How Does Code Optimization Impact Third-party Library Detection for Android Applications?

Zifan Xie*†
Huazhong University of Science and Technology
Wuhan, China
xzff@hust.edu.cn

Ming Wen*†‡
Huazhong University of Science and Technology
Wuhan, China
mwenaa@hust.edu.cn

Tinghan Li
Huazhong University of Science and Technology
Wuhan, China
lith@hust.edu.cn

Yiding Zhu*†
Huazhong University of Science and Technology
Wuhan, China
ydzhu@hust.edu.cn

Qinsheng Hou
Shandong University; QI-ANXIN Technology Research Institute
Qingdao, China
houweidejia959@gmail.com

Hai Jin*§
Huazhong University of Science and Technology
Wuhan, China
hjin@hust.edu.cn

Abstract

Android applications (apps) widely use third-party libraries (TPLs) to reuse functionalities and simplify the development process. Unfortunately, these TPLs often suffer from vulnerabilities that attackers can exploit, leading to catastrophic consequences for app users. To mitigate this threat, researchers have developed tools to detect TPL versions in the app. If an app is found using a TPL vulnerable version, these tools will issue warnings. Although these tools claim to resist the effects of code obfuscation, our preliminary study indicates that code optimization is common during the app release process. A lack of consideration for the impact of code optimizations significantly reduces the effectiveness of existing tools. To fill this gap, this work systematically investigates how and to what extent different optimization strategies affect existing tools. Our findings have led to a new tool named LibHunter, designed to against major code optimization strategies (e.g., Inlining and CallSite Optimization) while also resisting code obfuscation and shrinking. Extensive evaluations on a dataset of apps with optimization, obfuscation, and shrinking enabled show LibHunter significantly outperforms existing tools. It achieves F1 value that surpass the best tools by 29.3% and 36.1% at the library and version levels, respectively. We also applied LibHunter to detect vulnerable TPLs in the top Google Play apps, which shows the scalability of our approach, as well as the potential of our approach to facilitate malware detection.

Keywords

Code Optimization, Third-party Library, Android

ACM Reference Format:
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 *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695554

1 Introduction

Android apps now hold a predominant share of the smartphone app market, with over 2.44 million apps available in the Google Play Store [43]. A significant contributor to Android's success is its open-source ecosystem, which fosters extensive developer engagement and innovation. The ecosystem has led to the proliferation of diverse third-party libraries (TPLs), which are integral to over 60% of the code in many Android apps. These libraries enhance app functionality with features like advertising, social networking, and payment services, streamlining development processes significantly [23, 29, 41, 62]. Numerous apps incorporate over twenty different TPLs, according to recent studies [8, 20–22, 32, 52, 60].

However, such TPLs are not always free of risks, as they frequently contain vulnerabilities that can pose significant threats to users. Recent studies indicate that 74.95% of these libraries are vulnerable and extensively used in various applications and other TPLs, making them attractive targets for malicious exploits [4, 11, 14, 17, 48, 49]. This widespread integration not only increases the attack surface but also potentially exposes users to severe security threats. For instance, the notable vulnerabilities in Apache Log4j2 [31] impacted over 35,000 Java packages, representing more than 8% of the Maven Central Repository [38]. Despite efforts to rectify these issues, many remain unresolved due to complex dependency chains, delaying effective remediation and leaving thousands of artifacts vulnerable. Consequently, one demanding challenge for developers, particularly for platforms like Google Play, is to timely

*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology (HUST), Wuhan, 430074, China
†Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, HUST, Wuhan, 430074, China
‡Corresponding author.
§Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695554

detect and manage such vulnerable libraries to safeguard end-users from potential threats. Therefore, identifying vulnerable TPLs has emerged as a critical and highly sought-after task, and detecting TPL versions has become a widely recognized standard known as Software Composition Analysis (SCA) [2, 24, 26].

Effective detection of TPLs is crucial for numerous security protocols. However, the utilization of code protection techniques such as obfuscation (e.g., renaming classes and members), code shrinking (e.g., removing unused code and resources), and optimization (e.g., applying aggressive strategies to reduce app size and enhance performance) significantly complicates this process. Common tools like ProGuard [34], DashO [7], Allatori [1], and Android's R8 compiler [35] are capable of implementing these code transformations to protect an app's code. A recent analysis indicates that 24.9% of 1.7 million free Android apps from Google Play undergo obfuscation for distribution. This percentage increases to 50.0% among apps that have achieved more than 10 million downloads [51].

Recent studies mainly utilize similarity-based methods for TPL detection. These tools claim to withstand the effects of code obfuscation and shrinking [5, 6, 10, 13, 16, 45]. For example, LibPecker [63] constructs precise class signatures based on class dependencies and uses an adaptive matching approach to compare libraries and application classes based on a fuzzy weighted similarity, especially when handling package flattening or class repackaging. LibScan [53], the state-of-the-art tool, focuses on method-opcode similarity by analyzing the set-based inclusion relationship of per-method opcodes and also evaluates call-chain-opcode similarity. Although these tools are effective against code obfuscation and shrinking, they overlook the effects of code optimization while optimization is widely considered in modern compiler designs [25, 56, 57]. Notably, R8, the component of Android's standard build tools, is extensively used by developers for comprehensive whole-program optimization, which substantially changes code structures. This situation prompts our investigation into how such optimizations affect the performance of existing state-of-the-art TPL detection tools.

We first assess ***the prevalence of R8*** usage by examining whether apps choose to protect their code with R8. Specifically, we analyzed 2,347 open-source apps from F-Droid [15] by inspecting their Gradle build files and checking whether these apps employ R8 for distribution. Our findings reveal that 41.1% of apps use R8, indicating that deploying apps compiled with R8 is a common practice. We then assess ***how code optimization affects the effectiveness of existing tools***. We construct $dataset_1$ with five app variants each using different strategies using R8: "D8 alone", "Obfuscation", "Shrinking", "Optimization + Shrink", and "Obfuscation + Optimization + Shrink" (R8's default strategy). Evaluations reveal that tools like LibScan struggle with R8's default strategy, with F1 scores significantly reduced to 41.1% for library-level and 14.7% for version-level detection, highlighting the challenges posed by optimizations. However, using $dataset_1$ alone does not clarify which optimization strategy most affects the performance. We then construct $dataset_2$, consisting of 13 app variants, each enabling a unique R8 optimization strategy. This dataset helps to gain a deeper understanding of ***the effects of different optimization strategies***. Our findings suggest that the Inlining and CallSiteOptimization (CSO) greatly diminish the effectiveness of existing tools, thus calling actions to mitigate such specific impacts.

Aiming to design better TPL detection tools that can resist code optimization, we further examine the working mechanisms of how such optimizations are applied in practice. We observe that ***certain optimization processes can be approximated and deduced via fine-grained static analysis***. For instance, for CSO, we can analyze argument usages at method callsites to track unused or consistently assigned parameters, thus tracking the prototype for an optimized method. Based on such a key insight, we propose an effective TPL detection tool named LibHunter via integrating fine-grained static analysis. LibHunter's goal is to address the challenging scenarios of detecting TPL versions when the app has been optimized, obfuscated, and shrunk while maintaining low costs. Specifically, LibHunter mainly contains three steps. First, LibHunter extracts class features for both the target app and TPLs. It then enhances the TPL class features based on the approximation of CSO via static analysis. Such enhanced features enable LibHunter to establish over-approximated class correspondence relations between app and TPL classes. Second, LibHunter calculates method similarities using the method's opcodes and strings. If this similarity exceeds a predefined threshold, the methods are considered fully matched. However, many methods cannot be fully matched due to Inlining. For such cases, we consider them partially matched if a significant portion of a TPL method's features are found within an app method. The insight is that if an app method is synthesized from TPL methods, it will contain features of those TPL methods. Third, LibHunter approximates the Inlining process along with computing method similarities to finalize the method matching correspondence for partially matched methods. These matched methods are called cross-inlining matching methods. Finally, LibHunter computes a confidence score to determine the final class correspondence and further calculates the similarity between the TPL and the app. If the similarity exceeds a threshold, it confirms the existence of the TPL.

Our evaluations show that LibHunter excels in challenging scenarios (i.e., optimized, obfuscated, and shrunk apps), it can achieve the F1 score of 71.4% and 51.1% at the library and version levels, respectively. On average, the F1 value of LibHunter outperforms the best baselines by 29.3% and 36.1% at the library and version levels. TPL detection tools can be used to detect vulnerabilities by pinpointing the TPL versions [32, 59]. For each detected TPL, if its version lies in the affected version range of a known CVE, the tool will issue a warning (i.e., an App-CVE pair). By empolying LibHunter, we can reveal the real-world risks of vulnerable TPLs by applying it to actual apps. Finally, LibHunter reported 3,761 warnings for 10,000 top Google Play apps. As a comparison, we found that baseline only report 2,039 warnings for these apps. This is because the baseline struggles to detect apps when they are optimized. Such results show the scalability of our approach, as well as the potential of our approach to facilitate malware detection.

To summarize, we make the following major contributions:

- **Originality.** We are the first to target detecting TPL versions in optimized Android apps, and we observe that deploying apps with code optimization is a prevalent practice.
- **Empirical Study.** We conduct a systematic study to investigate how code optimization affects the effectiveness of existing tools. Our findings indicate that optimization significantly affects the

effectiveness of existing state-of-the-art tools, with Inlining and CallSiteOptimization (CSO) casting the most substantial impact.

- **Evaluation.** We implement our approach as a prototype, LibHunter. Extensive evaluations demonstrate its effectiveness in detection of TPL versions, which also outperforms the SOTA baselines, especially in optimized applications.
- **Artifact.** We make our approach, the dataset of TPLs and related apps available at:

https://github.com/CGCL-codes/LibHunter

2 Background and Motivation

2.1 The D8 and R8 compiler

App developers frequently employ code obfuscation techniques to protect against reverse engineering. Without these protective measures, malicious individuals can decompile and scrutinize apps, exposing potential security weaknesses. Of 1.7 million free Android apps available on Google Play, approximately 24.9% undergo obfuscation prior to their launch [51]. This percentage increases to 50.0% for popular apps that have surpassed 10 million downloads [51].

R8, the default tool for the Android build process, now performs all of the code shrinking, obfuscation, and optimization tasks, which has replaced the obfuscator *Proguard* [58]. D8 was the built-in DEX compiler from Android Gradle Plugin 3.1.0 (March 2018), and R8 from Android Gradle Plugin 3.4.0 (April 2019) [58]. The D8 compiler converts a *.class* file to a *.dex* file. R8 extends D8 by incorporating additional capabilities for code shrinking, obfuscation, and optimization [44]. If developers do not apply third-party plugins to enable other obfuscators, they should include the statement "minifyEnabled true" in the "build.gradle" configuration file to enable R8 for these capabilities when building apps. Otherwise, D8 will be enabled by default, along with some simple optimizations, such as *Switch Rewriting* and *String Optimizations.*

Table 1 shows the default strategies supported by R8, which are automatically activated if users enable R8. Users can further specify "-dontX" to disable a particular strategy. For example, specifying "-dontoptimize" would turn off code optimization [18, 19].

- **Obfuscation**: It includes *Package Flatten* and *Identifier Renaming*. The former disrupts the original code hierarchy structure, repackaging classes from several packages into a single one. The latter involves renaming packages, classes, methods, and variables. For instance, identifiers may be changed to meaningless characters like x and y.
- **Shrinking**: It identifies and securely eliminates unused classes, fields, methods, and attributes from the app.
- **Optimization**: It encompasses 13 advanced strategies that modify code to enhance runtime performance and decrease the app's size. Due to page limitations, we have placed the descriptions of each strategy on our website [37]. Notably, these strategies are internal options, and users can enable or disable all. Moreover, R8 also supports other strategies (e.g., DeadCodeRemoval). However, these optimizations are not controlled by internal options but are embedded into the optimization process. Therefore, they are excluded from the 13 options.

Table 1: Supported Strategies for R8. "Obf" denotes Obfuscation, "Opt" denotes Optimization, "Srk" denotes Shrinking.

Type	Strategy	Type	Strategy
Obf	Package Flatten	Opt	InitializedClassesAnalysis
	Identifier Renaming		CallSiteOptimization
Srk	Tree Shaking		HorizontalClassMerger
Opt	Inlining		NameReflection
	ClassInlining		VerticalClassMerger
	Devirtualization		StringConcatenation
	EnumValueOptimization		EnumUnboxing
	Outlining		SideEffectAnalysis

2.2 Related Work

Existing TPL detecting tools. Currently, many detection tools have been proposed, aiming to counteract the effects of code obfuscation and shrinking. LibScout [3] leverages class hierarchy to map out TPL features. It creates library signatures using a fixed-depth Merkle tree, which simplifies the package layer and gathers non-obfuscated partial method signatures. LibPecker [63] builds precise class signatures from class dependencies. It employs an adaptive class matching that performs a fuzzy weighted similarity match between library and app classes if the similarity surpasses a certain threshold. It's particularly sensitive to package flattening or class repackaging as its package matching depends on package hierarchy. LibID [61] identifies features resilient to obfuscation and introduces a multi-phase process. Initially, it matches candidate TPL classes with app classes if each basic block of an app class matches an identical block in a library class. Then, through dependency matching, it selects the matched pairs from these candidates, employing a binary integer programming model to maximize matched pairs. Finally, it confirms the matched TPL version by evaluating the proportion of matched classes in the app package. ATVHunter [59] adopts a two-phase approach for detecting TPLs. The first phase assigns a unique serial number to each basic block in a method, transforming the CFG from adjacency lists into a method signature based on these numbers. The second phase utilizes fuzzy hashing across sliding-window opcodes to mitigate localized feature changes due to obfuscation. LibScan [53] is a state-of-the-art tool. It focuses on method-opcode similarity, examining the set-based inclusion relationship of per-method opcodes. It also assesses call-chain-opcode similarity based on the set-based inclusion relationship of call-chain opcodes. While the LibScan paper does include experiments on the impact of optimization on tool performance, it lacks further analysis of how optimizations affect performance and does not propose solutions to resist these effects.

Although these tools have been proven to resist code obfuscation and shrinking, none has taken optimization into consideration. In fact, as part of Android's default build tools, R8 is widely adopted by developers to enable extensive whole-program code optimization strategies, significantly altering the code structure. Therefore, we are motivated to investigate systematically how code optimization impacts the TPL detection tools.

3 Empirical Study

We aim to conduct a comprehensive study into how code optimization affects the performance of existing TPL detection tools by answering the following research questions:

- **RQ1: Pervasiveness of R8**: How pervasive it is for Android apps to utilize the R8/D8 compiler?
- **RQ2: Impact of Code Optimization**: How does code optimization affect the performance of existing TPL detection tools?
- **RQ3: Impact of Optimization Strategies**: Which code optimization strategies cast a greater impact on the performance of TPL detection tools?

3.1 Dataset Construction

APP Collection. To conduct experiments on TPL detection, we need to create a dataset that maps apps to TPLs. This dataset should meet two criteria: (1) it must provide the mapping information between apps and TPLs, and (2) it should include the complete version set for each TPL. Following existing research [53], we opt for F-Droid [15], an open-source Android app repository, to identify which TPL versions are used by each app. Specifically, we crawled 4,151 open-source apps from F-Droid. By analyzing the Gradle build files, we were able to determine all the TPLs, along with their versions that are utilized by each app. Following existing work [55], we focus on TPLs that involve vulnerabilities (i.e., associated with at least one CVE). If an app uses a vulnerable version of a TPL, it may suffer from potential security threats.

To investigate the impact of code optimization on the performance of TPL detection tools, we randomly select 200 apps from the above 4,151 ones. We discover that these 200 apps correspond to 31 unique vulnerable TPLs, resulting in a total of 379 app-TPL pairs. We manually compiled five variants of these apps by specifying the following options: (1) D8 alone, (2) Obf, (3) Srk, (4) Opt + Srk, and (5) Obf + Opt + Srk (R8's default strategy). In particular, "D8" refers to enabling D8 without R8; "Obf" means only R8's obfuscation was enabled; and "Opt + Srk" indicates only R8's optimization and Shrinking were enabled. We designed this variant since some optimization strategies (like HorizontalClassMerger and EnumUnboxing) only work when Shrinking is enabled. Consequently, our dataset comprises five sets of apps: one set of 200 D8-compiled apps and four sets of R8-compiled apps (200×4) using different compiling options, which is marked as $dataset_1$.

As aforementioned, R8 offers 13 advanced optimization strategies. However, these strategies are internal options, allowing users only to turn them all on or off. Therefore, using $dataset_1$ alone cannot provide insights into which specific optimization strategy impacts the TPL detection tool's performance the most. To gain a deeper understanding of the effects of different optimization strategies, we first obtained the R8 source code (commit=48c8b6) and manually modified the code to compile 13 R8 variants, each enabling only one optimization strategy. For instance, the variant "r8_Inlining" only enables the Inlining while disabling the others. We then randomly selected 50 apps from $dataset_1$ and compiled them using these 13 R8 variants. Since many optimization options are only activated when shrink is enabled, we also enable Shrinking during the compilation process. Consequently, this dataset comprises 650 (50×13) apps, which is marked as $dataset_2$.

Metrics. We evaluate performance of TPL detection tools at library and version levels. *library-level* detection refers to recognizing the TPLs used within an app without detailing their versions. *Version-level* detection involves not only identifying the correct TPLs but also determining their specific versions. We then employ the metrics $Precision = \frac{TP}{TP+FP}$, $Recall = \frac{TP}{TP+FN}$, $F1$-value $= \frac{2 \times Recall \times Precision}{Recall + Precision}$ following existing studies [53, 59].

For library-level detection, a *true positive* (TP) occurs when an app uses a TPL, and the tool successfully identifies it. The *false positive* (FP) happens when the tool mistakenly reports a TPL that the app does not actually include. The *false negative* (FN) arises when the tool fails to detect a TPL that is present in the app. Similarly, at the version level, a *false positive* means the tool identifies the specific version of a TPL while the app uses another version. In this context, a *true positive* refers to correctly reporting a TPL version that does exist in the app. For example, if a tool reports that an app is using *Retrofit 2.5.0*, but the actual TPL version is *2.6.0*, we consider this case a true positive at the library level but a false positive at the version level.

TPL Collection. We crawled all versions of 31 vulnerable TPLs from sources such as Maven Central [38] and Google's Maven repository [36], etc. TPLs are typically distributed in '.jar' or '.aar' formats. The '.jar' files contain bytecode files, whereas '.aar' files encompass bytecode along with additional Android-specific files. Finally, our dataset contains 31 unique TPLs and 3,447 corresponding versions.

Tool Selection. We select four state-of-the-art TPL detection tools, including LibScout [3], LibPecker [63], LibID [61], and LibScan [53] as the target tools to evaluate their performance.

3.2 RQ1: Pervasiveness of R8/D8

We focus on the pervasiveness of employing the R8/D8 compiler for apps. Typically, developers produce two app variants: a debug version for troubleshooting and a release version for distribution. We check whether the release version has R8/D8 enabled since it can cast real impact. From the initial set of 4,151 apps in $dataset_1$, we filtered out those with a Gradle Plugin version below 3.1.0, leaving us with 2,347 apps. We exclude certain apps because D8 hadn't been released before Gradle Plugin 3.1.0. Additionally, these apps haven't been updated for many years, making them outdated. We then analyze the Gradle configuration files of these apps by the following steps: (1) We first determine if any other obfuscators, such as Proguard [34] and Allatori [1], have been used, as these have been extensively studied among existing researches [53, 59]. (2) Subsequently, we identify the option "minifyEnabled true" following the oficial documentation [35]. If it is present, we consider R8 is being employed. Otherwise, we consider D8 is being used.

Result. In our analysis of 2,347 apps, we found that 1,380 (58.8%) use D8, 964 (41.1%) use R8, and only 3 apps (0.001%) resort to other obfuscators. This suggests that deploying apps compiled with R8 is a prevalent practice. We conjecture that commercial apps released on vendors like Google Play [33] may also use R8 to protect their code. This observation prompts us to conduct a systematic analysis of how code optimization affects the efficiency of TPL detection tools. However, existing studies mainly focus on other uncommon obfuscators (e.g., DashO [7]), which might cause bias.

Answer to RQ1: Our analysis of 2,347 Android apps revealed a significant trend: 58.8% utilize D8, while 41.1% opt for R8, highlighting the prevalent use of R8 in enhancing app security through code optimization, obfuscation, and shrinking in modern app development.

3.3 RQ2: Impact of Code Optimization

We aim to evaluate the performance of existing TPL detection tools on $dataset_1$ as collected in Section 3.1.

Overall Results. Table 2 shows the comparison between different TPL detection tools. We can find all tools perform best on D8-compiled apps, achieving their highest F1 value. We can find LibScan leads 91.6% for library-level detection and 85.8% for version-level detection. For obfuscated apps, LibScan's F1 value slightly dipped to 84.4% for version-level, demonstrating a certain level of resilience. However, others tools like LibID, plummeted to the F1 of 0.6%, indicating severe struggles with code obfuscation.

The decline became more pronounced with the R8 default strategy (Opt + Obf + Srk), where even the highest F1 value nosedived to a disappointing range around 41.1% for library-level and merely 14.7% for version-level detection (i.e., achieved by LibScan). Such results clearly indicate a significant drop in the effectiveness of TPL detection tools when facing complex app compilations.

FP analysis. For **D8-compiled** apps, false positives primarily stem from simple method-level optimizations introduced by D8, impacting tools that rely on method features. For example, if the app method contains opcodes from the TPL method, LibScan will simply consider these two methods as matching since some obfuscators (e.g., DashO) may insert redundant code into the app method. However, D8 introduces *String Optimizations*, which optimize bytecode by precomputing string operations. This may change the opcodes in the app methods, leading to mismatches for LibScan. For **R8-obfuscated apps**, R8 will break the original structure of the code hierarchy and repackage classes into new packages. However, tools like LibScout, LibPecker, and LibID assume most obfuscators will not alter internal package hierarchy structures for TPL identification, thus significantly compromising their effectiveness. In **R8-shrunk apps**, where unnecessary classes, methods, and fields are pruned, removing numerous methods and fields from a class reduces the similarity between two classes. If tools set higher thresholds for method/class/library match similarity, shrinking can impact their performance significantly. For **R8-optimized apps**, R8 enables 13 advanced whole-program optimizations, which change the code structure and features dramatically. For example, *Inlining* will replace a function call with the callee's entire code at the call site instead of performing a regular function call; *VerticalClassMerger* will attempt to merge parent and child classes into one. These optimizations can eliminate existing methods or synthesize new methods in a class, thus significantly affecting existing tools that ignores the optimization for library matching.

Apps that enable code optimization, obfuscation, and shrinking simultaneously present a challenging scenario. This combination significantly reduces the F1 value of existing tools, mainly because their designs did not account for such modifications.

Answer to RQ2: Experiments show that advanced tools like LibScan struggle under R8's default strategy (Obfuscation + Optimization + Shrink), with F1 value dropping to about 41.1% for library-level and 14.7% for version-level detection. This significant drop underscores the challenge existing tools face, as they were not designed to handle such complexities introduced by code optimization.

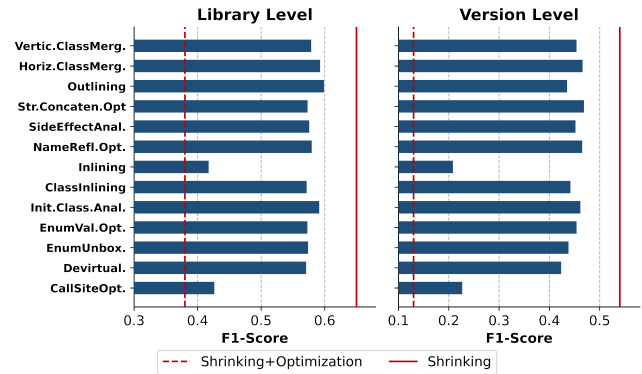


Figure 1: Detection results of LibScan on $dataset_2$. The red dashed line shows the worst-case scenario, and the red solid line shows the best-case scenario.

3.4 RQ3: Impact of Optimization Strategies

We delve further into how different optimization strategies impact tool performance. Since LibScan is currently the most advanced tool and has achieved the optimum results in Section 3.3, we further evaluate its performance on $dataset_2$. Note that each app in $dataset_2$ enabled only one optimization strategy, along with Code Shrinking.

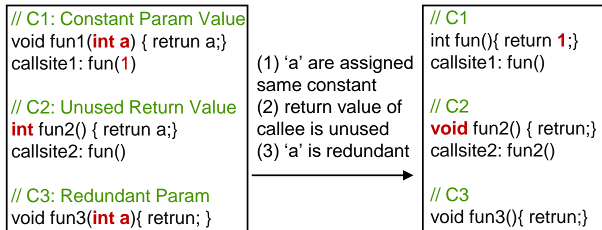
Results. Figure 1 shows the detection results of LibScan. For the library level, we use horizontal lines to demarcate the spectrum of results: one representing the optimal F1 value (0.64), obtained when shrinking alone was enabled, and the other indicating the worst score (0.38), achieved when both shrinking and all optimization options were activated. These benchmarks clarify the best and worst-case scenarios against which individual strategy performances were compared.

While most strategies show F1 value that are close to the best performance (within 57% to 60% range), indicating a limited impact on the effectiveness of LibScan, two strategies stand out due to their markedly lower F1 value. The strategies of *Inlining* and *CallSite Optimization (CSO)* recorded F1 value of 41.7% and 42.6%, respectively, positioning them nearest to the worst-case scenarios among all strategies. At the version level, a similar trend is observed, reinforcing the conclusion that *Inlining* and *CSO* are the most impactful strategies, notably diminishing the effectiveness of LibScan. This stark deviation suggests that these two optimization techniques significantly degrade the tool's performance when implemented independently. This analysis indicates that enabling either of these two options independently is likely to compromise the effectiveness of TPL detection, suggesting areas for further investigation.

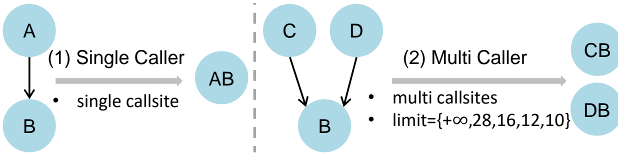
Analysis on Inlining and CallSiteOptimization (CSO). For apps with CSO enabled, the parameters of a method can be removed. As shown in Figure 2a, if a method's parameter is assigned the same constant across all callsites, R8 can eliminate this parameter and replace it with the constant. If the return value of the method is ignored at all callsites, its return type can be set to *void*. Additionally, if a parameter is not used within the method's body, it can also be removed. Although this strategy is simple, it significantly affects the effectiveness of LibScan. LibScan assumes that code obfuscation does not change the number of method parameters. Thus, in its

Table 2: Comparison on the Effectiveness of TPL Detection Tools to Different App Variants. PR=Precision, RC=Recall.

Detection Level	App Variants	LibScan			LibPecker			LibScout			LibID		
		PR	RC	F1	PR	RC	F1	PR	RC	F1	PR	RC	F1
library-level	D8-compiled	89.8%	93.4%	**91.6%**	77.3%	92.0%	84.0%	83.0%	74.5%	78.5%	77.4%	84.6%	80.8%
	Obf	89.8%	92.9%	**91.3%**	74.3%	21.1%	32.8%	66.7%	0.6%	1.1%	50.0%	0.3%	0.6%
	Srk	88.0%	64.3%	**74.3%**	82.1%	31.1%	45.1%	85.7%	10.2%	18.2%	56.3%	5.8%	10.6%
	Opt+Srk	83.7%	27.2%	41.1%	57.1%	2.2%	4.2%	79.3%	42.2%	**55.1%**	71.4%	1.7%	3.3%
	Opt+Obf+Srk	83.7%	27.2%	**41.1%**	57.1%	2.2%	4.3%	66.7%	0.6%	1.1%	50.0%	0.3%	0.6%
version-level	D8-compiled	80.5%	91.9%	**85.8%**	74.0%	89.5%	81.0%	84.0%	71.1%	77.0%	77.5%	81.0%	79.2%
	Obf	78.6%	91.1%	**84.4%**	80.8%	17.7%	29.1%	100.0%	0.6%	1.1%	100.0%	0.3%	0.6%
	Srk	53.9%	49.2%	**51.5%**	57.4%	20.6%	30.3%	66.7%	7.0%	12.7%	83.3%	4.9%	9.3%
	Opt+Srk	30.1%	10.1%	15.2%	87.5%	1.9%	3.7%	87.9%	39.1%	**54.1%**	80.0%	1.4%	2.7%
	Opt+Obf+Srk	29.1%	9.8%	**14.7%**	87.5%	1.9%	3.8%	100.0%	0.6%	1.1%	100.0%	0.3%	0.6%



(a) Illustration of CallSite Optimization in R8.



(b) Illustration of Inlining in R8.

Figure 2: Illustration of Inlining and CallSite Optimization. These two strategies cast the greatest impact on the TPL detection tool LibScan.

pre-match phase, if a method within an app class does not align with any method in a TPL class in terms of type and parameter count, these classes are deemed non-matchable. Clearly, the CSO strategy breaks this assumption because the number of method parameters can decrease, thus impacting LibScan's effectiveness.

For apps with Inlining enabled, the features of relevant methods will be altered. The main advantage of Inlining is to broaden the optimization scope by optimizing caller and callees together. Figure 2b shows the process of Inlining. Specifically, if a callee has only one callsite in the program (called Single Caller), R8 attempts to inline the callee into the caller. In another scenario, where a callee is invoked multiple times (called Multi Caller), R8 may not opt to inline since this could increase the program's size (as the body of the callee is replicated multiple times). Therefore, R8 heuristically defines an array that dictates the maximum size of callee allowed for inlining based on the number of callsites. The array is defined as $\{+\infty, 28, 16, 12, 10\}$. For example, if a callee is called at two callsites, it will be inlined only if its size is 28 or smaller; if a callee is called at four callsites, the size restriction is 10. However, if the callee has more than five callsites, Inlining will not occur. Therefore, if a TPL method inlines its callees and synthesizes into an app method m_a, then the code structure of m_a will significantly change. Since LibScan was not designed with the impact of Inlining, this strategy has increased its false negative rate for method matching. In its coarse-match stage, if an app method inlines other methods, it will not match the corresponding TPL method.

Insights. We found that Inlining and CSO have the most significant impact on LibScan's performance. However, addressing the challenges introduced by such optimizations is non-trivial. Fortunately, we observe that ***such optimization behaviors can be approximated and inferred via fine-grained static analysis***. For instance, for CSO, we can analyze argument usages at method callsites to track unused parameters or return value, thus inferring the prototype for an optimized method (**Insight#1**). For Inlining, we can follow the Inlining process and analyze which methods are likely to inline their callees within TPL binaries as well as how these methods are synthesized into an app method (**Insight#2**). Therefore, we can design a new TPL detection tool by integrating such insights to resist the impact of optimization.

Answer to RQ3: Our study shows that Inlining and CallSiteOptimization strategies exert the most substantial influence on LibScan's effectiveness. They alter the code structure, thus breaking the assumption of LibScan. It opens up the possibility of developing a new tool that specifically models the mechanisms of Inlining and CallSiteOptimization, aiming to resist their effects.

4 Approach

In this section, we present LibHunter that integrates the insights gained from our empirical studies to detect TPL versions in apps, which can effectively resist default strategies of R8 (optimization, obfuscation, and shrinking). The workflow of LibHunter is illustrated in Figure 3, which starts by taking an app with a TPL as input and outputs a verdict on the TPL's presence within the app.

However, determining the presence of TPL is challenging, especially when we strive to accommodate obfuscated, optimized, and shrunk code. To equip LibHunter with the capability of optimization resilience, the key insight is to extract optimization-resilient semantics to compare the app's semantics with the candidate TPL. As aforementioned, the optimization process can be approximated via

fine-grained static analysis. Therefore, our proposed optimization-resilient semantics mainly include two components: (1) The enhanced TPL features by integrating **Insight#1**. Since CSO alters a method's prototype, we can infer the form of a TPL method's prototype after CSO by referring the call graph and datafow of the TPL through fine-grained static analysis. This aids in the matching of method signatures. (2) The cross-inlining method matching by integrating **Insight#2**. Typical Inlining will synthesize multiple TPL methods into one app method. We follow this process to determine whether a method will inline its callees, thus predicting the synthesized method's features. This enhances method matching precision and boosts TPL detection effectiveness. By integrating these optimization-resilient components, LibHunter can accurately detect TPL versions. LibHunter is composed of three steps: *Signature-based Class Matching*, *Similarity-Based Method Matching* and *Cross-Inlining Method Matching*, which are described as follows.

4.1 Signature-based Class Matching

In this step, LibHunter builds signatures (i.e., a set of class features) to match the classes between the app and TPL. The signatures should remain consistent even if the classes have been optimized, shrunk, or obfuscated. Through our investigation, we identify certain code features that remain stable through code transformations. Thus, if the features of an app class and a TPL class are matched, they move on to the next steps; otherwise, the determination of class correspondence for this pair ceases.

We categorize such code features used for fingerprinting into two types: field-level and method-level features. Following existing research [3, 53, 59], we assign fuzzy signatures to each field and method to resist obfuscation. For fields, LibHunter records the declaration of the field and uses a placeholder `X` for user-defined classes. For example, a field declared as `MyClass f2` would have a fuzzy signature of `X`. LibHunter denotes all fields within a class as a set, S_{field}. For method fuzzy signatures, LibHunter leverages the method's return type and parameter types. For example, a method with the prototype "String fun(int p0)" would have its fuzzy signature extracted as "Ljava/lang/String;,int". Finally, LibHunter records all the methods in a set named S_{method}. We use S_{field} and S_{method} as the class signature.

Although the use of fuzzy signatures is a standard practice in existing works [3, 53, 59], CSO can alter the number of parameters in a method, which can change the method fuzzy signature. To address this, we need to infer what the method's prototype would look like after CSO. Specifically, we construct enhanced TPL features through static analysis. For a given TPL method m, LibHunter obtains all callers of m by querying the call graph and performs intraprocedural data flow analysis on these callers. LibHunter records the arguments passed to parameters of m. If a parameter is consistently assigned the same constant across all callsites, it is marked as unused. Additionally, LibHunter identifies whether the return value of m is discarded across all callsites, and if so, marks the return type as unused. Finally, LibHunter identifies parameters within the body of method m that are unused, marking them as unused. Subsequently, we will update the fuzzy signature for m. We use the "?" symbol from regular rule to indicate that a marked parameter can appear zero or one times. For example, given a method with the prototype "`int fun(int p1, long p2)`", if the return type and "`p2`" are marked as unused, its fuzzy signature is represented as a regular rule "`^(int|void),int(,long)?$`". This rule can be used to perform method signature matching. We call such regular rules as enhanced TPL features.

LibHunter then aims to locate candidate classes in the app that correspond to those classes in the TPL. Specially, given a class in TPL with the corresponding signature, LibHunter traverses all the classes in the app. We check whether S^{APP}_{fields} is contained in S^{TPL}_{fields} and $S^{APP}_{methods}$ is contained in $S^{TPL}_{methods}$ (the method signatures are matched using regular rules). We consider this class pair qualified for the next steps of class correspondence detection if they meet the requirements. The reason is that when unused methods and fields are removed from a class, the remaining methods and fields should stay within the original class.

4.2 Similarity-Based Method Matching

The signature-based class matching step might match multiple app classes to a single TPL class and vice versa, potentially leading to incorrect matches. Therefore, in this step, LibHunter try to match methods between the app class and TPL class to further determine if some correspondences are false. Consequently, this process eliminates incorrect class matches and reduces the instances where multiple app classes are incorrectly matched to one TPL class.

Typical inlining will integrate the callees into a caller when certain conditions are met. Since the caller and callees might originally belong to different classes, an app class could contain code from multiple TPL classes. Therefore, we do not seek direct class-level feature matching but rather based on method matching. Specifically, LibHunter utilizes opcodes and strings within methods for method matching because these fundamental features are commonly used in other works for assessing method similarity [53, 59]. We categorize matching methods into two types: Fully Matching and Partially Matching. Fully Matching refers to instances where the similarity between an app method and a TPL method exceeds a threshold, resulting in a direct match. For Partially Matching, it refers to situations where the app method, having inlined other methods, contains features of the caller method in TPL. For instance, if a TPL method `x1` inlines another method `x2` and synthesizes into app method `y`, then the features of `x1` should be contained in the features of `y`. These partially matched methods undergo final matching in the Cross-Inlining Method Matching phase.

For an app class c_a and a TPL class c_t, assume that m_a and m_t are a method within c_a and c_t, respectively. if their fuzzy signatures match (through regular rule), they are considered as Fully Matching if they satisfy the following equation:

$$sim_{full}(m_a, m_t) = \frac{JD(m_a^{op}, m_t^{op})}{2} + \frac{JD(m_a^{str}, m_t^{str})}{2} \geq T_1 \quad (1)$$

where m_a^{op} and m_a^{str} are the opcodes set and strings set within m_a respectively; m_t^{op} and m_t^{str} are the opcodes and strings for m_t; the JD calculates the Jaccard similarity [42] between two feature sets; T_1 is the predefined threshold at method level. We set the weights for opcodes and strings at 0.5 each because both contribute to the accuracy of method matching and the 0.5 helps avoid bias toward either instructions or data content. According to the formula, a

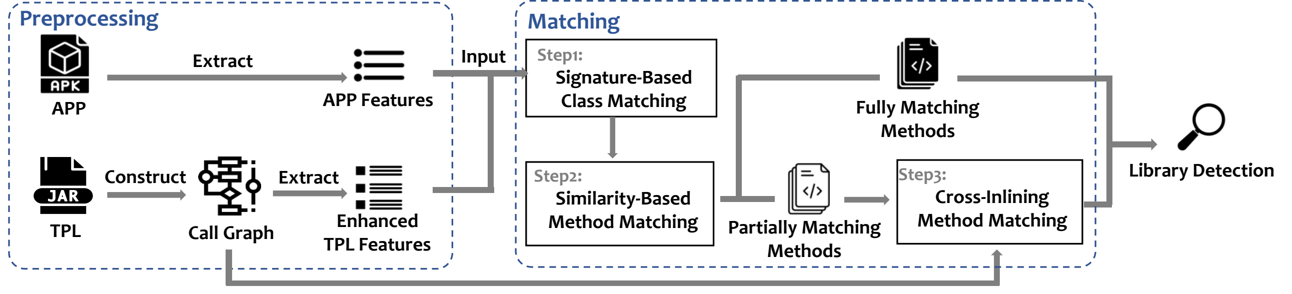


Figure 3: Workflow of LibHunter

higher similarity score indicates that the two methods share similar features. Theoretically, each TPL method should be matched by, at most, one app method. Therefore, if multiple app methods from the same class match a single TPL method, LibHunter identifies the app method with the highest sim_{full} value as the final match to the TPL method.

Then, we further identify partially matched methods within c_a and c_t that are not fully matched. Since an app method may inline other methods, we assess how many features of the TPL method are contained in the app method. Specifically, we consider two methods that satisfy the following formula to be partially matched.

$$sim_{partial}(m_a, m_t) = \frac{|m_a^{op} \cap m_t^{op}|}{|m_t^{op}| \times 2} + \frac{|m_a^{str} \cap m_t^{str}|}{|m_t^{str}| \times 2} \geq T_1 \quad (2)$$

For an app method m_a, we do not select the TPL method with the highest $sim_{partial}$. Instead, we include all TPL methods in c_t that satisfy Equation 2 into a one-to-many list L_{m_a}. This is because some incorrect TPL methods only contain a few instructions (e.g., return), leading to a high $sim_{partial}$ value. We will determine the final matches in the Cross-Inlining Method Matching phase.

In LibHunter, TPL features are extracted directly from their binaries, and thus, they contain a complete set of class features. However, in-app versions of a TPL class may contain only a small subset of features due to code shrinking. Therefore, LibHunter only considers an app class c_a as a candidate match to a library class c_t if all methods in c_a match at least one method in c_t (either fully or partially). This is because code shrinking does not affect this index. This can help eliminate certain incorrect class correspondences.

4.3 Cross-Inlining Method Matching

To further identify the corresponding TPL method m_t from the list L_{m_a} for app method m_a, we attempt to model the method inlining process. This helps us track how certain TPL methods are synthesized into an app method.

We have already introduced the R8 Inlining strategy in Section 3.4. Following this strategy, we can analyze which methods are likely to inline their callees within TPL binaries. For each TPL method m_t in L_{m_a}, we generate its call chains on the call graph using depth-first traversal. We then attempt to synthesize m_t and its call chains, referring to the synthesized method as $m_t{'}$. Specifically, when encountering a function call, we determine whether to inline the callee. If the function call falls under the "Single Caller" category or the size of callee meets the criteria defined in "Multi Caller," LibHunter decides to inline it and appends it into the call chains of m_t. Note that inlining process is iterative, meaning each callee may also inline its own callees. We skip the standard library calls because our analysis scope is limited to the TPL binary. We will extract the opcode and string set of $m_t{'}$ as $m_t^{op'}$ and $m_t^{str'}$, respectively. If method m_a and $m_t^{str'}$ satisfy the following formula, we consider they are cross-inlining matched:

$$sim_{cross}(m_a, m_t{'}) = \frac{JD(m_a^{op}, m_t^{op'})}{2} + \frac{JD(m_a^{str}, m_t^{str'})}{2} \geq T_1 \quad (3)$$

If multiple TPL methods in L_{m_a} meet this condition, we select the TPL m_t method with the highest sim_{cross} value as the final match method.

4.4 TPL Version Identification

After determining the final method matches, we will integrate the results of Fully Matching and Cross-Inlining Matching to decide the final class matches and further identify the TPL version. Specifically, given an app class c_a and one of its candidate TPL classes c_t, let $MF_{c_a,c_t} = Fully(m_a, m_t)$ be the set of Fully Matching methods between the two classes, where m_a represents a method in the c_a and m_t represents a method in the c_t. Similarly, let $MC_{c_a,c_t} = Cross(m_a, m_t^{'})$ be the set of Cross Matching methods between the two classes, where $m_t^{'}$ is our synthesized method for m_t. We then compute the class correspondence confidence $Confidence(c_a, c_t)$, which is calculated as follows:

$$\begin{aligned} Confidence(c_a, c_t) = & \sum\nolimits_{(m_a,m_t) \in MF_{c_a,c_t}} size(m_t) \\ & + \sum\nolimits_{(m_a,m_t) \in MC_{c_a,c_t}} size(m_t^{'}) \end{aligned} \quad (4)$$

Where the $size(m)$ calculates the opcodes number for method m. When computing the confidence, we use the synthesized method $m_t^{'}$ instead of the original m_t. This is because the callees in the call-chains of m_t also share their features to help identify the corresponding app method m_a although these callees do not have an actual definition structure. Then, among the candidate TPL classes for app class, the class pair with the highest class correspondence confidence is considered the final matched classes. Assume all matched class pairs are stored in $CM(app, tpl)$. We further consider a TPL to be present in the app if it satisfies the following formula:

$$Sim(TPL, app) = \frac{\sum_{(c_a,c_t)\in CM(app,tpl)} Confidence(c_a, c_t)}{csize(app)} \geq T_2 \quad (5)$$

Where $csize(app)$ calculates the total number of opcodes in the app and T_2 is the threshold at the library level. According to the formula, the more opcodes shared between the TPL and the app, the more likely the TPL is present in the app. If multiple versions of the same TPL satisfy the formula 5, we select the version with the largest similarity score as the final result for that TPL. This includes providing the identified TPLs with group ID, artifact ID, and version number.

5 Evaluation

This section introduces the experiments performed to evaluate the effectiveness and usefulness of LibHunter. We first determine the thresholds (i.e., T_1 and T_2) under a subset of apps (40×5) in $dataset_1$. To avoid the overfitting problem, only the other apps in $dataset_1$ are used to conduct the following experiments.

Then, we attempt to answer the following question:

- **RQ4 (Effectiveness):Can LibHunter outperform state-of-the-art methods in terms of optimized, obfuscated, and shrunk apps?** We compare LibHunter with baselines to assess and compare their effectiveness. We also apply LibHunter to *dataset2* to evaluate its effectiveness in resisting various optimization strategies.
- **RQ5 (Components' Contribution): How is the contribution of the major components of LibHunter?** We evaluate two main components of LibHunter: the enhanced TPL features and cross-inlining method matching. We aim to determine if these components can effectively enhance the performance of LibHunter against optimization.
- **RQ6 (Usefulness): Can LibHunter be applied to real-world Android apps for vulnerability detection?** Since LibHunter can detect the versions of TPL, we can refer to the NVD [50] to identify if the detected versions are affected by known vulnerabilities. If LibHunter finds that an app uses certain vulnerable versions, it will report a warning, aiding security analysts in identifying vulnerabilities.

5.1 Experiment Setup

Implementation. We implemented LibHunter in Python. We first convert TPLs from ".jar/.aar" files to ".dex "(Dalvik bytecode) files using D8 [47], and then use Androguard [9] to extract the features of the TPLs (.dex) and the app (.apk). We can store the extracted features of the TPLs in local files, allowing us to reuse these files during the detection process without repeatedly parsing the TPLs. Moreover, our evaluation was conducted on a Linux Server equipped with two Intel(R) Xeon(R) Gold 6248R CPUs and 256GB RAM.

Dataset and Metrics. We used the datasets introduced in Section 3.1 ($dataset_1$ and $dataset_2$) to evaluate the performance of LibHunter. Recall that $dataset_1$ includes (200×5) app variants. To determine appropriate thresholds, we randomly selected 20% of the apps (i.e., 40×5). These apps, used for tuning parameters, involve 61×5 APP-TPL pairs. To avoid bias, all remaining apps and pairs will be used for further evaluation, involving 318×5 APP-TPL pairs. We mark this part of the dataset as $dataset_1'$. Then we use F1 value as metric to evaluate the performance of LibHunter.

Thresholds Selection. LibHunter introduces two thresholds. The T_1 is used to set the method-level threshold. An app method and a TPL method are only considered matched if their similarity exceeds this threshold. The second threshold, T_2, is at the TPL level. If their similarity between the TPL and app below this threshold, the TPL is considered not present in the app. Following existing studies [54, 61], we employ *Grid Search* [28] to empirically determine the thresholds by optimizing the F1 value. Specifically, we initially set both thresholds to 0 and gradually adjust them in steps of 0.05 up to an upper bound of 1. We then select the values that achieve the highest F1 value. Consequently, we choose the pair $T_1 = 0.75$ and $T_2 = 0.2$ to achieve the highest F1 value. These thresholds are used to conduct the following experiments.

5.2 RQ4: Effectiveness of LibHunter

Overall Results. Table 3 shows the comparison results between LibHunter and other baselines in terms of TPL detection *w.r.t.* F1 value. We can find that LibHunter demonstrates strong performance across various app variants. At library level, LibHunter achieves the F1 value of 92.4%, 92.7%, and 85.7% for D8-compiled apps, obfuscated apps, and shrunk apps, respectively. Notably, LibHunter excels in challenging scenarios involving optimized apps (i.e., "Opt+Srk" and "Opt+Obf+Srk"), it can achieve the F1 value of 71.9% and 71.4%, respectively. On average, the F1 value of LibHunter outperforms the best baselines by 29.3% (71.4%-42.1%) for "Opt+Obf+Srk" apps. Regarding the version level, the performance of LibHunter remains superior in most scenarios. In particular, LibHunter achieves the F1 value of 89.8%, 89.2%, and 79.5% for D8-compiled apps, obfuscated apps, and shrunk apps, respectively. We found that LibScout achieves the best results on "Opt+Srk" apps, with an F1 value of 55.6%. LibScout performs well on "Opt+Srk" apps not because it is specifically designed for code optimization, but because it relies solely on the package hierarchies of TPL for detection, and code optimization strategies do not affect the package hierarchies. For instance, if a TPL (e.g., okhttp3) has package hierarchies containing okhttps/internal/cache2, LibScout uses graph matching algorithms to detect whether similar package hierarchies exist in the target app. In contrast, LibHunter delivers the best results, improving the F1 value significantly over the best baseline by 36.1% (51.1%-15.0%). However, in challenging scenarios (i.e., "Opt+Obf+Srk"), LibScout's F1 value dramatically drops to 1.4%. For these apps, R8 enables code obfuscation by default, which significantly alters the package hierarchies of the target app. As a result, the performance of LibScout is greatly diminished. These results significantly outperform those of other tools, highlighting LibHunter 's effectiveness and practicality in handling apps with various code transformations. This makes it an effective tool for detecting TPLs for apps.

The promising results achieved by LibHunter are largely due to its optimization-resilient features. In contrast, other TPL detection tools were designed without considering code optimizations, focusing instead solely on obfuscation and shrinking (see Section 3.4). Thus, it is difficult for them to resist the effects of optimization.

Table 3: Comparison on the Effectiveness of TPL Detection Tools on $dataset_1^{'}$

Level	App Variants	LibHunter	LibScan	LibPecker	LibScout	LibID
Library	D8-compiled	**92.4%**	91.8%	84.2%	77.6%	81.9%
	Obf	**92.7%**	91.6%	33.1%	1.4%	3.2%
	Srk	**85.7%**	73.3%	43.8%	16.5%	10.3%
	Opt+Srk	**71.9%**	42.1%	3.9%	56.9%	0.8%
	Opt+Obf+Srk	**71.4%**	42.1%	4.0%	1.4%	0.8%
Version	D8-compiled	**89.8%**	86.0%	81.2%	76.4%	79.3%
	Obf	**89.2%**	84.9%	29.0%	1.4%	2.4%
	Srk	**79.5%**	51.1%	30.7%	13.3%	9.3%
	Opt+Srk	51.9%	15.5%	3.3%	**55.6%**	0.8%
	Opt+Obf+Srk	**51.1%**	15.0%	3.4%	1.4%	0.8%

Table 4: Comparison on Different Optimization Strategies

	Library Level		Version Level	
Strategy	**LibHunter**	**LibScan**	**LibHunter**	**LibScan**
CallSiteOpt.	**78.7%**	42.6%	**62.4%**	22.7%
Devirtual.	**76.1%**	57.1%	**60.2%**	42.3%
EnumUnbox.	**76.8%**	57.4%	**59.0%**	43.8%
EnumVal.Opt.	**78.1%**	57.3%	**61.7%**	45.4%
Init.Class.Anal.	**77.8%**	59.1%	**60.5%**	46.2%
ClassInlining	**75.5%**	57.2%	**63.5%**	44.2%
Inlining	**82.5%**	41.7%	**69.7%**	20.8%
NameRefl.Opt.	**79.1%**	58.0%	**60.3%**	46.5%
SideEffectAnal.	**77.6%**	57.6%	**59.0%**	45.2%
Str.Concaten.Opt	**77.8%**	57.3%	**60.5%**	46.9%
Outlining	**77.1%**	59.9%	**61.9%**	43.5%
Horiz.ClassMerg.	**81.3%**	59.3%	**63.5%**	46.6%
Vertic.ClassMerg.	**77.1%**	57.9%	**61.3%**	45.4%

Resilience to different optimization strategies. We compare LibHunter with other baselines on apps optimized with different strategies, as shown in Table 4. The results reveal that LibHunter outperforms the baselines across all optimization strategies. Specifically, at the library level, LibHunter achieves F1 values of 78.7% and 82.5% for CSO and Inlining, respectively. At the version level, it achieves F1 values of 62.4% and 69.7% for these strategies. In contrast, the F1 values of LibScan drops to 22.7% and 20.8%, respectively. This highlights LibHunter's effectiveness in countering CSO and Inlining.

We find that LibHunter also excels with other strategies, notably outperforming LibScan. For instance, in scenarios involving *String-Concatenation*, which directly converts string concatenation operations into final strings to reduce memory usage, LibHunter achieves a 77.8% F1 value at the library level, surpassing LibScan's 57.3%. This is because LibHunter uses opcodes and strings of methods to calculate similarity for method matching. In contrast, LibScan requires that an app's method opcodes contain those of a TPL method for a match. Many optimization strategies, such as *StringConcatenation*, will modify opcodes, thereby making it challenging for the app methods to contain the corresponding TPL method's opcodes. These results underscore the robustness of LibHunter's approach against common optimization strategies.

Table 5: F1 Scores achieved by LibHunter and its variants

Level	App Variants	LibHunter	LibHunter$_c$	LibHunter$_i$
Library	D8-compiled	**92.4%**	92.2%	91.6%
	Obf	**92.7%**	92.4%	92.1%
	Srk	**85.7%**	85.3%	86.7%
	Opt+Srk	**71.9%**	66.3%	60.1%
	Opt+Obf+Srk	**71.4%**	65.9%	59.5%
Version	D8-compiled	**89.8%**	89.8%	89.4%
	Obf	**89.2%**	88.8%	89.0%
	Srk	**79.5%**	78.3%	77.8%
	Opt+Srk	**51.9%**	45.7%	40.8%
	Opt+Obf+Srk	**51.1%**	47.7%	39.9%

5.3 RQ5: Contribution of Major Components

Two major components that contribute to LibHunter's performance are the enhanced TPL features and the Cross-Inlining Method Matching component. In this RQ, we aim to investigate the contributions of these components separately by designing two variants and comparing them with LibHunter on $dataset_1^{'}$.

- **LibHunter$_c$**: This variant disables the enhanced TPL features. Specifically, when constructing the fuzzy signature of TPL methods, we do not enhance them into regular rules. Instead, we use the original fuzzy signature. This variant helps evaluate whether the enhanced TPL features can mitigate the impact of CSO.
- **LibHunter$_i$**: This variant disables the Cross-Inlining Method Matching component. Specifically, when calculating class correspondence confidence using formula 4, we no longer consider the results of Cross Matching methods (i.e., MC_{c_a,c_t}). Instead, we only use the results of Fully Matching methods (i.e., MF_{c_a,c_t}) to calculate the confidence. This variant helps assess whether this component can mitigate the impact of Inlining.

Results. The results are shown in Table 5. In particular, LibHunter and its variants LibHunter$_c$ and LibHunter$_i$ achieve comparable results for non-optimized apps. For D8-compiled, obfuscated, and shrunk apps, LibHunter and its variants LibHunter$_c$ and LibHunter$_i$ achieve similar results, with all F1 values experiencing minimal declines of less than 1% for both library and version level. This indicates that the removal of these components does not affect performance on non-optimized apps.

However, when optimization is enabled, as shown in the results of "Opt+Srk" and "Opt+Obf+Srk", the F1 value of LibHunter$_c$ drops to 66.3% and 65.9% at library level. In comparison, LibHunter can outperform it by 5.6% (71.9%-66.3%) and 5.5% (71.4%-65.9%) for these app variants. This is because our enhanced TPL features improve class and method matching accuracy. In practice, we found that an optimized app might have thousands of method parameters altered by CSO. LibHunter$_c$ ignores these parameter changes and uses the original fuzzy signature, leading to mismatches for methods.

The F1 value of LibHunter$_i$ is compromised when optimization is enabled. LibHunter$_i$ achieves an F1 value of 60.1% and 59.5% for "Opt+Srk" and "Opt+Obf+Srk" apps at library level, respectively. Therefore, LibHunter can outperform this variant by 11.8% (71.9%-60.1%) and 11.9% (71.4%-59.5%) for these optimized apps. After analyzing some of the missed cases, we found not using this component can lead to missed method matches. For instance, suppose multiple

TPL methods being synthesized into a single app method through Inlining. Without this component, these TPL methods fail to find corresponding app methods, thus not contributing to the overall similarity between the TPL and the app. This demonstrates that this component can enhance the accuracy of method matching, thereby increasing the overall TPL detection performance.

5.4 RQ6: Usefulness of LibHunter

TPL detection tools can be used to identify vulnerabilities by pinpointing the TPL versions [12, 30, 32, 46, 59, 61, 63]. For each detected TPL, if its version lies in the affected version range of a known CVE, the tool will issue a warning (i.e., an App-CVE pair). To analyze the usefulness of LibHunter, we conduct a comprehensive study on the top 10,000 Google Play apps, aiming to uncover the real-world risks posed by vulnerable TPLs. For comparison, we also applied LibScan to these apps to detect vulnerable libraries.

Dataset Collection. We collected commercial apps from Google Play based on their number of installations and crawled the top 10,000 popular apps. Additionally, we used 31 vulnerable TPLs (introduced in Section 3.1) for reference checking. These 31 TPLs involve a total of 94 CVEs. We searched the NVD [50] to find the TPL versions affected by these 94 CVEs. Due to page limitations, we put detailed information about these 31 TPLs and their associated CVEs on our website [37]. We then applied LibHunter to analyze these apps. If LibHunter detects that an app used a vulnerable TPL version, it reports a warning (i.e., an App-CVE pair).

Experimental Results. In our collection of 10,000 Google Play apps, LibHunter identifies that 18.6% (1,855/10,000) of them used 31 vulnerable TPLs, corresponding to 8,659 versions of these TPLs. Among these 8,659 TPL versions, 43.4% (3,761/8,659) versions were found to be affected by the 94 known CVEs (note that a single TPL version can be linked to multiple CVEs), leading LibHunter to report 3,761 warnings. We found that LibHunter can process an app every 15 minutes on average, demonstrating its scalability. As a comparison, we found that LibScan identified only 8.3% (829 out of 10,000) of the apps as involving the 31 vulnerable TPLs, ultimately generating 2,039 warnings. We observed that the lower number of warnings reported by LibScan compared to LibHunter is mainly **due to its inadequacy in detecting optimized apps.** However, many apps on Google Play are protected using R8, which challenges the effectiveness of LibScan.

Upon closely examining the warnings reported by LibHunter, we found that the two most affected TPLs were *Okhttp* [39] and *Retrofit* [40]. This is because a large number of apps rely on these networking libraries to communicate with the remote server. Unfortunately, *Okhttp* has involved two vulnerabilities: CVE-2016-2402, and CVE-2021-0341. *Retrofit* is also affected by CVE-2018-1000850. The failure of these apps to promptly update the TPL to the latest versions may pose potential threats to users. We understand that the existence of vulnerable TPLs doesn't necessarily mean the vulnerabilities can be exploited since R8 might remove the affected methods during shrinking process. However, security analysts can still further analyze the warnings using other tools (e.g., constructing PoCs or patch presence test tools [27, 55]), thus simplifying their efforts for inspecting the vulnerability.

6 Discussion

Limitations. This study is constrained by two main limitations: (1) *The inability to model all optimization strategies.* While we have designed components against the two most impactful optimization strategies (i.e., CSO and Inlining), we have not designed components to resist other strategies that could also affect the performance of TPL detection tools. For instance, the *HorizontalClassMerger* strategy merges multiple classes with the same characteristics (e.g., the same number and type of fields) into a single class, which makes it challenging for LibHunter to match TPL and app classes accurately. However, the two strategies we focused on, CSO and Inlining, have been proven to be the most influential on TPL detection, as discussed in Section 3.4. Additionally, LibHunter has also demonstrated good results against other optimization strategies (see Section 5.3). LibHunter's ability to achieve high F1 scores compared to previous tools proves its resilience to optimization. The core insight of our approach is that certain optimization processes can be approximated and inferred via fine-grained static analysis, and our approximation process can be generalized to other strategies. For example, for NameReflection, this optimization transforms certain reflection calls into direct function calls. We can infer which reflection calls are likely to be converted by following its mechanism, thereby extracting enhanced TPL features accordingly via static analysis. However, we acknowledge that modeling all optimizations with static analysis is undoubtedly challenging and effort-consuming, we leave this task as our important future work. (2) *Bias of obfuscation strategies.* This work focuses on investigating the impact of R8 on existing TPL detection tools, hence it does not consider other advanced obfuscators. Commercial obfuscators can offer sophisticated features to protect code against reverse engineering. For instance, DashO [7] provides control flow randomization by inserting redundant constraints and functions, which reduce the readability of the code. Some obfuscators can also obfuscate strings to reduce code readability. However, our target code protection tool, R8, is the default tool in the Android build process. Our empirical study in RQ1 shows that over 41.1% of apps use R8 for distribution (see Section 3.2). The effectiveness of LibHunter in resisting code optimization underscores its usefulness in detecting TPL versions.

7 Conclusion

App vendors must ensure the security of apps while developers often protect their code before release. Therefore, identifying TPL versions is essential. However, existing TPL detection tools do not account for the impact of code optimization. Therefore, we have conducted a systematic study to explore how code optimization affects the performance of these tools. Our findings reveal that optimization significantly influences their effectiveness. Based on these findings, we have developed LibHunter designed to withstand the effects of code optimization, obfuscation, and shrinking. We evaluated LibHunter on a large dataset, and the results show that LibHunter can resist advanced code optimization strategies.

Acknowledgments

We sincerely thank all anonymous reviewers for their valuable feedback and guidance in improving this paper. This work was sponsored by National Natural Science Foundation of China (No.62372193).

References

[1] Allatori. 2024. https://allatori.com/. Accessed: 2024-06.

[2] Sumaya Almanee, Arda Ünal, Mathias Payer, and Joshua Garcia. 2021. Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1347–1359. https://doi.org/10.1109/ICSE43902.2021.00122

[3] 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, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 356–367. https://doi.org/10.1145/2976749.2978333

[4] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 1021–1036. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bhoraskar

[5] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin T. Vechev. 2016. Statistical Deobfuscation of Android Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 343–355. https://doi.org/10.1145/2976749.2978422

[6] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*. 175–186.

[7] DashO. 2024. https://www.preemptive.com/products/dasho/. Accessed: 2024-06.

[8] 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, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2187–2200. https://doi.org/10.1145/3133956.3134059

[9] Anthony Desnos, Geoffroy Gueguen, and Sebastian Bachmann. 2015. Androguard: Reverse engineering, malware and goodware analysis of android applications... and more (ninja!).

[10] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, 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. In *Security and Privacy in Communication Networks - 14th International Conference, SecureComm 2018, Singapore, August 8-10, 2018, Proceedings, Part I (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Vol. 254)*, Raheem Beyah, Bing Chang, Yingjiu Li, and Sencun Zhu (Eds.). Springer, 172–192. https://doi.org/10.1007/978-3-030-01701-9_10

[11] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/towards-measuring-supply-chain-attacks-on-package-managers-for-interpreted-languages/

[12] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. 2019. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/automating-patching-of-vulnerable-open-source-software-versions-in-application-binaries/

[13] Yue Duan, Lian Gao, Jie Hu, and Heng Yin. 2019. Automatic Generation of Non-intrusive Updates for {Third-Party} Libraries in Android Applications. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 277–292.

[14] William Enck, Damien Octeau, Patrick D. McDaniel, and Swarat Chaudhuri. 2011. A Study of Android Application Security. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association. http://static.usenix.org/events/sec11/tech/full_papers/Enck.pdf

[15] F-Droid: Free and Open Source Software. 2024. https://f-droid.org. Accessed: 2024-06.

[16] 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, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 638–648. https://doi.org/10.1145/3106237.3106305

[17] 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, WISEC 2012, Tucson, AZ, USA, April 16-18, 2012*, Marwan Krunz, Loukas Lazos, Roberto Di Pietro, and Wade Trappe (Eds.). ACM, 101–112. https://doi.org/10.1145/2185448.2185464

[18] Guardsquare. 2024. Configuration - Optimizations. https://www.guardsquare.com/manual/configuration/optimizations. Accessed: 2024-06.

[19] Guardsquare. 2024. Configuration - Usage. https://www.guardsquare.com/manual/configuration/usage. Accessed: 2024-06.

[20] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 421–431. https://doi.org/10.1145/3180155.3180228

[21] Qiang He, Bo Li, Feifei Chen, John C. Grundy, Xin Xia, and Yun Yang. 2022. Diversified Third-Party Library Prediction for Mobile App Development. *IEEE Trans. Software Eng.* 48, 2 (2022), 150–165. https://doi.org/10.1109/TSE.2020.2982154

[22] Jie Huang, Nataniel P. Borges Jr., Sven Bugiel, and Michael Backes. 2019. Up-To-Crash: Evaluating Third-Party Library Updatability on Android. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*. IEEE, 15–30. https://doi.org/10.1109/EUROSP.2019.00012

[23] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. 2017. The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1037–1049. https://doi.org/10.1145/3133956.3134064

[24] Nasif Imtiaz, Seaver Thorn, and Laurie A. Williams. 2021. A comparative study of vulnerability reporting by software composition analysis tools. In *ESEM '21: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 11-15, 2021*, Filippo Lanubile, Marcos Kalinowski, and Maria Teresa Baldassarre (Eds.). ACM, 5:1–5:11. https://doi.org/10.1145/3475716.3475769

[25] Haoxiang Jia, Ming Wen, Zifan Xie, Xiaochen Guo, Rongxin Wu, Maolin Sun, Kang Chen, and Hai Jin. 2023. Detecting JVM JIT Compiler Bugs via Exploring Two-Dimensional Input Spaces. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 43–55. https://doi.org/10.1109/ICSE48619.2023.00016

[26] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 224:1–224:13. https://doi.org/10.1145/3597503.3639100

[27] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 1149–1163. https://doi.org/10.1145/3372297.3417240

[28] Steven M. LaValle, Michael S. Branicky, and Stephen R. Lindemann. 2004. On the Relationship between Classical Grid Search and Probabilistic Roadmaps. *Int. J. Robotics Res.* 23, 7-8 (2004), 673–692. https://doi.org/10.1177/0278364904045481

[29] Bodong Li, Yuanyuan Zhang, Juanru Li, Runhan Feng, and Dawu Gu. 2019. APPCOMMUNE: Automated Third-Party Libraries De-duplicating and Updating for Android Apps. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 344–354. https://doi.org/10.1109/SANER.2019.8668009

[30] 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 *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 335–346. https://doi.org/10.1109/ICSE.2017.38

[31] Apache Log4j2. 2024. https://github.com/apache/logging-log4j2. Accessed: 2024-06.

[32] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: fast and accurate detection of third-party libraries in Android apps. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 653–656. https://doi.org/10.1145/2889160.2889178

[33] Google Play. 2024. https://play.google.com/. Accessed: 2024-06.

[34] Proguard. 2024. https://www.guardsquare.com/proguard. Accessed: 2024-06.

[35] R8. 2024. https://developer.android.google.cn/studio/build/shrink-code#optimization. Accessed: 2024-06.

[36] Google Maven Central repository. 2024. https://maven.google.com/. Accessed: 2024-06.

[37] LibHunter Repository. 2024. https://github.com/CGCL-codes/LibHunter. Accessed: 2024-06.

[38] Maven Central repository. 2024. https://www.maven.org/. Accessed: 2024-06.

[39] Okhttp Repository. 2024. https://square.github.io/okhttp/. Accessed: 2024-06.

[40] Retrofit Repository. 2024. https://github.com/square/retrofit. Accessed: 2024-06.

[41] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. 2012. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, Tadayoshi Kohno (Ed.). USENIX Association, 553–567. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/shekhar

[42] Jaccard Similarity. 2024. https://en.wikipedia.org/wiki/Jaccard_index. Accessed: 2024-06.

[43] "Statista". 2024. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Accessed: 2024-06.

[44] Android Studio. 2024. Shrink, obfuscate, and optimize your app. https://developer.android.com/build/shrink-code. Accessed: 2024-06.

[45] 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. *Comput. Secur.* 80 (2019), 257–272. https://doi.org/10.1016/J.COSE.2018.07.024

[46] 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. *Comput. Secur.* 80 (2019), 257–272. https://doi.org/10.1016/j.cose.2018.07.024

[47] D8 Tool. 2024. https://developer.android.com/studio/releases/platform-tools. Accessed: 2024-06.

[48] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. 2021. Supply-Chain Vulnerability Elimination via Active Learning and Regeneration. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 1755–1770. https://doi.org/10.1145/3460120.3484736

[49] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. 2021. Supply-Chain Vulnerability Elimination via Active Learning and Regeneration. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 1755–1770. https://doi.org/10.1145/3460120.3484736

[50] National vulnerability database. 2024. https://nvd.nist.gov. Accessed: 2024-06.

[51] 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, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 222–235. https://doi.org/10.1145/3274694.3274726

[52] Apps with most third-party libraries. 2024. http://privacygrade.org/third_party_libraries. Accessed: 2024-06.

[53] 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. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 3385–3402. https://www.usenix.org/conference/usenixsecurity23/presentation/wu-yafei

[54] Liang Xiao, Ruili Wang, Bin Dai, Yuqiang Fang, Daxue Liu, and Tao Wu. 2018. Hybrid conditional random field based camera-LIDAR fusion for road detection. *Inf. Sci.* 432 (2018), 543–558. https://doi.org/10.1016/j.ins.2017.04.048

[55] Zifan Xie, Ming Wen, Haoxiang Jia, Xiaochen Guo, Xiaotong Huang, Deqing Zou, and Hai Jin. 2023. Precise and Efficient Patch Presence Test for Android Applications against Code Obfuscation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 347–359. https://doi.org/10.1145/3597926.3598061

[56] Zifan Xie, Ming Wen, Shiyu Qiu, and Hai Jin. 2024. Validating JVM Compilers via Maximizing Optimization Interactions. In *Proceedings of the the ACM Conference on Architectural Support for Programming Languages and Operating Systems*.

[57] Jing Yang, Yibiao Yang, Maolin Sun, Ming Wen, Yuming Zhou, and Hai Jin. 2022. Isolating Compiler Optimization Faults via Differentiating Finer-grained Options. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 481–491. https://doi.org/10.1109/SANER53432.2022.00065

[58] Geunha You, Gyoosik Kim, Seong-je Cho, and Hyoil Han. 2021. A Comparative Study on Optimization, Obfuscation, and Deobfuscation tools in Android. *J. Internet Serv. Inf. Secur.* 11, 1 (2021), 2–15. https://doi.org/10.22667/JISIS.2021.02.28.002

[59] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. ATVHUNTER: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1695–1707. https://doi.org/10.1109/ICSE43902.2021.00150

[60] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. 2022. Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review. *IEEE Trans. Software Eng.* 48, 10 (2022), 4181–4213. https://doi.org/10.1109/TSE.2021.3114381

[61] Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. 2019. LibID: reliable identification of obfuscated third-party Android libraries. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 55–65. https://doi.org/10.1145/3293882.3330563

[62] Xiao Zhang, Amit Ahlawat, and Wenliang Du. 2013. AFrame: isolating advertisements from mobile applications in Android. In *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, Charles N. Payne Jr. (Ed.). ACM, 9–18. https://doi.org/10.1145/2523649.2523652

[63] 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 *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 141–152. https://doi.org/10.1109/SANER.2018.8330204