An Empirical Study on the Robustness of Android Third-Party Library Detection Tools Against Advanced Obfuscation

Dahan Pan
Shanghai Jiao Tong University
Shanghai, China
dhpan98@sjtu.edu.cn

Zhuohao Zhang
Gosec Research Group
Shanghai, China
mm0n57er@gmail.com

Yunjia Min
Shanghai Jiao Tong University
Shanghai, China
minyunjia@sjtu.edu.cn

Runhan Feng
Purple Mountain Laboratories
Nanjing, Jiangsu, China
fengrunhan@pmlabs.com.cn

Yuanyuan Zhang
Shanghai Jiao Tong University
Shanghai, China
yyjess@sjtu.edu.cn

Abstract

Third-party libraries (TPLs) play a crucial role in Android app development by providing reusable functionalities, improving development efficiency, and reducing time-to-market. However, detecting and analyzing TPLs is essential, as their vulnerabilities, outdated versions, or malicious modifications can introduce security risks and compromise the integrity of Android apps. Existing TPL detection approaches struggle against code obfuscation, a prevalent practice in Android apps. While prior research has explored obfuscation-resistant detection methods, they largely overlook advanced obfuscation techniques such as package hierarchy obfuscation.

To bridge this gap, we first investigate the prevalence of advanced obfuscation in contemporary Android apps using ObfDetector, a novel static analysis tool capable of detecting identifier renaming, package hierarchy obfuscation, and code optimization. Our large-scale study on 77,504 closed-source Google Play apps and 619 widely used apps reveals extensive obfuscation adoption. We further evaluate the resilience of state-of-the-art TPL detection tools under realistic obfuscation conditions using a newly constructed benchmark dataset, exposing significant performance deficiencies, with F1-scores dropping below 50% for library-level detection and under 10% for version-level identification. Finally, we conduct a systematic failure analysis to uncover key architectural limitations in existing TPL detection frameworks and propose design guidelines for next-generation detection tools. We release our obfuscation detection tool and benchmark dataset to support further research in Android security.

1 Introduction

The past decade has witnessed an exponentially rapid development of the Android app market. As of 2024, over 2.6 million apps are hosted on the Google Play Store, the primary marketplace for Android apps [1]. Third-party libraries significantly facilitate this process by enabling developers to integrate a wide range of functionalities easily, including advertising, social media, and data analytics. However, the widespread use of TPLs poses security risks, as their internal implementation details is beyond developers' control. Malicious TPLs may surreptitiously collect users' data, threatening end-user privacy. Moreover, vulnerabilities within these libraries can affect the apps that integrate them, leading to the propagation of security threats [18, 23]. Therefore, extensive research has been carried out on the mitigations against the threats caused by TPLs, such as isolating TPLs from apps [30] and patching outdated versions [11], etc. However, these mitigations are often built on the premise of precise detection of TPLs from app binaries, which is also the focus of existing academic literature. [8, 38, 40, 42, 47]

Considering protecting intellectual property rights against reverse engineering and reducing app size, etc., many developers have resorted to code obfuscation techniques. This aligns with Google's official recommendations [2] and has become a prevalent practice in Android app development [17]. Code obfuscation tools such as ProGuard [4] and R8 [3], have been integrated into Android Studio, the most popular Android development environment for ease of use. Dominik et al. show that roughly 50% of the most popular apps with over 10 million downloads are obfuscated before being released [36].

Code obfuscation in Android apps presents a significant challenge for TPL detection, as it removes many distinguishable features in the code that could otherwise be used for matching and identifying TPLs. In recent years, the emergence of advanced obfuscation techniques, including package hierarchy obfuscation and optimization-oriented code transformation, has further exacerbated this challenge. However, the extent to which these advanced obfuscation techniques are deployed in real-world Android apps remains unclear. The most recent large-scale analysis of code obfuscation in real-world Android apps did not perform an in-depth analysis of advanced code obfuscation techniques such as package hierarchy obfuscation [17, 36], diminish their practical relevance given the rapid evolution of Android app ecosystem. To bridge this gap in the literature, in this study, we pose the following research question:

- **RQ1**. What is the prevalence of advanced obfuscation techniques in real-world Android apps?

To answer this research question, it is essential to detect the specific obfuscation schemes employed by apps, which remains challenging due to the absence of significant indicator. Through an in-depth manual analysis of real-world obfuscated apps, we derived a set of heuristics rules for identifying the obfuscation schemes in Android apps. Leveraging these heuristics, we developed ObfDetector, a novel static analysis tool capable of detecting *identifier renaming*, *package hierarchy obfuscation*, and *code optimization* in Android apps. Comprehensive evaluation on both open-source and closed-source datasets demonstrated ObfDetector 's effectiveness, consistently achieving high detection accuracy, with F1-scores exceeding 85% across all target obfuscation schemes. We further performed a large-scale obfuscation analysis, covering 77,504 closed-source Google Play apps and 619 popular apps. The result revealed a widespread adoption of advanced obfuscation in both app categories. This phenomenon naturally leads to our second research question:

- **RQ2**. How do state-of-the-art TPL detection tools perform against advanced code obfuscation in terms of detection effectiveness and efficiency?

To address the challenge of code obfuscation, existing studies mainly utilize similarity-based methods [38, 40, 42, 44, 47], extracting code features resistant to code obfuscation, to match Android apps with TPLs. However, the evaluation of existing studies is faced with one problem. These tools are typically evaluated using an outdated dataset [35], which fail to effectively capture the emerging advanced obfuscation techniques. Additionally, the lack of a standardized benchmark dataset hinders systematic comparisons of these detection tools. Therefore, we construct a benchmark dataset that incorporates different obfuscation schemes. Based on the benchmark dataset, we empirically evaluated the robustness of existing TPL detection tools. Our evaluation reveals critical performance deficiencies in state-of-the-art detection tools, with library-level identification achieving less than 50% F1-scores and version-level detection attaining less than 10% F1-scores (which will be detailed in Section 4). These findings necessitate our third research question:

- **RQ3**. What factors result in the poor performance of existing TPL detection tools, and how are these factors distributed among the detection failures observed in practise?

To answer this question, we conduct an in-depth manual analysis on the failure of the tools and identified four deficiencies in current detection paradigms. Based on these findings, we provide corresponding design guidelines for the development of future TPL detection tools.

Overall, our contributions can be summarized as follows:

- We design, implement, and evaluate ObfDetector, an advanced Android obfuscation detection tool capable of identifying three major contemporary code obfuscation techniques. We conduct large-scale experiments on both a closed-source app dataset and a widely used app dataset to assess its effectiveness.
- We systematically evaluate state-of-the-art Android TPL detection tools in terms of detection accuracy and runtime efficiency using our self-constructed benchmark dataset, which closely reflects real-world Android apps. Furthermore, we perform an in-depth analysis of these tools and proposed design guidelines to inform future research.
- We will open-source ObfDetector and our benchmark dataset for TPL detection evaluation, facilitate future advancements in Android TPL research. The source code of ObfDetector and benchmark datasets will be available at https://doi.org/10.5281/zenodo.16126235.

The rest of the paper is organized as follows. We first introduce the background in Section 2. We then describe our large-scale analysis of obfuscation in the Section 3. The empirical results are presented in Section 4, followed by a discussion in Section 5. We introduce related work in Section 6 and conclude our paper in Section 7.

2 Background

Android code obfuscation tools play a pivotal role in the Android software development lifecycle by enabling developers to produce applications that are both compact and efficient. A widely adopted feature in the Android Gradle Plugin (AGP) [13], the `minify` option, allows developers to integrate *shrinking*, *obfuscation*, and *optimization* strategies into their build processes. Specifically, the *shrinking* strategy eliminates unused code, the *obfuscation* strategy shortens class and member names, and the *optimization* strategy refines code structures to further reduce app size and enhance runtime performance. These strategies collectively empower developers to create lightweight and high-performance Android applications.

However, while these techniques improve app efficiency, they also introduce significant challenges for third-party library (TPL) detection, a critical task in security analysis. TPL detection tools rely on explicit semantic information, package hierarchies, and fine-grained code patterns to identify vulnerable libraries, outdated versions, or malicious modifications within an app. The transformations introduced by obfuscation and optimization schemes obscure these features, making it difficult to perform accurate analysis.

In recent years, Android R8, developed by Google, has emerged as the dominant tool for obfuscation and optimization in Android app development. Starting with AGP version 3.4.0 (and continuing through the latest version, 8.9), the Android Gradle Plugin exclusively supports Android D8/R8 as the default compiler. Empirical studies [40] analyzing the open-source Android app dataset F-Droid reveal that among 2,347 apps, 1,380 (58.8%) utilize D8, 964 (41.1%) use R8, and only three apps rely on alternative compilers. Although discrepancies may exist between open-source datasets and real-world app markets, it is evident that Android D8 and R8 have become the *de facto* standards for app compilation, obfuscation, and optimization.

Given this context, we summarize three key ways in which Android R8 modifies the original code features during the compilation process.

Identifier Renaming. Identifier renaming transforms package names, class names, method names, and variable names into semantically meaningless strings, thereby removing explicit semantic information without altering the program's functionality. This technique is nearly ubiquitous across Android obfuscation tools, making it a critical indicator of whether an app has undergone obfuscation. By stripping away meaningful identifiers, this approach undermines the effectiveness of TPL detection tools that depend on explicit semantic information for analysis.

Package Hierarchy Obfuscation. Package hierarchy obfuscation has gained increasing adoption in recent years, with mainstream tools like ProGuard and R8 offering robust support for this technique. It encompasses two primary modes: package flattening and repackaging. In ProGuard's configuration (which R8 inherits), the `-flattenpackagehierarchy` option relocates a specified package and its contents to its parent package, while the `-repackageclasses` option moves all leaf nodes within a package to a designated target package. These transformations disrupt the original hierarchical relationships between packages and classes, rendering TPL detection tools that rely on package hierarchy information ineffective.

Code Optimization. Code optimization is an intrinsic feature of modern Android compilers, enabling instruction-level refinements to enhance runtime performance. Common optimization strategies include *DeadCodeRemoval*, among others. Notably, Android R8 introduces additional optimization schemes. According to prior research [40], R8 implements 13 advanced optimization strategies, with *Inlining* and *CallSiteOptimization* having the most significant impact on TPL detection. These optimizations modify code at the instruction level, introducing changes that can lead to false negatives and false positives in TPL detection methods reliant on fine-grained code characteristics.

While these obfuscation and optimization strategies enhance app performance, they also compromise the accuracy of Android TPL detection tools. However, the prevalence of these techniques—particularly *Package Hierarchy Obfuscation* and *Code Optimization*—in real-world app markets remains unclear. *Package Hierarchy Obfuscation* is a relatively novel technique, and no comprehensive academic studies have yet quantified its adoption. Similarly, *Code Optimization* often involves subtle, fine-grained modifications that are difficult to detect and analyze. Consequently, understanding the extent and impact of these obfuscation schemes in practice represents a critical focus of our research.

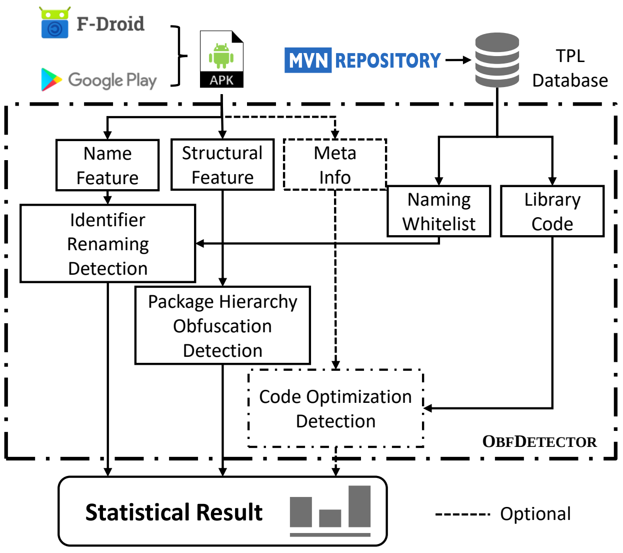


Figure 1: The Detection Process of ObfDetector

3 Code Obfsucation Detection

In this section, we address the first key research question, **RQ1**, formulated in our study. Since the emergence of Android R8, existing studies on obfuscation have become increasingly outdated, and there has been limited academic research investigating the prevalence of obfuscation in real-world applications.

To bridge this gap, we developed an Android obfuscation detection tool, ObfDetector , capable of effectively identifying three major obfuscation techniques: *identifier renaming*, *package hierarchy obfuscation*, and *code optimization*. The latter two can be considered as advanced obfuscation techniques. Other obfuscation techniques, such as control flow flattening and code injection, are primarily adopted by niche or commercial obfuscators, and are not widely observed in mainstream applications. As such, they fall outside the scope of this paper. We systematically evaluated the detection effectiveness of ObfDetector on both open-source and closed-source Android app datasets. Furthermore, leveraging ObfDetector , we conducted a large-scale analysis of applications from Google Play and other widely used datasets to investigate the real-world adoption of obfuscation techniques in Android applications.

3.1 Design of ObfDetector

As illustrated in Figure 1, ObfDetector first extracts relevant information from the APK and then performs identifier renaming detection, package hierarchy obfuscation detection, and code optimization detection. These analyses are conducted using a whitelist-based approach and code-related information extracted from our constructed third-party library (TPL) dataset, which was collected from the Maven repository [14]. Given that certain obfuscation techniques exhibit subtle characteristics that are challenging to detect directly, we have designed a series of heuristic methods to enhance detection accuracy.

3.1.1 Identifier Renaming Detection. Building upon previous research, we developed an identifier renaming detection approach based on a whitelist mechanism, which operates according to four primary rules: 1) Identifiers (including package names, class names, and function names) with lengths less than or equal to 2 are considered obfuscated characters; 2) Identifiers with lengths between 2 and 5 are detected using a whitelist-based strategy; 3) Identifiers longer than 5 are assumed to be unobfuscated; 4) If the proportion of identifiers classified as obfuscated exceeds 30%, we consider the app to have undergone identifier renaming obfuscation.

The boundary settings, 2 and 5, are informed by prior work [19] and align with our observations in real-world applications, where obfuscated class names are typically short and rarely exceed five characters (e.g., aa, ab, aaa). Building on this, the whitelist employed in Rule 2 is derived from a word list we constructed specifically for this purpose. We used a web crawler to collect all TPLs from the Maven [14] repository and extracted their *artifactId* and *groupId* fields. All words from these fields were aggregated into a comprehensive word list containing 83,644 unique entries, of which 6,372 words are between two and five characters in length. For identifiers falling within this length range, we exclude those present in our word list and classify the remaining ones as obfuscated. Our rationale is that whitelisted words reflect common developer naming practices, making it unlikely that identifiers matching these entries are the result of obfuscation. The 30% threshold mentioned in Rule 4 was determined through manual inspection and empirical tuning on a small, self-constructed dataset. This value was selected as it offered a reasonable trade-off between precision and recall in our preliminary experiments.

3.1.2 Package Hierarchy Obfuscation Detection. Since package hierarchy obfuscation exhibits less apparent characteristics than identifier renaming, we designed a heuristic method comprising three steps to detect potential obfuscation in Android applications:

Step 1 We scanned all the TPLs from our TPL dataset, counted the number of classes under each package, and recorded the package names where the class count exceeded the threshold θ.

Step 2 We sorted the packages in the APK to be detected based on the number of classes under each package. After excluding the package names from the list obtained in the *Step 1*, we recorded the top three package names and their respective class counts.

Step 3 We analyzed the top three package names based on class count rankings and determined whether the app had undergone package hierarchy obfuscation based on their hierarchical levels.

For the threshold θ in Step 1, we observed that detection performance remained stable across a broad range (0.05 to 0.75), showing low sensitivity to minor variations. Therefore, we selected 0.4 — the median of this range — as the optimal value. The purpose of constructing a package list in Steps 1 and 2 is to prevent misclassification caused by obfuscation features inherent in TPLs. Through our practical evaluations, we observed that most false positives in package hierarchy obfuscation detection occurred when an unobfuscated APK incorporated an obfuscated TPL, leading the obfuscation characteristics of the library to be mistakenly attributed to the app itself. However, in TPL detection tasks, this scenario does not affect detection outcomes. In Step 3, our experimental results indicated that the best detection performance was achieved when an app was classified as obfuscated if its top-ranked package was at hierarchical level 1. This approach effectively differentiates Android applications influenced by package hierarchy obfuscation in TPL-related tasks.

3.1.3 Code Optimization Detection. Detecting whether an Android APK has undergone optimization based solely on bytecode analysis is inherently challenging, as code optimization does not introduce distinctive features. Due to the diversity of coding practices, it is often difficult to directly infer whether a given code segment has been optimized.

Through our investigation of Android APKs, we made a key observation: certain APKs contain files ending with `version` within the `META-INF` directory. These files typically list the names and specific versions of libraries used within the APK. Since these libraries invoke official Android APIs, their names and version information are preserved for app signature verification. Further analysis revealed that such information is present in most apps, independent of whether they have undergone obfuscation. We analyzed 10,523 applications released after January 2023, sourced from Google Play and downloaded via the AndroZoo dataset. Our analysis revealed that over 60% of these apps include files containing package name and version information. By leveraging this metadata, ObfDetector can trace the original source code of the corresponding libraries and their specific versions, enabling instruction-level comparisons to determine whether optimization has been applied.

The detection of whether an app has undergone code optimization follows the process. For a specific library l included in an APK named with l_{apk}, we first retrieve its corresponding original source code from a TPL database and compile it using D8 to generate the DEX file l_{lib}. Then, we identify methods within classes that have identical names in both l_{apk} and l_{lib} as anchors and establish the mapping dict. We further analyze cross-references to expand this mapping as much as possible. This step is based on the assumption that obfuscated apps contain class names and function names that remain unobfuscated. This assumption is derived from previous work [19] as well as our large-scale survey of real-world apps. Finally, we detect code optimization by instruction-level comparation of these paired classes and methods.

3.2 Effectiveness Evaluation

3.2.1 Experimental Settings and Datasets. Our evaluation experiments on ObfDetector were conducted in a virtual machine configured with 256GB of RAM, running the Linux 5.4.0-200-generic kernel (Ubuntu 20.04 64-bit). The host machine was equipped with an Intel® Xeon® Gold 6330 CPU @ 2.00GHz. Our Android app obfuscation detection tool was implemented in `Python 3.9.0` and utilized `Androguard 4.1.0` to analyze *APK* and *DEX* files, extracting features for obfuscation detection.

To evaluate ObfDetector and assess its obfuscation detection capabilities, we employed two Android app datasets, D_{open} and D_{closed}, defined as follows:

- The Open-sourced App Dataset D_{open}. This dataset comprises 908 Android apps whose source code was obtained from the F-Droid open-source repository. Each app was compiled using F-Droid's automated build system [10].
- The Closed-sourced App Dataset D_{closed}. This dataset consists of 77,504 Android apps, encompassing all apps and their updates released on Google Play [22] since 2022.

We evaluated ObfDetector on both D_{open} and D_{closed}. For D_{open}, we derived ground-truth labels for obfuscation techniques by analyzing the *mapping.txt* file generated during the app compilation process. This method offers higher accuracy compared to the widely used approach in prior academic studies, which relies on parsing Gradle configuration files. For D_{closed}, being a closed-source dataset, we randomly selected 100 apps for manual labeling. Labeling the obfuscation mechanisms used in closed-source Android applications is inherently challenging due to the limited availability of reference information. To address this, three authors — each with over four years of experience in reverse engineering Android apps — participated in the labeling process. To ensure reliability, they adopted a cross-validation approach: each author independently labeled the apps, and discrepancies were resolved through discussion to maximize accuracy. Drawing on their expertise, they carefully analyzed the decompiled code of each app, considering indicators such as the presence of obfuscated classes in the main package, the package hierarchy structure, and other relevant features. To the best of our knowledge, this is the first publicly available labeled dataset specifically focused on obfuscation mechanisms in closed-source Android apps. We believe it offers a solid foundation for future research and can help advance progress in this area.

It is important to note that features indicative of code optimization are typically subtle and not readily observable from the decompiled artifacts. As a result, we did not attempt to label code optimization in the closed-source dataset, given the lack of explicit evidence to support reliable annotation.

3.2.2 *Evaluation Metrics and Results.*

In line with standard evaluation practices, we employed four metrics to assess the performance of ObfDetector: *Accuracy* $= \frac{TP+TN}{TP+TN+FP+FN}$, *Precision* $= \frac{TP}{TP+FP}$, *Recall* $= \frac{TP}{TP+FN}$, and *F1-score* $= \frac{2\times Recall\times Precision}{Recall+Precision}$, where *TP*, *TN*, *FP*, and *FN* denote the number of *true positive*, *true negative*, *false positive*, and *false negative* samples, respectively. Among these metrics, *Accuracy* quantifies the overall correctness of the predictions made by the tool. *Precision* and *Recall* evaluate the tool's performance from the perspectives of soundness and completeness, respectively. The *F1-score*, as the harmonic mean of *Precision* and *Recall*, provides a balanced assessment of the tool's effectiveness by considering both false positives and false negatives.

Table 1: Evaluation Results of Obfuscation Detection Tool

Open-Source Dataset D_{open}	Pre	Rec	F1	Acc
IR Detect	95.24%	86.29%	90.55%	91.76%
PHO Detect^{I}	78.57%	5.60%	10.45%	56.26%
PHO Detect^{II}	93.65%	63.87%	75.95	81.55%
PHO Detect^{I+II}	92.03%	64.63%	75.93%	81.32%
Opt Detect	75.51%	92.76%	83.25%	82.82%
Closed-Source Dataset D_{closed}	**Pre**	**Rec**	**F1**	**Acc**
IR Detect	90.16%	85.94%	88.00%	85.00%
PHO Detect^{I}	59.52%	65.79%	62.50%	70.00%
PHO Detect^{II}	100.00%	68.42%	81.25%	88.00%
PHO Detect^{I+II}	100.00%	63.16%	77.42%	86.00%

Abbreviations: IR = Identifier Renaming, PHO = Package Hierarchy Obfuscation, Opt = Optimization.

The detection results are presented in Table 1. Within the table, "PHO Detect^{I}" corresponds to detection results based on the class count per package rule, while "PHO Detect^{II}" is based on the package naming convention rule. Additionally, "PHO Detect^{I+II}" indicates the detection results where both rules are simultaneously satisfied.

For code optimization detection in D_{open}, as detailed in Section 3.1.3, our approach relies on library information stored in the META-INF folder of the *APK* files. Among the 908 analyzed *APK* files, 780 contained such library information, and therefore, the results in Table 1 pertain to this subset. We compiled the library's *JAR* and *AAR* files using D8 and matched the generated *DEX* files to the corresponding code within the *APK*.

For D_{closed}, since the obfuscation labels were manually annotated, determining whether code optimization was applied is inherently challenging. Consequently, we did not evaluate the effectiveness of code optimization detection on D_{closed}.

Regarding the distribution of positive and negative samples across the two datasets, in D_{open}, the proportions of apps exhibiting identifier renaming, package hierarchy obfuscation, and code optimization are 45.71%, 45.59%, and 46.17%, respectively. In D_{closed}, the proportions for identifier renaming and package hierarchy obfuscation are 64.00% and 38.00%, respectively. These figures indicate that the datasets are relatively balanced, which in turn supports the validity and credibility of the detection results obtained on them.

As shown in Table 1, ObfDetector demonstrates high effectiveness in detecting identifier renaming, package hierarchy obfuscation, and code optimization in Android apps. Specifically, among the detection rules for package hierarchy obfuscation, "PHO Detect^{II}" exhibited the best performance across both D_{open} and D_{closed}. Therefore, we adopted "PHO Detect^{II}" as the standard metric for assessing the prevalence of package hierarchy obfuscation techniques in real-world Android apps.

3.3 Large-scale Analysis

We conducted systematic obfuscation detection using ObfDetector on both the D_{closed} dataset (defined in Section 3.2.1) and the D_{pop} dataset comprising 619 top-ranked apps from APKPure's category-specific download charts [12]. Our analysis quantifies the adoption rates of three critical obfuscation schemes: identifier renaming, package hierarchy restructuring, and code optimization – all known to impact Android TPL detection accuracy.

Table 2: Ratio of Different Obfuscation Schemes in the Wild

Obfuscation Scheme	D_{closed}	D_{pop}
IR	45,270 / 77,504 (58.41%)	539 / 611 (88.22%)
PHO	22,539 / 77,504 (29.08%)	267 / 611 (43.71%)
Opt	15,768 / 38,742 (40.70%)	360 / 527 (68.31%)

Abbreviations: IR = Identifier Renaming, PHO = Package Hierarchy Obfuscation, Opt = Code Optimization.

3.3.1 *Identifier Renaming.*

As evidenced in Table 2, our longitudinal analysis of 77,504 post-2022 Google Play applications reveals identifier renaming adoption in 58.41% of cases (45,270 apps). The prevalence increases to 88.22% (539/611 apps) in the D_{pop} dataset. This represents a significant increase from the 43% baseline reported in 2018 [17], suggesting an annualized growth rate of 3.8% in adoption. The near-90% adoption among popular apps establishes identifier renaming as a *de facto* standard for commercial Android applications, necessitating robust handling in TPL detection systems.

3.3.2 *Package Hierarchy Obfuscation.*

Our package name-based heuristic detection method identifies package hierarchy obfuscation in 29.08% of D_{closed} (22,539/77,504 apps) and 43.71% of D_{pop} (267/611 apps). While less prevalent than identifier renaming, this technique shows substantially higher adoption in popular apps compared to the general application population.

3.3.3 *Code Optimization.*

In D_{closed} and D_{pop}, the proportion of apps containing library version information is 49.98% and 86.25%, respectively. Applying our library-specific detection framework (Section 3.1.3) to 38,742 Google Play apps and 527 popular apps reveals code optimization in 40.70% (15,768 apps) and 68.31% (360 apps) of cases. Focusing specifically on code inlining optimizations – known to impact TPL detection accuracy [40] – our results demonstrate significantly higher optimization adoption in popular applications.

Through further investigation, we found that the use of code optimization during Android app development is independent of the inclusion of *META-INF/version* files - these are controlled by separate build modules. Specifically, when using Gradle to build a project, code optimization is configured via the `buildTypes` module, while the inclusion of such metadata files is governed by the `packagingOptions` module. This independence indicates that the detection results obtained from the subset of apps containing version files can effectively reflect the overall code optimization practices across the broader app population.

3.3.4 Conclusion. Our experimental findings address the research gap in **RQ1** through three key observations:

(1) Current obfuscation adoption rates substantially exceed historical baselines, particularly in popular apps.
(2) Obfuscation prevalence shows positive correlation with application popularity.
(3) Combined obfuscation strategies are prevalent, with 73.4% D_{pop} apps using multiple schemes.

These conclusions establish code obfuscation as a critical factor for Android TPL detection systems and provide empirical justification for our evaluation dataset construction (Section 4).

4 Empirical Study

This section presents a dual-aspect evaluation framework assessing prominent Android third-party library (TPL) detection tools through two critical dimensions: (1) *detection effectiveness* and (2) *runtime efficiency*, addressing research questions **RQ2** and **RQ3**. The absence of standardized benchmarks capable of simulating real-world obfuscation patterns in Android applications necessitates our methodological construction of an evaluation dataset. We conduct a systematic evaluation of four state-of-the-art detection tools using this benchmark, which will be made publicly available to facilitate reproducibility and future research.

4.1 Environment Setup

All evaluations of Android TPL detection tools were conducted on a virtualized environment running Ubuntu 20.04 LTS (64-bit). The host system provided allocated access to 48 physical cores of an Intel(R) Xeon(R) Gold 6330 CPU (2.00GHz base frequency) and 256GB DDR4-3200 registered ECC memory. To ensure fairness in computational resource allocation during efficiency comparisons, tool executions were constrained to 20 CPU cores through explicit core pinning configuration.

4.1.1 Benchmark Dataset Construction. As concluded in Section 3.3, code obfuscation and code optimization are widely used in Android apps, thus it is necessary to construct corresponding benchmark dataset to evaluate Android TPL detection tools.

We obtained the source code of Android apps from the F-droid open-source repository, modified the Gradle configuration files based on whether the code was obfuscated or optimized, and then compiled three different obfuscation mode datasets to assess the detection capabilities of existing Android TPL detection tools. The three datasets D_{non}, D_{obf}, and D_{obf}^{opt} contain 535, 495, and 484 APKs, respectively. The subscript/superscript notation indicates the obfuscation status and the optimization enablement of the R8 compiler, respectively. It is worth noting that "optimization" here refers to the 13 optimization techniques included in the Android R8 compiler that can be adjusted by modifying the Gradle configuration file, as mentioned in existing research [40], including *Inlining* and *CallSiteOptimization*. For optimizations such as *DeadCodeRemoval*, which are inherently enabled in R8 and cannot be disabled via configuration, we do not impose any restrictions.

In terms of TPLs selection, we obtained all the TPLs included in the apps from F-Droid, which contained 15,663 different version of libraries, and selected the top 100 most widely used TPLs, with a total of 1,308 different versions. Meanwhile, we built a blacklist based on the Android API reference documents [15] officially released by Google (as shown in Table 4) and excluded these libraries when generating TPL labels. This is because such libraries often retain their original names or are encapsulated in *APK* files as part of the signature mechanism.

Regarding the acquisition of TPL labels, unlike existing work that parses Gradle configuration files, we modified the compilation command so that the TPLs actually introduced during the app compilation process would be used as its labels. This method provides more comprehensive results, as it not only captures the TPLs directly depended upon by the app but also captures the indirect dependencies introduced by the libraries included in the TPLs.

4.1.2 Tool Selection. Our evaluation focuses only on relatively new, publicly available Android TPL detection tools that have not yet been systematically assessed, as well as older tools that have demonstrated strong performance in previous evaluations [43]. The characteristics of these tools are summarized in the Table 3. According to the statistics, these tools were released between 2016 and 2024, utilizing different code features to detect TPL in Android apps at varying levels of granularity. We selected four tools, including *LibScout* [7], *LibPecker* [46], *LibScan* [37], and *LibHunter* [39], as our comparison subjects. Specifically, *LibScan* (2023) and *LibHunter* (2024) represented newer tools, while *LibScout* and *LibPecker* were earlier tools with notable influence. Other tools were not included due to outdated environments (e.g., *LibRadar* [27]), inferior performance in prior work (e.g., *LibID* [44], *Orlis* [35]), or lack of open-source availability (e.g., *ATVHunter* [42]).

4.1.3 Experiment Metrics. As objective evaluators, we strive to design evaluation metrics that ensure fairness, aiming to provide an impartial assessment of the effectiveness and runtime efficiency of the selected Android TPL detection tools.

In terms of effectiveness, we compared existing Android TPL detection tools at both the library-level and the version-level. In library-level evaluation, a *true positive* (TP_l) sample means that any version of a library in groundtruth is correctly identified as existing. Conversely, a *false positive* (FP_l) sample refers to the incorrect identification of any version of a library as existing when it is not in groundtruth. A *false negative* (FN_l) sample, on the other hand, occurs when all versions of a library in groundtruth are erroneously classified as non-existent. In version-level evaluation, a *true positive* (TP_v) sample means that a specific version of a library in groundtruth is correctly identified as existing. Conversely, a *false positive* (FP_v) sample occurs when a version of a library that is not present in groundtruth is mistakenly identified as existing, even if it pertains to a different version of the same library. A *false negative* (FN_v) sample, on the other hand, refers to the scenario where a specific version of a library that is present in groundtruth is incorrectly classified as non-existent. We employ the metrics *Precision*, *Recall*, and *F1-score* as same as described in Section 3.2.1, both at

Table 3: Comparison of Third-Party Library Detection Tools

Tool	Basic Information				Design & Implementation				
	Year	Avail.	Lang.	Eval.	Preproc.	Gran.	Method	Features	Obf.
LibScout*	2016	✓	Java	✓	Soot	Package	Match	Package Hierarchy Structure, Package Signature Hash	ProGuard
LibRadar,	2016	✓	Python	✓	Apktool	Package	Clustering	Package Feature Hash	Not Mentioned
LibPecker*	2018	✓	Java	✓	dexlib2, Soot	Class	Match	Package Hierarchy Structure, Class Dependency Signature	ProGuard
ORLIS	2018	✓	Java	✓	Soot, SDHash	Library, Class	Match	Library Digest, Class Digest	ProGuard, Allatori, DashO
LibID	2019	✓	Python	✓	dex2jar, AG	Class	Match	Class Signature, Class Dependency	ProGuard
ATVHunter	2021	✗	Java	✗	Apktool, Soot, AG	CDG, Method	Match	Class Dependency Graph, Opcode Sequence	ProGuard, Allatori, DashO
LibScan*	2023	✓	Python	✗	dx, AG	Class	Match	Class Signature, Method Opcode	ProGuard, Allatori, DashO
LibHunter*	2024	✓	Python	✗	dx, AG	Class	Match	Class Signature, Method Opcode	Android R8

Abbreviations: Avail. = Source Available, Lang. = Programming Language, Eval. = Has Been Evaluated by Other Works, Preproc. = Preprocessing Tool, Gran. = Granularity, Obf. = Target Obfuscation Tools, * = Selected to be Compared, AG = AndroGuard, CDG = Class Dependency Graph.

Table 4: Category-wise Prefixes for Android Development

Categories	Prefix
Android Platform	android., java., javax., org.apache.http., org.json, org.w3c.dom., org.xml.sax, org.xmlpull., dalvik.
Jetpack	androidx., tools.build.jetifier
Kotlin	kotlin., org.w3c., kotlinx.
Ktor	io.ktor.
Kotlin Gradle Plugin	org.jetbrains.kotlin, org.jetbrains.kotlinx

the library-level and at the version-level Android TPL detection. Since the number of negative samples typically far exceeds that of positive samples in the TPL detection task, we do not employ the metric of *Accuracy* as it can be misleading.

In terms of runtime efficiency, we ensure identical computational resources and sufficient memory availability for all tools. We then compare their runtime efficiency based on the time overhead required to process tasks of the same scale.

4.2 Effectiveness

We evaluate the effectiveness of Android TPL detection tools at both library and version levels using our benchmark datasets (Section 4.1.1) and analyze the results to uncover factors influencing tool performance. This analysis addresses **RQ2** and **RQ3**.

4.2.1 Effectiveness Evaluation. We evaluated the four Android TPL detection tools, which are LibScout, LibPecker, LibScan, and LibHunter, at both the library-level and version-level. The experimental results are presented in Table 5. For all tools, we determine the existence of an Android TPL based on the relationship between the detection similarity score and default threshold set by each tool.

At the library-level, *LibScout* can generate additional results based on whether the root package name of an Android TPL matches the corresponding package name in the app. These results are recorded separately in the table under *LibScout-LibMatch*.

At the version-level detection, the *F1-score* of all tools decreases significantly, indicating a drop in overall effectiveness. Since *LibScout-LibMatch* can only perform basic judgments by comparing package name-related information and cannot output specific versions, it it incapable of conducting version-level TPL detection. Beneath the apparent rapid decline in *F1-score* at the version-level, the *Recall* values of each tool do not exhibit a significant drop, suggesting that the tools can still identify the correct library versions. The primary issue is that many incorrect versions are also identified, leading to a high number of false positives and reducing the precision of the detection results.

Finding 1. Existing Android TPL detection tools exhibit significant performance limitations even when analyzing apps without obfuscation or optimization, particularly those methodologies reliant on fine-grained code features.

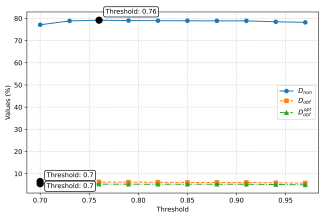
Detailed Analysis. The experimental results presented in Table 5 reveal significant performance limitations of fine-grained code feature-based tools (*LibHunter*, *LibScan*, and *LibPecker*) on the non-obfuscated and non-optimized dataset (D_{non}). At the library-level, these tools achieve notably low *F1-scores* of 24.58%, 38.20%, and 26.77%, respectively. Version-level performance degrades further, with all four tools exhibiting *F1-scores* below 26% (2.62%, 12.14%, 3.05%, and 25.04%).

The fundamental challenge stems from a critical mismatch in the compilation pipeline. TPL detection tools typically analyze TPLs in Java bytecode (*.class* files) but process Android apps in Android bytecode (*dex* format). To bridge this gap, tools like `d8` [3], `dx`, and `dexlib2` [16] are used to convert TPL *.class* files into *.dex* format. However, this conversion process rarely aligns with the compilers and configurations employed during the original app compilation. Even with identical source code, discrepancies in generated bytecode are inevitable due to differences in compiler implementations and optimization strategies.

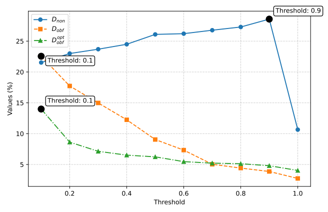
Table 5: Effectiveness Comparison of TPL Detection Tools

Detection Level	Dataset	LibHunter			LibScan			LibPecker			LibScout			LibScout-LibMatch		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Library-Level	D_{non}	14.47	81.71	24.58	38.10	38.31	38.20	21.76	34.77	26.77	74.46	80.01	77.14	74.65	86.67	80.21
	D_{obf}	14.08	80.14	23.95	19.33	5.33	8.35	4.56	5.62	5.04	22.20	3.78	6.46	70.66	34.05	45.95
	D_{obf}^{opt}	14.16	80.03	24.06	15.75	3.45	5.66	4.79	5.73	5.22	20.14	3.15	5.45	71.37	33.52	45.62
Version-Level	D_{non}	1.33	70.43	2.62	7.26	37.15	12.14	2.15	33.54	4.05	22.75	76.28	35.04	—	—	—
	D_{obf}	1.30	68.07	2.55	8.38	5.09	6.33	0.40	4.57	0.74	12.02	3.19	5.05	—	—	—
	D_{obf}^{opt}	1.31	67.74	2.57	6.44	3.34	4.40	0.34	3.37	0.62	11.71	2.67	4.35	—	—	—

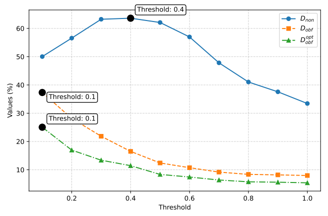
P: Precision (%), R: Recall (%), F1: F1-score (%). "—" indicates unavailable data.



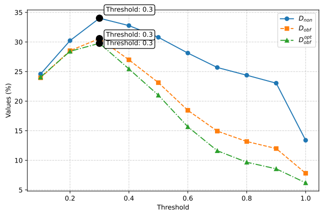
(a) LibScout



(b) LibPecker

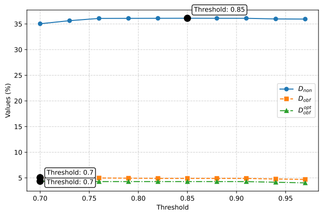


(c) LibScan

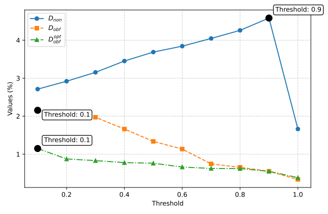


(d) LibHunter

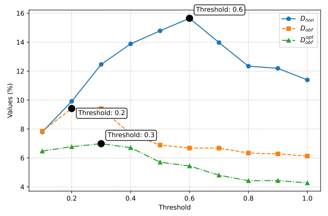
Figure 2: Optimal Threshold on Different Datasets in Library-Level Detection



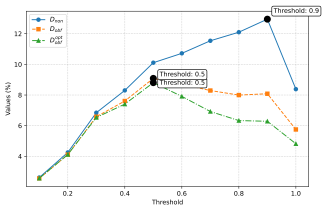
(a) LibScout



(b) LibPecker



(c) LibScan



(d) LibHunter

Figure 3: Optimal Threshold on Different Datasets in Version-Level Detection

This issue is exacerbated by the prevalence of Android R8 as the default compiler in modern app development. As noted in prior work [40], R8 applies optimization strategies like *DeadCodeRemoval* by default (different from *TreeShaking* in code shrinking), even when explicit obfuscation/optimization flags are disabled. These optimizations introduce subtle instruction-level modifications, such as the insertion of synthetic methods to enhance interoperability between internal and external Android components. Such transformations alter code characteristics critical for TPL detection, increasing both false negatives (missed library matches) and false positives (spurious matches). Crucially, this challenge persists across both non-obfuscated and obfuscated apps, as compiler-induced code variations affect all compilation scenarios.

Addressing this issue remains an open challenge. Existing research has yet to propose a reliable method to infer an app's compilation toolchain without access to its source code. Even when the correct compiler (e.g., R8) is identified and applied during TPL conversion, contextual differences in compilation parameters, dependency graphs, or build configurations can still produce divergent bytecode artifacts. This fundamental inconsistency undermines the reliability of bytecode-based TPL detection, irrespective of the tools or methodologies employed

Finding 2. Android apps sometimes depend on TPLs that do not contain actual code. When TPL detection tools preprocess these libraries (e.g. converting them into Android bytecode using tools like dx), such libraries may be excluded, leading to a decrease in both *Precision* and *Recall* of the detection results.

Detailed Analysis. Our analysis reveals that Android apps developed with Kotlin may incorporate metadata-only libraries from *Kotlin Multiplatform* (KMP), a framework enabling cross-platform code sharing (Android, JVM, JS, Native, etc.) through

Kotlin's `expect/actual` mechanism. KMP libraries primarily serve as abstraction layers, providing platform-agnostic interfaces while deferring concrete implementations to platform-specific modules. Consequently, these libraries lack executable code and instead contain metadata that guides the Kotlin compiler in resolving APIs and managing dependencies based on the target platform during compilation.

The detection of KMP libraries in Android TPL detection tasks holds indirect yet critical significance. While KMP libraries themselves do not contain executable code, they often establish dependencies on platform-specific libraries that provide concrete implementations. These transitive dependencies may introduce security risks (e.g., vulnerabilities in the implementation libraries) that propagate through the dependency chain. By analyzing invocation relationships between KMP libraries and their implementation dependencies, TPL detection tools can uncover hidden attack surfaces rooted in otherwise undetectable metadata-only artifacts.

This underscores a nuanced challenge: bytecode-centric TPL detection pipelines inherently overlook metadata-only libraries, creating blind spots in dependency graphs. Addressing this gap requires innovative strategies to trace metadata-driven dependency relationships and assess their security implications through associated implementation libraries.

4.2.2 Threshold Analysis. We noticed that the four selected Android TPL detection tools all adopt a similarity-based detection mechanism. They typically determine the relationship between the Android app and the third-party library by comparing the computed similarity with a given threshold. We attempted to adjust the detection thresholds for all the Android TPL detection tools when facing different datasets, in order to explore the relationship between the threshold and Android obfuscation patterns. The results are shown in Figure 2 and Figure 3.

Finding 3. The optimal threshold of existing TPL detection tools varies across apps with different obfuscation modes, making it challenging to achieve consistent detection performance.

Detailed Analysis. In the library-level and version-level detection results of all four Androir TPL detection tools, only *LibHunter*'s optimal threshold for library-level detection remains unaffected by dataset differences (as shown in Figure 2d), staying at 0.3. In all other cases, the optimal detection thresholds for the tools on obfuscated datasets (D_{obf} and D_{obf}^{opt}) are generally lower than those on the non-obfuscated dataset (D_{non}).

To make it more clear, we select the newer tools (*LibScan* and *LibHunter*) as examples. *LibScan* achieves the optimal threshold of 0.4 on the non-obfuscated dataset (as shown in Figure 2c), whereas on obfuscated datasets, the optimal threshold is 0.1. On the other hand, *LibHunter* maintains a stable optimal threshold of 0.3 across different datasets of obfuscation patterns (as shown in Figure 2d). This phenomenon indicates that the code features extracted by *LibScan* are influenced by code obfuscation and code optimization, while *LibHunter*, after improvements, has eliminated this impact. However, *LibHunter*'s optimal performance on D_{non} and D_{obf} is weaker that of *LibScan*, which may be due to *LibHunter*'s excessive focus on ensuring that features remain unaffected by obfuscation and optimization, leading to the introduction of additional false negatives and false positives. Such phenomena are often difficult to avoid completely.

4.2.3 Distribution of Causes. To further illustrate the connection between the causes we identified for the poor performance of existing Android TPL detection tools and their observed failures, as well as to investigate the extent to which each cause contributes to the degradation in detection effectiveness, we designed a library-level ablation study on the D_{non} dataset for in-depth analysis.

As a first step, we provide a summary of the previously identified causes as follows, where **C1**, **C2**, and **C3** represent the underlying causes identified through in-depth analyzes corresponding to **Finding 1**, **Finding 2**, and **Finding 3**, respectively. Detailed descriptions of these causes can be found in Sections 4.2.1 and 4.2.2.

- **C1**: Code transformation.
    - **C1.1**: Compiler/toolchain mismatch.
    - **C1.2**: Default dead code removal.
- **C2**: Libraries without code (e.g., resource-only or metadata-only).
- **C3**: Suboptimal detection thresholds.

Subsequently, to further assess the impact of the aforementioned causes on Android TPL detection, we treat the detection results with all identified causes eliminated as the upper bound of our experiment. We then incrementally reintroduce each influencing factor to the detection process. By observing the corresponding changes in detection performance, we evaluate the extent to which each factor contributes to the overall degradation.

Specifically, as shown in Table 6, the result labeled as Experiment 1 corresponds to the 'upper bound' setting, where we perform TPL detection using a package name-based matching approach on the D_{non} dataset after removing code-free TPLs. The detection threshold is fine-tuned to its optimal value. In Experiment 2, we reintroduce code-free TPLs into the dataset, resulting in the corresponding performance drop. For Experiment 3, we adopt a class hierarchy–based detection approach, with its detection threshold also tuned to the optimal value. Due to the default *DeadCodeRemoval* option in compilers, which typically removes code at the class level, structural mismatches in class hierarchies often lead to degraded performance. Finally, in Experiments 4 and 5, we evaluate *LibScan*, the best-performing code feature-based TPL detection tool on the D_{non} dataset. We report results using both its optimal threshold (Experiment 4) and its default threshold setting (Experiment 5).

Table 6: The Distribution of the Different Causes

No.	Introduced Cause(s)	Resulting F1-score
1	None (Upper Bound)	86.3%
2	+ C2 (Code-free TPLs)	82.3%
3	+ C1.2 (Default Dead Code Removal)	79.2%
4	+ C1.1 (Compiler/Toolchain Mismatch)	63.6%
5	+ C3 (Default Threshold Setting)	38.2%

The results in Table 6 indicate that **C1.1** and **C3** each lead to an approximately 20% reduction in the overall F1 score of TPL detection, suggesting that they a re the primary factors contributing to poor detection performance. **C1.2** and **C2** result in a more modest decline of around 4% in the F1 score, indicating that they also negatively affect detection accuracy, albeit to a lesser extent.

It is important to note that we are unable ti isolate and independently introduce each cause to precisely quantify its individual effect. Therefore, we infer the relative impact of each factor based on the observed stepwise degradation in detection performance.

Based on the evaluation of detection effectiveness conducted in this section, we draw two key conclusions. Regarding **RQ2**, existing Android TPL detection tools exhibit limited effectiveness. Notably, fine-grained code feature-based approaches may even perform worse than coarse-grained package name-based matching methods such as *LibScout-LibMatch*. As for **RQ3**, we identify three categories of factors that negatively impact detection effectiveness. Among them, **compiler/toolchain Mismatch** and **suboptimal detection thresholds** are the primary contributors to degraded performance.

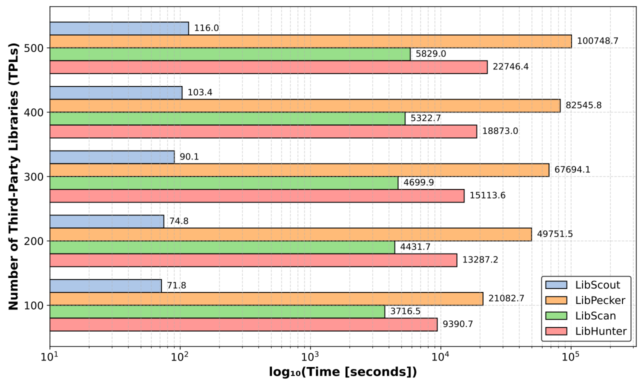


Figure 4: Efficiency Comparison of TPL Detection Tools

4.3 Efficiency Comparison

For the four Android TPL detection tools mentioned in Section 4.1.2, we also evaluated their runtime efficiency. The experimental results and our corresponding analysis provide answers to research questions **RQ2** and **RQ3**. We randomly selected 100 Android apps and tested them on datasets containing 100, 200, 300, 400, and 500 TPLs, respectively. For fairness, we configured all tools to run with 20 processes. Specifically, since *LibScout* was originally designed for single-process execution, we launched 20 concurrent instances. *LibPecker*, which inherently utilizes multiple CPU cores, was restricted to using only 20 CPU cores. For *LibScan* and *LibHunter*, both of which support multi-process execution via command-line configuration, we explicitly set the concurrency level to 20. This setup ensures a fair evaluation of each tool's runtime efficiency under the same computational resources, assuming sufficient memory availability.

To better simulate real-world scenarios, we excluded the preprocessing stage of TPLs for each tool. Specifically, for *LibScout*, we pre-profiled the TPLs, while for the other three detection tools, we pre-converted the Java bytecode of TPLs into Android bytecode using dx. The detection time results for each tool are shown in Figure 4. From the results, we can observe that the detection time of each tool increases linearly with the number of TPLs.

Finding 4. The time cost differences among different detection tools are significant, with the maximum disparity reaching nearly 1000 times.

Detailed Analysis. Based on the detection performance overhead, we roughly categorize the TPL detection tools into three levels.

(1) **Fastest**. *LibScout* achieves the highest speed by pre-building profiles for all TPLs before detection. During actual detection, it simply imports these templates, significantly reducing processing time.
(2) **Intermediate**. *LibScan* and *LibHunter* fall into this category. They process TPL features in a unified manner and store the results in memory during detection. This approach introduces additional runtime overhead, and if memory is insufficient, it may lead to crashes or other failures.
(3) **Slowest**. *LibPecker* exhibits the lowest efficiency as it re-extracts code features for each app-library pair during detection. This repetitive processing results in substantial extra overhead.

Regarding **RQ2** and **RQ3**, most existing Android TPL detection tools lack design considerations for runtime detection efficiency and scalability. There is still room for further optimization to achieve higher detection efficiency.

5 Discussion

5.1 Threat to Validity

In this section, we discuss two potential threats to the validity of our work and describe how we mitigate them. Both threats are relevant to ObfDetector. The first concerns ObfDetector's detection of code optimization. In our design, we use *Inlining* as a criterion for determine whether code has been optimized and evaluate its effectiveness on an open-source dataset. However, obfuscators in closed-source apps may differ and may not adopt inlining, potentially leading to false negatives and underestimating detection rates. Nevertheless, this does not affect our conclusion that code optimization is widely used in real-world scenarios. The second threat relates to evaluating ObfDetector on closed-source dataset D_{closed}. Manual labeling is inherently error-prone, and perfect accuracy cannot be guaranteed. However, validation on the open-source dataset D_{open} helps ensure the robustness of our findings.

5.2 Future Research Directions

Based on the four findings presented in Section 4, we propose several recommendations for designing future Android third-party library (TPL) detection tools. These recommendations aim to enhance detection accuracy and runtime efficiency in the presence of obfuscation and optimization techniques.

(1) According to Finding 1, we suggest that in the feature extraction and match phase of Android TPL detection tools, more flexible matching strategies can be adopted. For instance, contrastive learning and other deep learning methods can be used to establish correspondences between functions or classes, improving tolerance to differences caused by different compilers. Alternatively, instead of converting the TPL's Java bytecode, a direct mapping between the library's Java bytecode and the app's Android bytecode can be established.
(2) According to Finding 2, we suggest that future work can consider first constructing a comprehensive TPL metadata database, identifying dependencies between libraries, and leveraging these dependencies to enhance detection results, thereby improving overall detection performance.
(3) According to Finding 3, since it is difficult to simultaneously extract code features for Android TPL detection that are unaffected by obfuscation while achieving strong detection performance, future detection tools can first classify the apps to be detected (e.g., by determining the obfuscation methods they use). Then, based on the different categories of apps, different detection schemes can be applied, leading to a higher overall detection performance.
(4) According to Finding 4, we suggest that future Android TPL detection tools adopt a profiling mechanism similar to *LibScout*. By pre-building profiles for target TPLs, tools can significantly reduce the computational overhead during actual detection, improving efficiency and scalability.

Collectively, these recommendations advocate for obfuscation-aware detection strategies, metadata-driven dependency analysis, adaptive classification frameworks, and profile-based optimizations to advance the reliability and efficiency of Android TPL detection in real-world security scenarios.

6 Related Work

6.1 Android Third-Party Library Detection

Early research on Android third-party library (TPL) detection primarily relied on explicit information such as package names, class names, and class signatures. These methods are based on static analysis and mainly involve matching known library package names or identifying TPLs through signatures. A representative study is LibScout [8], which can identify Android TPLs using package names and package hierarchy information. However, with the widespread adoption of obfuscation techniques (e.g., ProGuard and R8) in Android app development, detection methods relying solely on explicit information such as package names and class names have shown their limitations.

To overcome the limitations of name and signature-based detection, researchers have proposed detection methods based on code snippets or code features [8, 20, 21, 24, 26, 29, 31–33, 35, 38, 40–42, 44, 47]. These methods not only match class names but also identify libraries through code similarity, enabling the detection of libraries even when their names have been obfuscated. Representative works include LibRadar, LibDetect [21], Orlis [35], LibPecker [47], LibID [44], PanGuard [33], ATVHunter [42], LibScan [38], and LibHunter [40]. Among these, LibScan and LibHunter are relatively recent works. LibScan extracts features from Android apps and TPL code at the class level. It performs feature matching from three perspectives: class signature correspondence, method opcode similarity, and call chain opcode similarity. Even in apps obfuscated using traditional methods such as ProGuard [4], DashO [5], and Allatori [6], LibScan can still achieve high detection accuracy and demonstrates excellent performance. Building upon the rules of LibScan, LibHunter introduced improvements by designing specific rules for the recently popular Android D8 and R8 compilers [3]. This enhancement allows LibHunter to better withstand the impact of R8 obfuscation and optimization on detection, achieving improved detection results under different compilation strategies.

6.2 Android Obfuscation Detection

Existing Android TPL detection tools are highly susceptible to the impact of code obfuscation, which affects the accuracy of TPL detection. Therefore, detecting the obfuscation patterns in Android apps and investigating the prevalence of obfuscation across the entire Android app marketplace is crucial for improving the accuracy of TPL detection in Android apps [45].

Early obfuscation detection tools primarily focused on detecting obfuscation techniques based on keyword features, such as identifier renaming, Java reflection, and string encryption. In 2015, IREA [25] introduced the first method for detecting Android obfuscation patterns based on keyword features. In 2018, Shuaike Dong et al. [17] proposed a prototype of an obfuscation detection tool based on Androguard, which was used to detect apps from Google Play, third-party app markets, and malware datasets. In the same year, OBFUSCAN [36] conducted the first large-scale analysis of Android apps, examining 1.7 million apps. The results showed that nearly 25% of these apps had undergone obfuscation. In 2019, AndrODet [28] introduced the first detection method for control flow obfuscation in Android apps. It manually extracted features from Android bytecode and utilized machine learning classifiers for training and classification to identify potential obfuscation in the code. In 2022, AndroDet* [9] made improvements on the original AndrODet, incorporating techniques from the field of Natural Language Processing (NLP) and employing deep learning models for obfuscation detection, which significantly enhanced the detection effectiveness. In addition to obfuscation detection tools in the Android domain, the work by Pei Wang et al. [34] published in 2018 is capable of detecting Symbol Renaming, Exotic String Encoding, Decompilation Disruption, and Control Flow Flattening in iOS apps. Although there are certain differences compared to Android-based methods, the detection approaches presented in their work offer valuable insights that can inspire advancements in Android obfuscation detection.

7 Conclusion

Android third-party library (TPL) detection has been a long-standing research problem in the field of software security. However, existing studies have yet to achieve satisfactory results. In this paper, we design three research steps to investigate the root causes of these limitations. First, we design, implement, and evaluate the Android obfuscation detection tool ObfDetector to analyze obfuscation usage in real-world apps, confirming the necessity of considering code obfuscation in TPL detection tool design. Second, we construct a benchmark dataset that simulates real app environments to evaluate the detection effectiveness and runtime efficiency of existing TPL detection tools, verifying their suboptimal performance. Finally, through an in-depth analysis of the design and implementation of these tools, we summarize four key factors affecting their performance and provide design recommendations for future research.

References

[1] 2023. Android and Google Play statistics. https://www.appbrain.com/stats.

[2] 2023. Shrink, obfuscate, and optimize your app. https://developer.android.com/build/shrink-code.

[3] 2025. D8 dexer and R8 shrinker. https://r8.googlesource.com/r8/.

[4] 2025. GuardSquare. ProGuard. https://www.guardsquare.com/proguard.

[5] 2025. PreEmptive. DashO. https://www.preemptive.com/pro-ducts/dasho.

[6] 2025. Smardec Inc. Allatori. http://www.allatori.com.

[7] Michael Backes. 2025. *LibScout*. https://github.com/reddr/LibScout

[8] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 356–367. https://doi.org/10.1145/2976749.2978333

[9] Mauro Conti, Vinod P., and Alessio Vitella. 2022. Obfuscation detection in Android applications using deep learning. *Journal of Information Security and Applications* 70 (2022), 103311. https://doi.org/10.1016/j.jisa.2022.103311

[10] 2010-2025 F-Droid Contributors. 2025. F-Droid. https://f-droid.org/. Accessed: 2025-03-10.

[11] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2187–2200.

[12] APKPure Developer. 2025. APKPure. https://apkpure.com/. Accessed: 2025-03-10.

[13] Google Developer. 2025. *Android Gradle plugin 8.9 release notes*. https://developer.android.google.cn/build/releases/gradle-plugin

[14] Maven Repository Developer. 2025. *Maven Repository: Search/Browse/Explore*. https://mvnrepository.com/

[15] Android Developers. 2025. Package Index. https://developer.android.com/reference/packages. Accessed: 2025-03-10.

[16] dexlib2 Developer. 2025. *dexlib2 is a library for reading/modifying/writing Android dex files*. https://mvnrepository.com/artifact/org.smali/dexlib2

[17] Shuaike Dong, Menghao Li, Wenrui Diao, Liu Xiangyu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. 2018. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild.

[18] William Enck, Damien Octeau, Patrick D McDaniel, and Swarat Chaudhuri. 2011. A study of android application security.. In *USENIX security symposium*, Vol. 2. 1–38.

[19] Runhan Feng, Zhuohao Zhang, Yetong Zhou, Ziyang Yan, and Yuanyuan Zhang. 2024. Accurate and Efficient Code Matching Across Android Application Versions Against Obfuscation. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 204–215. https://doi.org/10.1109/SANER60148.2024.00028

[20] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. 2017. CodeMatch: obfuscation won't conceal your repackaged app *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 638–648. https://doi.org/10.1145/3106237.3106305

[21] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. 2017. CodeMatch: obfuscation won't conceal your repackaged app. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 638–648. https://doi.org/10.1145/3106237.3106305

[22] Google. 2025. Google Play. https://play.google.com/store/games?device=windows. Accessed: 2025-03-10.

[23] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. 101–112.

[24] Qiang He, Bo Li, Feifei Chen, John Grundy, Xin Xia, and Yun Yang. 2022. Diversified Third-Party Library Prediction for Mobile App Development. *IEEE Transactions on Software Engineering* 48, 1 (2022), 150–165. https://doi.org/10.1109/TSE.2020.2982154

[25] Marian Kühnel, Manfred Smieschek, and Ulrike Meyer. 2015. Fast Identification of Obfuscation and Mobile Advertising in Mobile Malware. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. 214–221. https://doi.org/10.1109/Trustcom.2015.377

[26] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: Scalable and Precise Third-Party Library Detection in Android Markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 335–346. https://doi.org/10.1109/ICSE.2017.38

[27] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and Accurate Detection of Third-Party Libraries in Android Apps. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 653–656.

[28] O. Mirzaei, J.M. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano. 2019. AndrODet: An adaptive Android obfuscation detector. *Future Generation Computer Systems* 90 (2019), 240–261. https://doi.org/10.1016/j.future.2018.07.066

[29] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. 2014. AdDetect: Automated detection of Android ad libraries using semantic analysis. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. 1–6. https://doi.org/10.1109/ISSNIP.2014.6827639

[30] Shashi Shekhar, Michael Dietz, and Dan S Wallach. 2012. {AdSplit}: Separating smartphone advertising from applications. In *21st USENIX Security Symposium (USENIX Security 12)*. 553–567.

[31] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, Annamalai Narayanan, and Lipo Wang. 2016. LibSift: Automated Detection of Third-Party Libraries in Android Applications. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 41–48. https://doi.org/10.1109/APSEC.2016.017

[32] Wei Tang, Yanlin Wang, Hongyu Zhang, Shi Han, Ping Luo, and Dongmei Zhang. 2022. LibDB: an effective and efficient framework for detecting third-party libraries in binaries. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 423–434. https://doi.org/10.1145/3524842.3528442

[33] Zhushou Tang, Minhui Xue, Guozhu Meng, Chengguo Ying, Yugeng Liu, Jianan He, Haojin Zhu, and Yang Liu. 2019. Securing android applications via edge assistant third-party library detection. *Computers & Security* 80 (2019), 257–272. https://doi.org/10.1016/j.cose.2018.07.024

[34] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. 2018. Software Protection on the Go: A Large-Scale Empirical Study on Mobile App Obfuscation. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 26–36. https://doi.org/10.1145/3180155.3180169

[35] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. 2018. ORLIS: obfuscation-resilient library detection for Android *(MOBILESoft '18)*. Association for Computing Machinery, New York, NY, USA, 13–23. https://doi.org/10.1145/3197231.3197248

[36] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. 2018. A Large Scale Investigation of Obfuscation Use in Google Play. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 222–235. https://doi.org/10.1145/3274694.3274726

[37] Yafei Wu. 2025. *LibScan: towards more precise third-party library identification for android applications*. https://github.com/wyf295/LibScan

[38] Yafei Wu, Cong Sun, Dongrui Zeng, Gang Tan, Siqi Ma, and Peicheng Wang. 2023. LibScan: towards more precise third-party library identification for android applications *(SEC '23)*. USENIX Association, USA, Article 190, 18 pages.

[39] Zifan Xie. 2025. *Detect TPL versions for Android apps against optimization, obfuscation and shrinking*. https://github.com/CGCL-codes/LibHunter

[40] Zifan Xie, Ming Wen, Tinghan Li, Yiding Zhu, Qinsheng Hou, and Hai Jin. 2024. How Does Code Optimization Impact Third-party Library Detection for Android Applications?. In *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1919–1931.

[41] Liu Xinyu, Jin Ze, Liu Jiaxi, Liu Wei, Wang Xiaoxi, and Liu Qixu. 2023. ANDetect: A Third-party Ad Network Libraries Detection Framework for Android Applications. In *Proceedings of the 39th Annual Computer Security Applications Conference*. 98–112.

[42] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. 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)*. 1695–1707. https://doi.org/10.1109/ICSE43902.2021.00150

[43] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. 2020. Automated Third-Party Library Detection for Android Applications: Are We There Yet?. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 919–930.

[44] Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. 2019. LibID: reliable identification of obfuscated third-party Android libraries *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 55–65. https://doi.org/10.1145/3293882.3330563

[45] Xiaolu Zhang, Frank Breitinger, Engelbert Luechinger, and Stephen O'Shaughnessy. 2021. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Science International: Digital Investigation* 39 (2021), 301285. https://doi.org/10.1016/j.fsidi.2021.301285

[46] Yuan Zhang. 2025. *Detecting Third-Party Libraries in Android Applications with High Precision and Recall.* https://github.com/yuanxzhang/LibPecker

[47] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. 2018. 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)*. 141–152. https://doi.org/10.1109/SANER.2018.8330204