, an internet connection and Google Play Services are required. 52
6.2 Recall and specificity for different threshold values. 57

Listings

1 Simple Method before and after control flow randomization 17
2 String decryption algorithm (Code generated by DashO) 18
3 Obfuscated call to the string decryption method (Code generated by DashO) 19
4 Usage of the Experiment Orchestrator 41
5 Frida Hook 41

List of Tables

3.1 Benchmark Results of Gu et al [5]: Comparison of TPL detection tools at library and version level . . . 16

4.1 Example scenario: package flattening . . . 32
4.2 Example scenario: package repacking. . . . 33

6.1 Composition of the LibScan dataset with respect to obfuscation tools and techniques. . . 47
6.2 Overview of the evaluated variants combining prefilter and improvement strategies. . . . 49
6.3 Dynamically Loaded Modules . . . 51
6.4 Matching results of classes from dynamically loaded modules and the base application, including the reasons for high similarity. . . . 53
6.5 Results of the original signature-based prefilter, showing maximum false negatives per APK, total false negatives, average specificity, and average recall across different similarity thresholds. . . . 54
6.6 Results of the API-call-based prefilter, showing maximum false negatives per APK, total false negatives, average specificity, and average recall across different API-call matching thresholds. . . . 54
6.7 Results of the combined prefilter variants, showing maximum false negatives per APK, total false negatives, average specificity, and average recall for each variant. . . . 55
6.8 Performance comparison of LibPecker optimizations using 453 libraries . . . 55
6.9 Average time per library for real-world applications compared to the LibScan dataset . . 56
6.10 **Variant 1 (Baseline):** Results without detection accuracy improvements using the default prefilter . . . 56
6.11 **Variant 2:** Results of the improved LibPecker implementation without detection enhancements using the combined prefilter . . . 56
6.12 **Variant 3:** Results of the improved LibPecker implementation with detection enhancements using the default prefilter . . . 56
6.13 **Variant 3.I:** Results of the improved LibPecker implementation with detection enhancements using the default prefilter. Mapped classes are retained and not removed during the processing. . . . 58
6.14 **Variant 3.II:** Results of the improved LibPecker implementation with detection enhancements using the default prefilter. Adapted CDS are used. . . . 58
6.15 **Variant 4:** Results of the improved LibPecker implementation with detection enhancements using the combined prefilter . . . 58
6.16 Comparison of LibPecker Variants across overall performance metrics. Best values are highlighted in bold. . . . 58
6.17 Relative performance improvements of LibPecker optimizations . . . 59

A.1 Policies of R8. Policies marked with stars are handled as assertions when merging is restricted to synthetic classes 88
A.2 Policies of D8 89
A.3 Policies used exclusively by D8 or R8 90

Bibliography

[1] StatCounter, “Mobile Operating System Market Share Worldwide,” 2025, [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide, [Accessed: 2026-01-06].

[2] B. Zhou, J. Wu, X. Ling, T. Luo, and J. Zhang, “Version-level third-party library detection in android applications via class structural similarity,” Apr. 2025. DOI: 10.48550/arXiv.2504.13547.

[3] MITRE, “About the CVE Program,” 2026, [Online]. Available: https://www.cve.org/About/Overview, [Accessed: 2026-01-06].

[4] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, “Detecting third-party libraries in android applications with high precision and recall,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 141–152. DOI: 10.1109/SANER.2018.8330204.

[5] J. Gu, H. Lu, G. Nan, Y. Lin, K. Wang, Y. Guo, Y. Cao, and Y. Liu, *Revisiting third-party library detection: A ground truth dataset and its implications across security tasks*, arXiv preprint arXiv:2509.04091, 2025. DOI: 10.48550/arXiv.2509.04091.

[6] M. Backes, S. Bugiel, and E. Derr, “Reliable third-party library detection in android and its security applications,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 356–367, ISBN: 9781450341394. DOI: 10.1145/2976749.2978333. [Online]. Available: https://doi.org/10.1145/2976749.2978333.

[7] Y. Wang, H. Wu, H. Zhang, and A. Rountev, “Orlis: Obfuscation-resilient library detection for android,” in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2018, pp. 13–23.

[8] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, “A large scale investigation of obfuscation use in google play,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC ’18, San Juan, PR, USA: Association for Computing Machinery, 2018, pp. 222–235, ISBN: 9781450365697. DOI: 10.1145/3274694.3274726. [Online]. Available: https://doi.org/10.1145/3274694.3274726.

[9] A. Kelec and Z. urić, “A proposal for addressing security issues related to dynamic code loading on android platform,” *Computer Systems Science and Engineering*, vol. 35, pp. 271–282, Jan. 2020. DOI: 10.32604/csse.2020.35.271.

[10] Z. Dong, H. Liu, L. Wang, X. Luo, Y. Guo, G. Xu, X. Xiao, and H. Wang, “What did you pack in my app? a systematic analysis of commercial android packers,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, Singapore, Singapore: Association for Computing Machinery, 2022, pp. 1430–1440, ISBN: 9781450394130. DOI: 10.1145/3540250.3558969. [Online]. Available: https://doi.org/10.1145/3540250.3558969.

[11] Z. Qu, S. Alam, Y. Chen, X. Zhou, W. Hong, and R. Riley, "Dydroid: Measuring dynamic code loading and its security implications in android applications," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017, pp. 415–426. DOI: 10.1109/DSN.2017.14.

[12] Z. Ning and F. Zhang, "Dexlego: Reassembleable bytecode extraction for aiding static analysis," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 690–701. DOI: 10.1109/DSN.2018.00075.

[13] OWASP, "Mobile Application Security," [Online]. Available: https://mas.owasp.org/MASVS/10-MASVS-CODE/, [Accessed: 2025-11.28].

[14] P. Paganini, "Threat actors rely on the 'versioning' technique to evade malware detections of malicious code uploaded to the Google Play Store," 2023, [Online]. Available: https://securityaffairs.com/149150/hacking/google-play-malware-versioning-evasion.html, [Accessed: 2026-01-16].

[15] X. Zhan, T. Liu, Y. Liu, Y. Liu, L. Li, H. Wang, and X. Luo, "A systematic assessment on android third-party library detection tools," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4249–4273, 2022. DOI: 10.1109/TSE.2021.3115506.

[16] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "ATVHunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1695–1707. DOI: 10.1109/ICSE43902.2021.00150.

[17] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16, Austin, Texas: Association for Computing Machinery, 2016, pp. 653–656, ISBN: 9781450342056. DOI: 10.1145/2889160.2889178. [Online]. Available: https://doi.org/10.1145/2889160.2889178.

[18] Y. Zhang, J. Wang, H. Huang, Y. Zhang, and P. Liu, "Understanding and conquering the difficulties in identifying third-party libraries from millions of android apps," *IEEE Transactions on Big Data*, vol. 8, no. 6, pp. 1511–1523, 2022. DOI: 10.1109/TBDATA.2021.3093244.

[19] Z. Xie, M. Wen, T. Li, Y. Zhu, Q. Hou, and H. Jin, "How does code optimization impact third-party library detection for android applications?" In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24, Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 1919–1931, ISBN: 9798400712487. DOI: 10.1145/3691620.3695554. [Online]. Available: https://doi.org/10.1145/3691620.3695554.

[20] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "LibD: Scalable and precise third-party library detection in android markets," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 335–346. DOI: 10.1109/ICSE.2017.38.

[21] J. Zhang, A. R. Beresford, and S. A. Kollmann, "LibID: Reliable identification of obfuscated third-party android libraries," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, pp. 55–65, ISBN: 9781450362245. DOI: 10.1145/3293882.3330563. [Online]. Available: https://doi.org/10.1145/3293882.3330563.

[22] Y. Wu, C. Sun, D. Zeng, G. Tan, S. Ma, and P. Wang, "LibScan: Towards more precise Third-Party library identification for android applications," in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 3385–3402, ISBN: 978-1-939133-37-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/wu-yafei.

[23] J. Huang, B. Xue, J. Jiang, W. You, B. Liang, J. Wu, and Y. Wu, “Scalably detecting third-party android libraries with two-stage bloom filtering,” *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2272–2284, 2023. DOI: 10.1109/TSE.2022.3215628.

[24] Y. Zhang, X. Luo, and H. Yin, “Dexhunter: Toward extracting hidden code from packed android applications,” English, in *Computer Security – ESORICS 2015 - 20th European Symposium on Research in Computer Security, Proceedings*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 20th European Symposium on Research in Computer Security, ESORICS 2015 ; Conference date: 21-09-2015 Through 25-09-2015, Germany: Springer Verlag, Jan. 2015, pp. 293–311, ISBN: 9783319241760. DOI: 10.1007/978-3-319-24177-7_15.

[25] R. Bhan, R. Pamula, K. Kumar, N. Jyotish, P. Tripathi, P. Faruki, and J. Gajrani, “Dlcdroid an android apps analysis framework to analyse the dynamically loaded code,” *Scientific Reports*, vol. 15, Jan. 2025. DOI: 10.1038/s41598-025-88003-6.

[26] L. Xue, H. Zhou, X. Luo, L. Yu, D. Wu, Y. Zhou, and X. Ma, “Packergrind: An adaptive unpacking system for android apps,” *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 551–570, 2022. DOI: 10.1109/TSE.2020.2996433.

[27] M. Zheng, M. Sun, and J. C. Lui, “Droidtrace: A ptrace based android dynamic analysis system with forward execution capability,” in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2014, pp. 128–133. DOI: 10.1109/IWCMC.2014.6906344.

[28] M. Li, W. Niu, J. Gong, S. Li, M. Zhang, and X. Zhang, “Bpfdex: Enabling robust android apps unpacking via android kernel,” *IEEE Transactions on Information Forensics and Security*, vol. PP, pp. 1–1, Jan. 2025. DOI: 10.1109/TIFS.2025.3594559.

[29] Guardsquare, “ProGuard,” 2016–2024, [Online]. Available: https://www.guardsquare.com/proguard, [Accessed: 2025-05-11].

[30] PreEmptive, “Java Security and Android Obfuscation with DashO,” 2024, [Online]. Available: https://www.preemptive.com/products/dasho/, [Accessed: 2025-05-11].

[31] Allatori, “Allatori - Java obfuscator,” 2005-2025, [Online]. Available: https://allatori.com, [Accessed: 2025-05-11].

[32] Google, “R8 Optimizer and Obfuscator,” 2025-09-29, [Online]. Available: https://developer.android.com/build/shrink-code, [Accessed: 2025-05-11].

[33] F-Droid, “Free and Open Source Android App Repository,” 2010-2025, [Online]. Available: https://f-droid.org, [Accessed: 2025-05-11].

[34] Google, “D8 Dexer,” 2023, [Online]. Available: https://developer.android.com/tools/d8, [Accessed: 2025-05-11].

[35] Smardec Inc, “Allatori Changelog,” [Online]. Available: https://allatori.com/changelog.html, [Accessed: 2025-26-11].

[36] PreEmptive, “DashO Changelog,” [Online]. Available: https://support.preemptive.com/hc/en-us/articles/31997866992017-Changelog.

[37] A. Niroshan, S. Seneviratne, and A. Seneviratne, “State of obfuscation: A longitudinal study of code obfuscation practices in google play store,” in *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2025, pp. 592–596, ISBN: 9798400706295. [Online]. Available: https://doi.org/10.1145/3672608.3707860.

[38] ProGuard, “ProGuard Mannual. Configuration Examples,” [Online]. Available: https://www.guardsquare.com/manual/configuration/examples, [Accessed: 2025-11-27].

[39] R. Kumar and A. Kurian, “A Systematic Study on Static Control Flow Obfuscation Techniques in Java,” 2018, [Online]. Available: https://arxiv.org/abs/1809.11037.

[40] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, “The Java® Virtual Machine Specification. Java SE 21 Edition,” 2023, [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se21/html/jvms-5.html, [Accessed: 2025-11-12].

[41] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, “The Java® Language Specification. Java SE 21 Edition,” 2023, [Online]. Available: https://docs.oracle.com/javase/specs/jls/se21/html/index.html, [Accessed: 2015-11-27].

[42] Ben Weiss, “Use R8 to shrink, optimize, and fast-track your app,” 2025, [Online]. Available: https://android-developers.googleblog.com/2025/11/use-r8-to-shrink-optimize-and-fast.html, [Accessed: 2025-26-11].

[43] Google, “D8/R8 Shrinker and Dexer,” 2025, [Online]. Available: https://r8.googlesource.com/r8, Commit 6676a2e8a7ca99f9a47d556495bbce23dfe70c48 [Accessed: 2025-11-06].

[44] Hluwa, “Frida-Dexdump,” 2023, [Online]. Available: https://github.com/hluwa/frida-dexdump, [Accessed: 2025-12-15].

[45] Android Open Source Project, “Dalvik executable format,” 2025, [Online]. Available: https://source.android.com/docs/core/runtime/dex-format, [Accessed: 2025-12-15].

[46] Frida, “Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers,” 2025, [Online]. Available: https://frida.re/, [Accessed: 2025-12-28].

[47] JesusFreke, “About Smali/Baksmali,” 2022, [Online]. Available: https://github.com/JesusFreke/smali, [Accessed: 2026-01-27].

[48] P. Graux, J.-F. Lalande, V. Viet Triem Tong, and P. Wilke, “Oats’inside: Retrieving object behaviors from native-based obfuscated android applications,” *Digital Threats*, vol. 4, no. 2, Aug. 2023. DOI: 10.1145/3584975. [Online]. Available: https://doi.org/10.1145/3584975.

[49] Google, “Android API reference - Package Index,” 2025, [Online]. Available: https://developer.android.com/reference/packages, [Accessed: 2026-01-14].

[50] Android, “Android App Bundle Samples Repository - Dynamic Code Loading,” 2023, [Online]. Available: https://github.com/android/app-bundle-samples/tree/main/DynamicCodeLoadingKotlin, [Accessed: 2025-12-16].

[51] Google, “Bundletool,” 2025, [Online]. Available: https://github.com/google/bundletool, [Accessed: 2026-01-03].

[52] Google Play, “Adobe reduces app size by 20% with app bundles and dynamic delivery,” [Online]. Available: https://play.google.com/console/about/adobe-casestudy/, [Accessed: 2025-12-16].

[53] Google Play, “Overview of Google Play services,” 2025, [Online]. Available: https://developers.google.com/android/guides/overview, [Accessed: 2025-12-18].

[54] OpenCV, “OpenCV is the world’s biggest computer vision library,” 2025, [Online]. Available: https://opencv.org/, [Accessed: 2025-12-18].

[55] J. Hu, L. Wei, Y. Liu, S.-C. Cheung, and H. Huang, “A tale of two cities: How webview induces bugs to android applications,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE ’18, Montpellier, France: Association for Computing Machinery, 2018, pp. 702–713, ISBN: 9781450359375. DOI: 10.1145/3238147.3238180. [Online]. Available: https://doi.org/10.1145/3238147.3238180.

[56] G. Lee, J. Kim, M.-s. Choi, R.-Y. Jang, and R. Lee, "Review of code similarity and plagiarism detection research studies," *Applied Sciences*, vol. 13, no. 20, 2023, ISSN: 2076-3417. DOI: 10.3390/app132011358. [Online]. Available: https://www.mdpi.com/2076-3417/13/20/11358.

[57] A. Marcelli, M. Graziano, and M. Mansouri, "How machine learning is solving the binary function similarity problem," in *USENIX Security Symposium*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247802102.

[58] Y. Hu, H. Wang, Y. Zhang, B. Li, and D. Gu, "A semantics-based hybrid approach on binary code similarity comparison," *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1241–1258, Jun. 2021, ISSN: 2326-3881. DOI: 10.1109/tse.2019.2918326. [Online]. Available: http://dx.doi.org/10.1109/TSE.2019.2918326.

[59] Flutter, "Flutter architectural overview," 2025, [Online]. Available: https://docs.flutter.dev/resources/architectural-overview, [Accessed: 2026-01-08].

[60] N. Abdou, "React Native Under the Hood: How Your JS, iOS, and Android Code Run Together?" 2025, [Online]. Available: https://dev.to/nour_abdou/react-native-under-the-hood-how-your-js-ios-and-android-code-run-together-3k2e, [Accessed: 2026-01-08].

[61] Apache Software Foundation, "Cordova Documentation," 2025, [Online]. Available: https://cordova.apache.org/docs/en/latest/guide/overview/index.html, [Accessed: 2026-01-08].

[62] Oracle, "Java® Platform, Standard Edition & Java Development Kit Version 21 API Specification," 2025, [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.compiler/javax/lang/model/util/Elements.Origin.html, [Accessed: 2025-11-12].

[63] Chromium, "Optimizing Java Code," [Online]. Available: https://chromium.googlesource.com/chromium/src/+/185ad34cd06171c0771b0da9fae67b28ba35f783/build/android/docs/java_optimization.md, [Accessed: 2025-11-26].

[64] D. F. Hsu, X. Lan, G. Miller, and D. Baird, "A comparative study of algorithm for computing strongly connected components," in *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, 2017, pp. 431–437. DOI: 10.1109/DASC-PICom-DataCom-CyberSciTec.2017.85.

Appendices

A
Class Merging

In this appendix, the merge policies used by R8/D8 are described in detail. An overview of the policies used is provided at the end of the appendix. A general description of class merging is given in 4.2.1.

A.1 Vertical Class Merging

This chapter provides a list of the policies to identify merge candidates for vertical merging. The following policies summarize the cases in which merging the supertype A into the subtype B is forbidden.

No Abstract Methods on Abstract Classes[Code] Merging class A into class B is forbidden if class A is abstract and class B is concrete while containing abstract methods. This would result in a concrete class with abstract methods, which is not allowed.

No Annotation Classes [Code] Merging annotation class A into class B is forbidden, because annotation classes behave similarly to interfaces, and merging them into class B is not permitted.

No Class Initialization Changes [Code] Merging class A into class B is forbidden if it could change the behavior or timing of class initialization (`<clinit>` methods). Specifically:

- If both class A and class B have class initializers, merging is forbidden because R8 does not support merging `<clinit>` methods.
- If class A or class B has `<clinit>` side effects, merging is forbidden as the order of execution cannot be guaranteed.
- If class A is an interface with a `<clinit>` method with side effects, merging into class B is forbidden.
- If class A has no static fields, no static methods, and is never instantiated, merging into class B is allowed, since the `<clinit>` method is never called.

No Directly Instantiated Classes [Code] Merging class A into class B is forbidden if class B directly instantiates class A.

No Enclosing Method Attributes [Code] Merging class A into class B is forbidden if either class has an `EnclosingMethod` attribute (local or anonymous inner classes), as this would break the relationship between the class and its enclosing method.

No Field Resolution Change [Code] Merging class A into class B is forbidden if class B implements an interface that declares a static field with the same name and type as an instance field in class A. After merging, instructions accessing the field could bypass the expected `IncompatibleClassChangeError`.

No Illegal Accesses [Code] Merging class A into class B is forbidden whenever previously legal accesses would become illegal. Specific cases include:

- Class A is public and class B is package-private in the same package.
- Classes A and B are in different packages, and class B is not public.
- Class A implements a non-public interface.
- A public method in class A is overridden by a non-public method in class B.
- Protected or package-private members in class A become inaccessible after merging into class B.

No Inner Classes [Code] Merging class A into class B is forbidden if either class contains inner classes, because inner classes hold references to their outer classes, which could break after merging.

No Interfaces With Invoke-Special to Default Method Into Class [Code] Merging interface A into class B is forbidden if interface A defines a default method which is called by class B. Differences in `invoke-special` instructions could lead to inconsistent method resolution.

No Interfaces with Unknown Subtypes [Code] Merging interface A into class B or any other type is forbidden if interface A has unknown or dynamic subtypes, or if A is an immediate interface of an instantiated lambda, due to potential runtime ambiguities.

No Invoke Super No-Such-Method Error [Code] Merging class A into class B is forbidden if:

- Class A is a non-interface class implementing an interface I with a default method m.
- Class B is a non-interface subclass of A.

In this case, merging would cause a `NoSuchMethodError`, because B would attempt to resolve the method in the wrong superclass context. Adding `implements I` to B is not a valid fix.

No Kept Classes [Code] Merging class A into class B is forbidden if class A is configured to be kept.

No Locking Merging [Code] Merging class A into class B is forbidden if either class may be used as a lock (via static synchronized methods or `synchronized(this)`).

No Method Resolution Change [Code] Merging interface A into class B is forbidden if class B implements another interface with a conflicting default method. Converting `invoke-interface A.m` to `invoke-virtual B.m` would hide an `IncompatibleClassChangeError`.

No Nested Merging [Code] Merging class A into class B is forbidden if class B is itself a source class in another merge, to prevent nested merging conflicts (e.g., if A→B and B→C, A cannot merge into B).

No Non-Serializable Class into Serializable Class [Code] Merging non-serializable class A into serializable class B is forbidden.

No Service Interfaces [Code] Merging service interface A into class B is forbidden if class B is pinned or kept according to keep rules.

Same API Reference Level [Code] Merging class or interface A into class or interface B is forbidden if A and B have different API reference levels. This policy is activated only when *Library API Modeling* is enabled in the system configuration

Same Feature Split [Code] Merging class or interface A into class or interface B is forbidden if they belong to different feature splits or if only one is in the base module.

Same Main Dex Group [Code] Merging class or interface A into class or interface B is forbidden if they do not belong to the same Main-Dex group or class list.

Same Nest [Code] Merging class A into class B is forbidden if they do not belong to the same nest, because private field access assumptions would be broken. Classes not in any nest may be merged freely.

Same Startup Partition [Code] Merging class A into class B is forbidden if class A is a startup class and class B is not, as startup loading behavior would be disrupted. This policy is deactivated if the startup profile is empty or if startup-boundary optimizations are enabled.

Successful Virtual Method Resolution in Target [Code] Merging class A into class B is forbidden if a virtual method from class A does not resolve to a single, unique target when looked up on class B.

A.2 Horizontal Class Merging

The following policies are used to merge classes that are siblings, that is, classes that share the same supertype. First, the single-class policies are applied to filter out unsuitable classes. Then, using the multi-class policies, the remaining merge-candidate groups are split into subgroups such that they fulfill all requirements for merging.

A.2.1 Single Class Policies

Check Synthetic Classes [Code] The Java API specification defines synthetic constructs as follows: ”A synthetic construct is one that is neither implicitly nor explicitly declared in the source code. Such a construct is typically a translation artifact created by a compiler” [62]. Therefore, synthetic classes are observable only in compiled code.

SyntheticKinds (SyntheticNaming.java, line 221ff) are an enum-like system that classifies the different types of synthetic artifacts R8 creates and provides an indication of what each synthetic construct represents. Not every synthetic construct is eligible for merging. Mergeability is defined for each SyntheticKind.

This policy ensures that this property and the runtime configuration is taken into account. The following rules are evaluated.Note that when code shrinking or optimization is disabled, we are restricted to synthetics.

- If merging is disabled, synthetic classes are filtered out.
- If we are restricted to synthetics, non-synthetic classes are filtered out.
- If we are restricted to synthetics, synthetic classes that are not mergeable are filtered out.

No Annotation Classes [Code] This policy prevents annotation types from being merged. A class is considered an annotation if it is declared using the *@interface* construct.

No Check Discard [Code] With the -checkdiscard directive, R8 can be configured to remove specific items [63]. If a class is configured for discard, it cannot be merged.

No Class Initializer With Observable Side Effects [Code] If a class A has a class initializer, it cannot be merged except in the following cases:

- Class A is a Kotlin-generated synthetic lambda class, or
- Class A's initializer has no observable side effects. This applies when the class initialization can be safely postponed.

The superclass of class A may have a class initializer. This does not pose a problem, since one of the rules for merging classes is that they must share the same parent class.

No Direct Runtime Type Checks [Code] A class A cannot be merged if it is used in a runtime type check, for example through an expression such as *obj instanceof A*.

No Enums [Code] Prevents enums, or classes that extend an enum, from being merged.

No Failed Resolution Targets [Code] If a class contains methods that failed to be resolved during analysis, it cannot be merged.

No Inner Classes [Code] Prevents classes with inner classes from being merged.

No Instance Field Annotations [Code] Prevents a class from being merged if one of its instance fields has an annotation.

No Interfaces [Code] Prevents an interface from being merged. This policy is activated only when *interface merging* is disabled in the system configuration.

No Keep Rules [Code] Prevents a class from being merged if the class is specified in a `-keep` rule.

No Kotlin Metadata [Code] Provides a sanity check to ensure that only classes with removed Kotlin metadata reach this point.

No Native Methods [Code] Prevents a class from being merged if it contains native methods.

No Records [Code] Prevents records from being merged.

No Resource Classes [Code] This policy prevents resource classes from being merged. A class is treated as a resource class if its name begins with *R$* or contains *$R$*.

No Service Loaders [Code] This policy prevents the merging of classes involved in Java's `ServiceLoader` mechanism.

Not Matched By No Horizontal Class Merging [Code] Checks whether a rule in the configuration disables horizontal class merging for this class. In addition to class-specific rules, it verifies whether shrinking and optimization are permitted at all.

Only Classes With Static Definitions And No Class Initializer [Code] This policy prevents a class from being merged if it contains a class initializer (`<clinit>`). Furthermore, if a class has non-static members, this policy also prevents it from being merged.

A.2.2 Multi Class Policies

Now, we present the multi-class policies. These policies take a group of classes as input and produce a set of subgroups as output. Each subgroup contains only classes that are mergeable according to this policy and all previously applied policies.

No Class Annotation Collisions [Code] This policy ensures that at most one class with annotations is present in a merge group. The group g is therefore divided into subgroups s such that each subgroup contains at most one class that has annotations. If the group g contains only a single annotated class, the original group g is returned, as it already satisfies the constraint.

Not Two Inits With Monitors [Code] This policy ensures that at most one class with a constructor that is synchronized or uses monitor instructions is present in a merge group. A constructor is considered to use monitor instructions if it contains synchronization constructs such as `synchronized(this)`{...}. The group g is split into subgroups s such that each subgroup contains at most one class of this kind. If the group g contains only a single class with such a constructor, the original group g is returned, since the constraint is already satisfied. This policy is activated only for specific ART VM versions.

Limit Interface Groups [Code] This policy prevents merged interfaces from becoming excessively large. It ensures that the total number of methods in a merged interface remains below 100. To achieve this, subgroups are created such that each subgroup contains at most 100 methods in total.

Minimize Instance Field Casts [Code] This policy aims to merge classes whose multisets of instance-field types are identical, thereby minimizing the number of instance-field casts. Two classes have the same instance-field-type multiset if they possess the same number of instance fields of identical types. For example, if a class has two instance fields of type `String` and one of type `int`, its multiset is $\{\text{String} : 2, \text{int} : 1\}$. All classes are grouped according to their instance-field-type multiset. If multiple classes share a unique multiset, they are placed together in a new group. If only a single class has a unique multiset, it is added to the smallest existing group.

No Class Initializer Cycles [Code] When the JVM executes a class initialization, a lock is acquired to ensure that the class initialization is not executed multiple times. The class initialization of a class C is executed, for example, when class C or any subclass of C is instantiated for the first time [40]. This policy aims to prevent class initializer cycles that can lead to deadlocks.

Consider two threads in the code snippet below. Thread t initializes class A (which, during its initialization, also initializes class B) and consequently holds the locks for both A and B. Simultaneously, thread u initializes class C. Thread u might have to wait for A's lock, but eventually thread t releases it, allowing thread u to proceed. In this scenario, a deadlock does not occur.

Since classes B and C are siblings, they are candidates for class merging. Consider the situation where B and C are merged into a single class BC. Thread t initializes class A and holds the lock for A. Now, thread t must initialize class BC instead of class C as it did before the merging. Therefore, t requires the lock for BC to proceed. Meanwhile, thread u attempts to initialize BC, acquires its lock, and waits for the release of A's lock. This situation creates a circular wait, resulting in a deadlock.

```
class A {
    static {
        new B();
    }
}
class B extends A {}
class C extends A {}
```

No Deadlocks [Code] The goal of this policy is to prevent deadlocks. The policy groups classes such that each group contains at most one class that either has static synchronized methods or is used as a lock. All other classes in the group must not contain static synchronized methods, nor can their references be used as locks (i.e., they must not be lock candidates).

No Default Interface Method Collisions [Code] We must prevent merging interfaces when doing so could change which default method is invoked at runtime. In the code, the developers of R8 provide an example of such a scenario. If $a.m()$ is called, the default implementation of the method m in interface J is executed. When interfaces I and K are merged, the default implementation of m from interface K would be added to interface I. In this case, it becomes unclear which implementation of m should be resolved for the call $a.m()$. Note that interface K does not need to implement the method m itself; it is sufficient if K inherits such a method. However, when an interface overrides m directly, this problem does not arise.

```
interface I {}
interface J {
    default void m() { print("J"); }
}
interface K {
    default void m() { print("K"); }
}
class A implements I, J {}
```

This policy is activated only when *interface merging* is enabled in the system configuration.

No Default Interface Method Merging [Code] This policy prevents merging interfaces that declare the same default interface method. Classes are grouped such that no pair of interfaces within a group have conflicting default methods.

No Weaker Access Privileges [Code] A strongly connected component (SCC) of a directed graph $G = (V, E)$ is a subset $S \subseteq V$ in which, for every pair of vertices $v, w \in S$, there exists a path from v to w and vice versa [64]. In the context of R8, the graph is treated as a bidirectional graph in which edges represent the extends and implements relationships. Such an SCC may contain both interfaces and classes. For example, consider the following hierarchy:

```
public interface I {}
public interface J extends I {}
public class A implements I {}
```

I, J, and A form an SCC since each type is reachable from the others through the bidirectional hierarchy edges.

Consider a group G of classes that can be merged without conflicts. To determine whether a class A can be added to this group, we perform the following checks:

1. Compute the set S_A of non-public virtual methods of all classes in A's SCC, and the set T_G of inherited interface methods of all classes in group G. If $S_A \cap T_G \neq \emptyset$, class A cannot be added.
2. For each class C_i in group G, compute the set S_i of non-public virtual methods of all classes in C_i's SCC. Compute the set T_A of inherited interface methods of class A. If there exists any S_i such that $S_i \cap T_A \neq \emptyset$, class A cannot be added.

In other words, this policy ensures that:

- Nothing in class A hides or overrides interface methods from group G.
- Nothing in group G hides or overrides interface methods that class A inherits.

Using this constraint, merge-group candidates are split into subgroups such that each subgroup satisfies this requirement.

Only Directly Connected Or Unrelated Interfaces [Code] This policy prevents merging interfaces in a way that would create cycles in the class hierarchy. Consider the following example:

```
interface I extends ... {}
interface J extends I, ... {}
interface K extends J, ... {}
```

The following merge groups are allowed: $\{I, J\}$, $\{J, K\}$, and $\{I, J, K\}$. However, attempting to merge $\{I, K\}$ would introduce a cycle:

```
interface IK extends J, ... {}
interface J extends IK, ... {}
```

Thus, only interfaces that are directly connected (via extends/implements) or completely unrelated can be merged. This policy is activated only when *interface merging* is enabled in the system configuration.

Preserve Interface Method Dispatch [Code] This policy prevents merging that would alter the dispatch of interface default methods, potentially changing program semantics. Consider the following example:

```
interface I { default void m() {print("I"); }
interface J extends I { @Override default void m() {print("J"); }
class A implements I {}
class B implements J {}
```

When we call $a.m()$, $I.m()$ is dispatched, since class A inherits the default implementation of m from interface I. As a result, “I” will be printed. After merging class B into A, this behavior changes: the merged class A now implements both I and J. A call to $a.m()$ will dispatch $J.m()$, and “J” will be printed instead. This behavior differs from the original program, which is undesirable.

Formally, this policy addresses the risk of altering program semantics during class merging. It begins by identifying a set of default methods of interest, specifically, interface default methods whose resolution could be affected by merging. The policy then systematically analyzes the merge group and all of its subclasses to determine whether the resolution of any of these methods would change as a result of the merge. If any such change is detected in any subclass, the merge is considered semantically unsafe and is therefore prohibited. This policy is activated only if the API level supports default and static interface methods.

Preserve Method Characteristics [Code] This policy allows methods to be merged only if they have the same visibility and library method override information. Note that methods that are static or private (excluding constructors) do not require merging. Such methods can be safely renamed.

Given a group of classes that can be merged, to determine whether a class A can be added to this group, the following checks are performed:

1. For each method in class A, check whether a method with the same signature exists in any class in the group.
2. If such a method exists (and it is not static or private), verify that it has the same visibility and library method override information.

If these conditions are satisfied for every method in class A, then class A can be safely added to the group.

Prevent Class Method And Default Method Collisions [Code] This policy prevents merging classes in a way that would alter the resolution of class methods versus interface default methods. Consider the scenario where class C implements an interface I with a default method m and extends class A. In this case, class A cannot be merged with a class B that contains a method m' with the same signature as m. Before merging, a call to $c.m()$ resolves to $I.m()$. After merging, $c.m()$ would resolve to $B.m'()$, changing the runtime behavior. An example illustrating this policy is:

```
class A {}
class B { public void m() { ... } }
interface I { default void m() { ... } }
class C extends A implements I {}
```

Respect Package Boundaries [Code] This policy ensures that classes are merged only if package boundaries are respected. A class A cannot be merged across package boundaries if any of the following conditions hold; such a class is called *restricted*:

- A is not public.

- A or one of its members has an annotation (annotations may access non-public items).
- A implements a non-public interface from the same package (interfaces from other packages are public by definition).
- A includes a member that is package-private or protected.
- A has a field of a non-public type (the compiler might add synthesized check-cast instructions dependent on the package).
- A method of A accesses classes or members that are both in the same package as A and private or package-private.

Groups are split by restricted and non-restricted classes. Restricted classes are grouped by package, so each subgroup contains classes from the same package. For each non-restricted class C, if a subgroup exists with classes from C's package, C is added to that subgroup. Remaining non-restricted classes form their own subgroup.

Synthetic Items Policy [Code] This policy separates synthetic classes from non-synthetic classes. It allows synthetic classes to be merged only with other synthetic classes, and non-synthetic classes only with non-synthetic classes.

Same Package For Non Global Merge Synthetic [Code] Synthetic classes that are not globally mergeable can only be merged with classes from the same package; such classes are considered *restricted*. Synthetic kinds indicate the reason a synthetic class was generated, and a synthetic class may have multiple synthetic kinds. A synthetic class A is restricted if none of its synthetic kinds are method kinds, or, if A has method kinds, all of them have the "allow merging globally" flag set to false. The group-splitting mechanism for these classes is analogous to that used in `RespectPackageBoundaries`.

Check Abstract Classes [Code] Classes are grouped according to whether they are abstract or not.

No Api Outline With Non-Api Outline [Code] Synthetic classes are generated by the compiler for various reasons. The "synthetic kind" represents the reason for generation and provides a hint about what the synthetic item represents. Classes are grouped based on whether they contain the `API_MODEL_OUTLINE` flag or not.

No Different Api Reference Level [Code] Classes are grouped according to their API reference level. This policy is activated only when *Library API Modeling* is enabled in the system configuration.

No Indirect Runtime Type Checks [Code] Classes that implement an interface with a runtime type check (directly or indirectly through a parent interface) are only merged with classes that implement the same interfaces.

Same Feature Split [Code] Classes can only be merged if they belong to the same feature split. Classes are grouped according to their feature split.

Same File Policy [Code] If enabled, this policy allows merging only for inner classes of the same outer class. The inner-class suffix “$¡InnerClassName¿” of the class descriptor is removed; for example, `Lcom/example/Foo$A` and `Lcom/example/Foo$B` are changed to `Lcom/example/Foo`. Only classes with identical edited class descriptors can be merged. The option can be enabled or disabled in the configuration.

Hint: Although in Kotlin multiple classes can be defined in the same file, this policy does not allow merging those classes. The name of the policy, `SameFilePolicy`, refers to the Java convention, where only one top-level class (excluding nested or anonymous classes) is allowed per file.

This policy is activated only when *same file policy* is enabled in the system configuration.

Same Instance Fields [Code] Classes can be merged only if they have the same multiset of relaxed instance field descriptions. The relaxed instance field description includes the following attributes:

- Access flags: public, private, protected, static, volatile, transient, enum (ignoring final and synthetic).
- Types for reference fields (class types and array types): all types are treated equally as `Object`.
- Types for non-reference fields (primitive types) are treated exactly.

Classes A and B can be merged if and only if they include the same number of every relaxed datatype description. For example, consider the following type declarations for classes A and B:

```
class A {
    public CustomClass a;
    public String b;
    public static double c;
}

class B{
    private String[] d;
    public int[] e;
    public double f;
}
```

The multiset of A is $\{$public Object: 2, public static double: 1$\}$ and of B is $\{$private Object: 1, public Object: 1, public double: 1$\}$, so the classes cannot be merged.

Same Main Dex-Group [Code] Two classes can only be merged if they belong to the same dex group. If one of the classes belongs to the main-dex group, they cannot be merged. This policy is activated only when *Main-Dex* information are available (can be configured using `---main-dex-list <file>` argument).

Same Nest Host [Code] If a class A is in a nest (i.e., A is a nested class or hosts nested classes), it can only be merged with classes in the same nest.

Same Parent Class [Code] Classes A and B can only be merged if they share the same parent class; that is, they are siblings in the class hierarchy.

Some policies are not executed to enforce constraints but to verify assumptions about the state resulting from previously executed policies. These policies are typically executed only in test environments. While such asserted policies are not relevant in production, they document the implicit behavior of preceding policies, which is not always obvious. For example, D8 executes only two single-class policies. Nevertheless, these two are sufficient to satisfy all the rules required by the remaining asserted single-class policies.

During the analysis, we discovered a policy that led to unexpected class merging behavior. The policy *NoClassAnnotationCollisions* enforces that, within each group, at most one class may have a class annotation. For every class that carries a class annotation, the policy creates a new subgroup. All remaining classes are then distributed across these subgroups using a round-robin strategy. Since this policy is executed early in the pipeline, many classes are distributed into different groups, which may prevent them from being merged. The following example illustrates this:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {}

public class GroupA {}
public class A1 extends GroupA { ... }
public class A2 extends GroupA { ... }

public class GroupB {}
@MyAnnotation public class B1 { ... }
@MyAnnotation public class B2 { ... }
```

Before applying *NoClassAnnotationCollisions*, the merge group is: [A1, A2, B1, B2, GroupA, GroupB]. The annotation class is already filtered out at this point (see the *NoAnnotationClasses* policy). The ordering is determined lexicographically by the class name.

When applying the *NoClassAnnotationCollisions* policy, new groups are created for the classes with annotations (Round 1). The other classes are added cyclically to the groups in the order defined by the existing merge group (Round 2&3):

- Round 1: [B1], [B2]
- Round 2: [B1, A1], [B2, A2]
- Round 3: [B1, A1, GroupA], [B2, A2, GroupB]

Since A1 and A2 end up in different groups, they will never be merged. However, if A2 is renamed to `GroupA_Child2`, the ordering changes to [A1, B1, B2, GroupA, GroupA_Child2, GroupB]. This results in the following merge groups:

$$[B1, A1, GroupA\_Child2] \quad \text{and} \quad [B2, GroupA, GroupB]$$

In this second scenario, A1 and `GroupA_Child2` are in the same merge group and will eventually be merged. This shows that whether two classes are merged can depend solely on how they are renamed. Adding, renaming, or removing classes may influence the outcome of class merging.

To inspect the created merge groups, add the following snippet immediately before the return statement of the `apply` method in the `HorizontalClassMergerPolicyExecutor` class:

```
System.out.println("---- " + policy.getName() + " ----");
linkedGroups.forEach(group -> System.out.println(group.toString()));
System.out.println("-------------");
```

Ensure that classes are not removed due to inlining. To prevent this in an experimental setup, add keep rules in the ProGuard configuration, for example: `-keep class package.className { *; }`. Then, disable the `NoKeepRule` and `NotMatchedByNoHorizontalClassMerging` policies, for instance by modifying the *canMerge* method to always return `true`. These modifications allow us to use keep rules to observe the behavior of the policies without affecting the behavior of the code under analysis. Alternatively, instead of disabling the two policies, inlining can be prevented by creating classes that are sufficiently complex to avoid being inlined.

Finally, add a rule to the ProGuard configuration to prevent the custom annotation from being stripped: `-keepattributes RuntimeVisibleAnnotations,RuntimeInvisibleAnnotations`.

Table A.1: Policies of R8. Policies marked with stars are handled as assertions when merging is restricted to synthetic classes

Only R8	Policy Name	Requires
	Single Class Policies	
	CheckSyntheticClasses	
X	NoCheckDiscard	
X	NoKeepRules	
X	NoClassInitializerWithObservableSideEffects	
X	NoMethodHandleFromLambda	Application has liveness
	NoResourceClasses*	
	NoAnnotationClasses*	
	NoDirectRuntimeTypeChecks*	
X	NoEnums*	
	NoInterfaces*	Interface Merging disabled
	NoInnerClasses*	
	NoInstanceFieldAnnotations*	
	NoKotlinMetadata*	
	NoNativeMethods*	
X	NoServiceLoaders*	
	NoRecords*	
	When *not* restricted to synthetics	
X	NotMatchedByNoHorizontalClassMerging	
X	NoFailedResolutionTargets	
	Multi Class Policies	
X	SameFilePolicy	Same File Policy enabled
	CheckAbstractClasses	
X	NoClassAnnotationCollisions	
	SameFeatureSplit	
X	SameStartupPartition	Startup Boundary Optimization disabled
X	SameInstanceFields	
	SameMainDexGroup	MainDex Information available
	SameNestHost	
	SameParentClass	
	SyntheticItemsPolicy	
X	RespectPackageBoundaries	
	NoDifferentApiReferenceLevel	Library API Modeling enabled
X	NoIndirectRuntimeTypeChecks	
X	NoWeakerAccessPrivileges	
X	PreventClassMethodAndDefaultMethodCollisions	
X	PreserveInterfaceMethodDispatch	API Level supports Default/Static Interface Methods
X	NotTwoInitsWithMonitors	Specific ART VM version (not restricted to synthetics)
X	MinimizeInstanceFieldCasts	
X	NoDefaultInterfaceMethodMerging	
X	NoDefaultInterfaceMethodCollisions	Interface Merging enabled
X	LimitInterfaceGroups	
X	OnlyDirectlyConnectedOrUnrelatedInterfaces	Interface Merging enabled
X	PreserveMethodCharacteristics	Application has liveness
	LimitClassGroups	
	FinalizeMergeGroup	
	When *not* restricted to synthetics	
X	NoClassInitializerCycles	
X	NoDeadLocks	

Table A.2: Policies of D8

only D8	Policy Name	Requires
	Single Class Policies	
	CheckSyntheticClasses	
X	OnlyClassesWithStaticDefinitionsAndNoClassInitializer	
	Assertions	
	NoResourceClasses	
	NoAnnotationClasses	
	NoDirectRuntimeTypeChecks	
	NoInterfaces	Interface Merging disabled
	NoInnerClasses	
	NoInstanceFieldAnnotations	
	NoKotlinMetadata	
	NoNativeMethods	
	NoRecords	
	Multi Class Policies	
X	SamePartialSubCompilation	Partial Sub Compilation Conf
	CheckAbstractClasses	
	SameFeatureSplit	
	SameMainDexGroup	MainDex information available
	SameNestHost	
	SameParentClass	
	SyntheticItemsPolicy	
X	NoApiOutlineWithNonApiOutline	
X	SamePackageForNonGlobalMergeSynthetic	
	NoDifferentApiReferenceLevel	
	LimitClassGroups	
	FinalizeMergeGroup	

Table A.3: Policies used exclusively by D8 or R8

Policy Name	Type	Requires
Exclusive D8 Policies		
OnlyClassesWithStaticDefinitionsAndNoClassInitializer	Single Class	
SamePartialSubCompilation	Multi Class	Partial Sub Compilation Conf.
NoApiOutlineWithNonApiOutline	Multi Class	
SamePackageForNonGlobalMergeSynthetic	Multi Class	
Exclusive R8 Policies		
NoCheckDiscard	Single Class	
NoKeepRules	Single Class	
NoClassInitializerWithObservableSideEffects	Single Class	
NoMethodHandleFromLambda	Single Class	Application has liveness
NoEnums	Single Class	
NoServiceLoaders	Single Class	
NotMatchedByNoHorizontalClassMerging	Single Class	Not restricted to Synthetics
NoFailedResolutionTargets	Single Class	Not restricted to Synthetics
SameFilePolicy	Multi Class	Same File Policy enabled
NoClassAnnotationCollisions	Multi Class	
SameStartupPartition	Multi Class	Startup Boundary Optimizsation disabled
SameInstanceFields	Multi Class	
RespectPackageBoundaries	Multi Class	
NoIndirectRuntimeTypeChecks	Multi Class	
NoWeakerAccessPrivileges	Multi Class	
PreventClassMethodAndDefaultMethodCollisions	Multi Class	
PreserveInterfaceMethodDispatch	Multi Class	API Level supports Default-/Static Interface Methods
NotTwoInitsWithMonitors	Multi Class	Specific ART VM version
NoClassInitializerCycles	Multi Class	Not restricted to Synthetics
NoDeadLocks	Multi Class	Not restricted to Synthetics
MinimizeInstanceFieldCasts	Multi Class	
NoDefaultInterfaceMethodMerging	Multi Class	
NoDefaultInterfaceMethodCollisions	Multi Class	Interface Merging enabled
LimitInterfaceGroups	Multi Class	
OnlyDirectlyConnectedOrUnrelatedInterfaces	Multi Class	Interface Merging enabled
PreserveMethodCharacteristics	Multi Class	Application has liveness

B
Setup Device

This section describes the steps required to prepare an Android device for dynamic analysis. The procedure includes configuring ADB access, unlocking the bootloader, and patching the boot image using Magisk.

ADB Initialization Connect the Android device to the host computer via USB and ensure that the Android Debug Bridge (ADB) is installed. To verify that the device is detected, list connected USB devices:

```
lsusb
```

An example output is shown below:

```
Bus 001 Device 003: ID 1bbb:0c01 T & A Mobile Phones TCL 50 5G
```

In this example, `1bbb` represents the vendor ID of the device. To allow non-root access to the device via ADB, create a udev rule:

```
sudo nano /etc/udev/rules.d/51-android.rules
```

Add the following line, replacing the vendor ID if necessary:

```
SUBSYSTEM=="usb", ATTR{idVendor}=="1bbb", MODE="0666", GROUP="plugdev"
```

Apply the changes and restart the ADB service:

```
sudo chmod a+r /etc/udev/rules.d/51-android.rules
sudo udevadm control --reload-rules
sudo udevadm trigger
adb kill-server
adb start-server
adb devices
```

Ensure that the current Linux user is a member of the `plugdev` group.

Bootloader Unlocking and Boot Image Patching Enable *Developer Options* on the device and activate *OEM unlocking*. Reboot the device into the bootloader and unlock it:

```
adb reboot bootloader
fastboot flashing unlock
```

Unlocking the bootloader will typically erase all user data.

The `boot.img` file must correspond exactly to the firmware version currently installed on the device. It can be obtained from the official factory image or firmware package provided by the device manufacturer. For Google Pixel devices, factory images are publicly available via the official Android developer website. After downloading the appropriate factory image archive, extract its contents and locate the included `boot.img` file.

Magisk must be downloaded from its official GitHub repository and installed on the device prior to patching the boot image. Use the Magisk application to patch the extracted `boot.img`. Once patched, transfer the resulting image back to the host system.

Reboot the device into the bootloader again:

```
adb reboot bootloader
```

For testing purposes, the patched image can be booted temporarily:

```
fastboot boot <patched-boot.img>
```

For permanent installation, flash the patched image:

```
fastboot flash boot <patched-boot.img>
```

Using a `boot.img` that does not match the installed system version may result in boot failures or unstable behavior.

Following these steps, root access is available on the device.

C
Code References

C.1 Vertical Merging Policies

- **No Abstract Methods On Abstract Classes Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoAbstractMethodsOnAbstractClassesPolicy.java
- **No Annotation Classes Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoAnnotationClassesPolicy.java
- **No Class Initialization Changes Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoClassInitializationChangesPolicy.java
- **No Directly Instantiated Classes Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoDirectlyInstantiatedClassesPolicy.java
- **No Enclosing Method Attributes Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoEnclosingMethodAttributesPolicy.java
- **No Field Resolution Changes Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoFieldResolutionChangesPolicy.java

- **No Illegal Accesses Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoIllegalAccessesPolicy.java
- **No Inner Class Attributes Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoInnerClassAttributesPolicy.java
- **No Interfaces With Invoke Special To Default Method Into Class Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoInterfacesWithInvokeSpecialToDefaultMethodIntoClassPolicy.java
- **No Interfaces With Unknown Subtypes Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoInterfacesWithUnknownSubtypesPolicy.java
- **No Invoke Super No Such Method Errors Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoInvokeSuperNoSuchMethodErrorsPolicy.java
- **No Kept Classes Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoKeptClassesPolicy.java
- **No Lock Merging Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoLockMergingPolicy.java
- **No Method Resolution Changes Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoMethodResolutionChangesPolicy.java
- **No Nested Merging Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoNestedMergingPolicy.java
- **No Non Serializable Class Into Serializable Class Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoNonSerializableClassIntoSerializableClassPolicy.java
- **No Service Interfaces Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce2

3dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/NoServiceInterfacesPolicy.java

- **Same Api Reference Level Policy**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/SameApiReferenceLevelPolicy.java
- **Same Feature Split Policy**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/SameFeatureSplitPolicy.java
- **Same Main Dex Group Policy**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/SameMainDexGroupPolicy.java
- **Same Nest Policy**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/SameNestPolicy.java
- **Same Startup Partition Policy**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/SameStartupPartitionPolicy.java
- **Successful Virtual Method Resolution In Target Policy**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/verticalclassmerging/policies/SuccessfulVirtualMethodResolutionInTargetPolicy.java

C.2 Horizontal Merging Policies

- **Check Abstract Classes**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/CheckAbstractClasses.java
- **Check Synthetic Classes**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/CheckSyntheticClasses.java
- **Limit Class Groups**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/LimitClassGroups.java

- **Limit Interface Groups**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/LimitInterfaceGroups.java
- **Minimize Instance Field Casts**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/MinimizeInstanceFieldCasts.java
- **No Annotation Classes**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoAnnotationClasses.java
- **No Api Outline With Non Api Outline**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoApiOutlineWithNonApiOutline.java
- **No Check Discard**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoCheckDiscard.java
- **No Class Annotation Collisions**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoClassAnnotationCollisions.java
- **No Class Initializer Cycles**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoClassInitializerCycles.java
- **No Class Initializer With Observable Side Effects**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoClassInitializerWithObservableSideEffects.java
- **No Dead Locks**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoDeadLocks.java
- **No Default Interface Method Collisions**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoDefaultInterfaceMethodCollisions.java
- **No Default Interface Method Merging**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoDefaultInterfaceMethodMerging.java

- **No Different Api Reference Level**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoDifferentApiReferenceLevel.java
- **No Direct Runtime Type Checks**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoDirectRuntimeTypeChecks.java
- **No Enums**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoEnums.java
- **No Failed Resolution Targets**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoFailedResolutionTargets.java
- **No Indirect Runtime Type Checks**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoIndirectRuntimeTypeChecks.java
- **No Inner Classes**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoInnerClasses.java
- **No Instance Field Annotations**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoInstanceFieldAnnotations.java
- **No Interfaces**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoInterfaces.java
- **No Keep Rules**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoKeepRules.java
- **No Kotlin Metadata**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoKotlinMetadata.java
- **No Method Handle From Lambda**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoMethodHandleFromLambda.java

- **No Native Methods**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoNativeMethods.java

- **No Records**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoRecords.java

- **No Resource Classes**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoResourceClasses.java

- **No Service Loaders**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoServiceLoaders.java

- **Not Matched By No Horizontal Class Merging**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NotMatchedByNoHorizontalClassMerging.java

- **Not Two Inits With Monitors**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NotTwoInitsWithMonitors.java

- **No Weaker Access Privileges**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/NoWeakerAccessPrivileges.java

- **Only Classes With Static Definitions And No Class Initializer**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/OnlyClassesWithStaticDefinitionsAndNoClassInitializer.java

- **Only Directly Connected Or Unrelated Interfaces**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/OnlyDirectlyConnectedOrUnrelatedInterfaces.java

- **Preserve Interface Method Dispatch**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/PreserveInterfaceMethodDispatch.java

- **Preserve Method Characteristics**
https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce2

3dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/PreserveMethodCharacteristics.java

- **Prevent Class Method And Default Method Collisions**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/PreventClassMethodAndDefaultMethodCollisions.java
- **Respect Package Boundaries**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/RespectPackageBoundaries.java
- **Same Feature Split**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/SameFeatureSplit.java
- **Same File Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/SameFilePolicy.java
- **Same Instance Fields**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/SameInstanceFields.java
- **Same Main Dex Group**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/SameMainDexGroup.java
- **Same Nest Host**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/SameNestHost.java
- **Same Package For Non Global Merge Synthetic**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/SamePackageForNonGlobalMergeSynthetic.java
- **Same Parent Class**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/SameParentClass.java
- **Same Partial Sub Compilation**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/SamePartialSubCompilation.java

- **Same Startup Partition**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/SameStartupPartition.java
- **Synthetic Items Policy**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/SyntheticItemsPolicy.java

C.3 Other R8 Classes

- **Synthetic Naming**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/synthesis/SyntheticNaming.java
- **Horizontal Class Merger Policy Executor**
  https://r8.googlesource.com/r8/+/6676a2e8a7ca99f9a47d556495bbce23dfe70c48/src/main/java/com/android/tools/r8/horizontalclassmerging/HorizontalClassMergerPolicyExecutor.java

C.4 Android Runtime

- **DEX File Version 40** (Android 15)
  https://android.googlesource.com/platform/art/+/refs/heads/android15-release/libdexfile/dex/dex_file.h
- **DEX File Version 41** (Android 16)
  https://android.googlesource.com/platform/art/+/refs/heads/android16-release/libdexfile/dex/dex_file.h

C.5 Libcore

- **Dex Class Loader**
  https://android.googlesource.com/platform/libcore/+/de876a01b29230b877c9f408348c38a90ee724a5/dalvik/src/main/java/dalvik/system/DexClassLoader.java
- **Dex Path List**
  https://android.googlesource.com/platform/libcore/+/de876a01b29230b877c9f408348c38a90ee724a5/dalvik/src/main/java/dalvik/system/DexPathList.java

Disclaimer

In the interest of transparency, we acknowledge the use of large language model (LLM) technology, such as ChatGPT, to optimize text passages and assist in code generation for this thesis. While the research, analysis, and intellectual contributions are our own, we utilized LLM assistance to refine phrasing, enhance clarity, improve readability, and translate conceptual ideas into functional code. All key decisions regarding content, structure, argumentation, and methodology remain solely our responsibility.

Erklärung

gemäss Art. 30 RSL Phil.-nat.18

Name/Vorname:	Kaufmann Dario
Matrikelnummer:	19-268-218
Studiengang:	MSc in Computer Science

Bachelor ☐ Master ☒ Dissertation ☐

Titel der Arbeit:	Towards Enhanced Android Third-Party Library Detection through Runtime-Assisted Static Analysis
LeiterIn der Arbeit:	Thomas Sutter, Prof. Dr. Timo Kehrer

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.
Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Ort/Datum 28.01.2026

[illegible]

Unterschrift