Towards Global Matches for Third-Party Library Detection in Android

Lige Zhan*
Wuhan University
Wuhan, Hubei, China
ligezhan@whu.edu.cn

Jiang Ming†
Tulane University
New Orleans, Louisiana, USA
jming@tulane.edu

Chenke Luo†
Tulane University
New Orleans, Louisiana, USA
cluo6@tulane.edu

Guojun Peng*
Wuhan University
Wuhan, Hubei, China
guojpeng@whu.edu.cn

Jianming Fu*‡
Wuhan University
Wuhan, Hubei, China
jmfu@whu.edu.cn

Abstract

The detection of third-party libraries (TPLs) is pivotal in upholding the security of Android supply chains. A significant hurdle in this domain pertains to ensuring accurate detection, particularly in the face of prevalent code obfuscation techniques. Recently, multiple research endeavors have advocated for the adoption of fuzzy library signatures to accommodate specific code features that can be affected by code obfuscation. Despite the potential promise of this approach, the use of such abstractions often results in the identification of multiple candidates that bear a striking resemblance. Unfortunately, prevailing methodologies tend to rely on local maxima of similarity matches, disregarding the broader contextual knowledge encapsulated within surrounding classes. This oversight culminates in suboptimal matching outcomes.

The key observation motivating our research is that knowledge about surrounding library classes can significantly enhance matching accuracy. In this regard, we aim to establish a correspondence between *sets* of classes in a given app and target library—leveraging this contextual information from surrounding classes enables a transition from local matches to global matches. We implemented our idea in a TPL detection tool, called *LibScope*. LibScope refines obfuscation-resilient library signatures and introduces a strategic iterative process (i.e., a back-and-forth game), thereby elevating the pairwise similarity between independent classes to similarities among sets of classes. We extensively evaluated LibScope with state-of-the-art TPL detection approaches published at top venues. The comparative results on two representative public benchmarks reveal that LibScope significantly outperforms its counterparts in terms of recall, precision, and accuracy in library version detection.

*Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University.
†Department of Computer Science, School of Science and Engineering, Tulane University.
‡Corresponding author

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3764541

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Supply chain security, Third-party library, Android

ACM Reference Format:
Lige Zhan, Jiang Ming, Chenke Luo, Guojun Peng, and Jianming Fu. 2026. Towards Global Matches for Third-Party Library Detection in Android. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3764541

1 Introduction

In contemporary society, the intricate interdependence of various sectors, encompassing governmental bodies, industrial entities, and the general populace, relies extensively on a sophisticated framework of open-source software components [1]. Each node within the software supply chain possesses the potential to exert substantial influence on the downstream flow of this complex network [2–4]. As a critical constituent of the software supply chain, third-party libraries (TPLs) find extensive utilization within Android applications (apps) due to their expansive functionalities and user-friendly nature. Remarkably, over 60% of the app's code comprises library code [5]. While TPLs significantly contribute to the efficiency of app development, they also introduce notable security threats [6–8].

A recent open-source security survey found that 78% of vulnerabilities in Android apps originate from TPLs [9]. Positioned upstream in the software supply chain, TPLs are prime targets for exploitation, enabling attacks such as backdoor injection, privacy invasion, and financial theft [10–14]. The CVE-2024-3094 vulnerability, with a maximum CVSS score of 10, exposed a supply chain attack affecting nearly *all* Linux distributions, potentially compromising numerous servers [15]. Similarly, the 2021 Log4j2 vulnerability in Apache threatened remote control over 60,000 open-source projects and impacted more than 70% of enterprise online systems [16]. These vulnerabilities extend beyond individual vendors, affecting all downstream software that integrates them. Detecting TPLs in Android apps is therefore crucial for mitigating supply chain risks [17], as identifying their categories and versions helps uncover known vulnerabilities and reduce the attack surface.

Recent advancements in TPL detection have primarily focused on similarity-based comparison approaches [18]. These methods

generate feature profiles for the TPL and app, comparing their similarity to determine the presence of TPL. The pioneering work, LibScout [19], employs fuzzy method signatures to mitigate the impact of identifier renaming obfuscation. However, it lacks robustness against other obfuscation, such as package flattening. Subsequent research has improved resilience against common obfuscation strategies [20]. For instance, LibLoom [21] utilizes Bloom Filters [22] to reframe similarity comparison as a set-inclusion problem, mitigating the effects of dead code removal. Additionally, LibScan [23] incorporates a comprehensive set of anti-obfuscation features to enhance the representation of comparison candidates.

Nowadays, the widespread adoption of commercial obfuscators [24–26], poses a significant challenge for TPL detection. Maintaining high detection accuracy in the presence of these obfuscators has become a central concern [27–29]. A common anti-obfuscation strategy involves extracting multiple fuzzy signatures to normalize code features that are susceptible to alteration. TPL detection is then formulated as a maximum similarity score matching problem. While recent approaches such as LibScan [23] continue to expand the set of fuzzy features, their use is a double-edged sword, potentially undermining detection accuracy [30]. This issue arises because fuzzy features often yield multiple highly similar candidates. For example, in TPL *Jsoup-1.14.1*, LibScan identifies ten candidates with similarity scores exceeding 90% for the class *HtmlTreeBuilderState*. However, selecting the final match solely based on the highest similarity often results in mismatches, as the selected candidates may lack intrinsic correlation. This disregard for contextual information ultimately leads to suboptimal outcomes.

In this paper, we attempt to obtain *global matches* for TPL detection by leveraging rich fuzzy features. Our key observation is that while considering individual class similarities is important, a holistic consideration of neighboring classes can lead to better matching outcomes. Hence, we broaden the scope of class similarity beyond the class itself to neighboring classes, which helps our approach capitalize on the wealth of contextual knowledge embedded within the surrounding classes, enabling a seamless transition from localized matching to a comprehensive global matching.

In particular, we develop a new TPL detection approach for Android apps, named *LibScope*. Our work involves refining existing fuzzy features, such as methods, fields, call relationships, to create matching candidates that are both obfuscation-resilient and efficient. We then conduct a strategic iterative comparison process to establish a correspondence between sets of classes in a given app and target library. Our algorithm is inspired by the principles of model theory's back-and-forth game [31]. This novel technique not only emphasizes the similarity between target match pairs, but also considers potential relationships with their neighboring candidates. As a result, LibScope excels at identifying global match pairs, ultimately yielding more precise and reliable detection outcomes.

To assess LibScope's effectiveness, we conducted experiments on two public benchmarks with ground truth and compared the results with state-of-the-art tools, including LibScout [19], LibID [32], LibLoom [21], and LibScan [23]. The results confirm LibScope's superiority across various obfuscation settings, achieving an 18.1% higher F1 score for TPL detection and a 33.4% improvement in version identification accuracy over the previous best tool. Additionally, our iterative back-and-forth gaming approach identified 20% more global matches compared to traditional local maxima algorithms, leading to improvements of 14.9% and 22.7% in TPL detection and version identification F1 scores, respectively. In summary, this paper makes the following contributions:

- LibScope focuses on achieving global match results by considering not only individual class similarities but also leveraging contextual information from neighboring classes, enhancing the overall quality of TPL detection.
- Our methodology empowers a global perspective, which not only augments the accuracy of TPL detection but also opens avenues for a more comprehensive understanding of the intricate relationships between software components within the supply chain.
- LibScope refines obfuscation-resilient features of the existing work and utilizes a more advanced algorithm for feature matching. Through comprehensive experiments, LibScope is shown to outperform state-of-the-art tools in various settings of commercial obfuscators.

Open Source We release LibScope's prototype and evaluation datasets at Github to facilitate reproduction and reuse,

2 Background, Related Work, and Motivation

In this section, we first provide background information of our study and its significance. Then, we review existing TPL detection approaches and identify their limitations (e.g., localized matching), which have prompted our research. Finally, we define the scope of the common obfuscation techniques that LibScope can withstand.

2.1 TPL Detection in Supply Chain Security

Android TPLs have significantly streamlined the app development process, emerging as an indispensable constituent of the software supply chain [33]. However, the security of the supply chain is inherently contingent on its weakest link. Positioned upstream in this intricate network, security vulnerabilities present within TPLs can potentially jeopardize the entire downstream dependency system [34, 35]. Insufficient attention to security concerns, such as the presence of backdoors or vulnerabilities within TPLs, poses a significant risk. Neglecting TPL detection allows security issues within these libraries to propagate to users' devices, perpetuating potential threats. Moreover, researchers focusing on security issues within non-library code confront challenges posed by TPLs, which introduce noise that can affect the accuracy of their analyses.

For these reasons, the investigation into Android TPL detection has garnered escalating significance [36, 37]. The primary aim is to *ascertain the use of specific TPLs and their respective versions within an app* [19]. The precision of TPL detection in apps significantly bolsters various security-related tasks that rely on TPL detection as a fundamental prerequisite. These tasks encompass a spectrum of activities, including repackaged app inspection [38–40], library isolation [41], malware detection [42], and patch update mechanisms [43–45]. Furthermore, the identification of specific in-app versions is imperative for tasks such as license violation detection [46] and TPL vulnerability assessment [47]. The cumulative efforts in these domains underscore the importance of TPL detection in fortifying and safeguarding supply chain security.

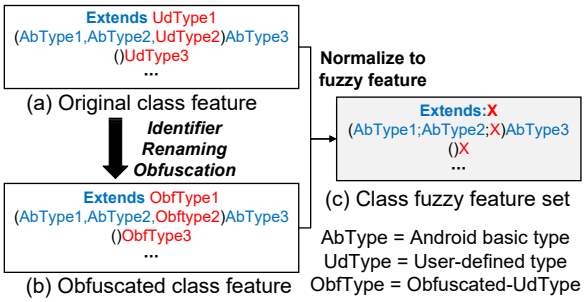

Figure 1: Fuzzy feature construction.

2.2 Similarity-Based TPL Detection Methods

Recent advancements in TPL detection have predominantly focused on measuring pairwise similarities between app and library features, with increasing emphasis on obfuscation resilience, feature granularity, and computational efficiency. For instance, LibScout [19] extracts parameter types and return value types of methods as features, strategically replacing non-Android basic types with placeholders to partially preserve features resistant to typical obfuscation techniques that refrain from renaming basic types. However, LibScout's reliance on constructing a special hash sequence (Merkle Tree [48]) based on package hierarchy renders it susceptible to obfuscation types like package flattening. Alternatively, LibID [32] leverages fuzzy features and basic blocks of the control flow graph as fine-grained distinguishing features. It utilizes the Binary Integer Programming (BIP) algorithm to maximize the number of unique matched pairs that fit the package hierarchy constraints. However, these constraints are established on the assumption that obfuscators do not alter package structure of apps, whereas obfuscation techniques such as package flattening can indeed do so [18]. In addition, the NP-hard nature of the BIP algorithm in LibID results in low computational efficiency, prompting the emergence of LibLoom [21]. LibLoom represents obfuscation-resilient features using a *Bloom Filter*, a specialized data structure that significantly improves comparison efficiency. The state-of-the-art work, LibScan [23], emphasizes the impediments imposed by modern obfuscators, which substantially impede the precise identification of TPLs. In response, LibScan constructs comprehensive fuzzy features to identify the best-matched candidate with the maximum similarity. Subsequently, it improves detection accuracy by comparing the call-chain relations.

2.3 Limitations of Fuzzy Feature Matching

Since the advent of LibScout [19], similarity-based comparison approaches have advocated for the incorporation of the so-called "fuzzy features" [18] to accommodate specific code attributes susceptible to alteration through Android obfuscation, such as identifier renaming. These methods involve extracting Java object types from disassembled code, preserving Android basic types that may persist during obfuscation, and normalizing user-defined types that might undergo obfuscation. Over time, the spectrum of potential fuzzy features has expanded to encompass a diverse range of multi-layer attributes, spanning class-level, field-level, and method-level features [23]. Figure 1 exemplifies the process of constructing fuzzy features. Initially, the return type and parameter types of the method

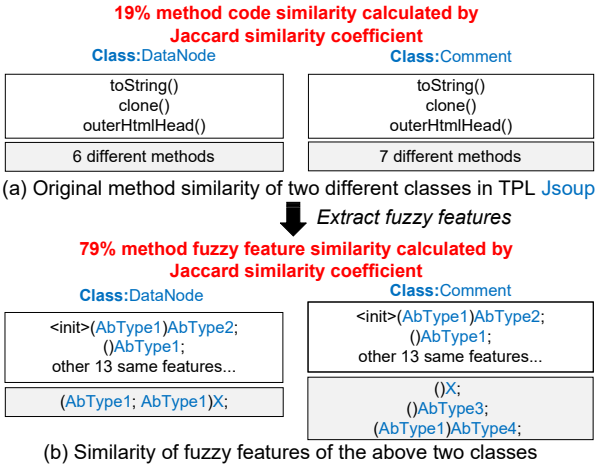

Figure 2: Fuzzy feature may lead to more similar candidates.

within the class structure are abstracted as Java object types (Figure 1(a) & (b)). Subsequently, within these types, the user-defined types are uniformly substituted by a common placeholder denoted as "X." In Figure 1(b), user-defined types undergo changes after identifier renaming. However, owing to the normalization of potentially obfuscated types using the same placeholder, fuzzy features exhibit the same characteristics even when these types are altered, as shown in Figure 1(c).

Expanding Matching Candidates While fuzzy features has been shown to notably enhance the efficacy of TPL detection against obfuscation, it is worth noting that the concurrent abstraction and normalization operations may also inadvertently bring about an unintended side effect: distinct classes may yield similar or even identical fuzzy features. The inclusion of placeholders further contributes to heightened feature generalization, augmenting the probability of encountering similar features. Figure 2 illustrates such an example, wherein two TPL classes, exhibiting a relatively low similarity score, present fuzzy features that are close to each other. In Figure 2(a), the overall similarity between these two classes is only 19%, calculated by the Jaccard similarity coefficient [49]. In contrast, upon representation using fuzzy features, these classes demonstrate a substantial surge in similarity, escalating sharply from 19% to 79%, as depicted in Figure 2(b). Consequently, the proliferation of highly similar candidates poses a significant challenge, potentially misleading conventional matching processes and producing local maxima results, as shown in Figure 4.

Table 1: Obfuscation techniques employed by commercial Android obfuscators [24–26].

Obfuscation Type	ProGuard	Allatori	DashO
Identifier Renaming	✓	✓	✓
Repackaging	✓	✓	✓
Package Flattening	✓	-	✓
Dead Code Removal	✓	✓	✓
Code Addition	✓	✓	✓
String Encryption	-	✓	✓
Control-Flow Randomization	-	✓	✓

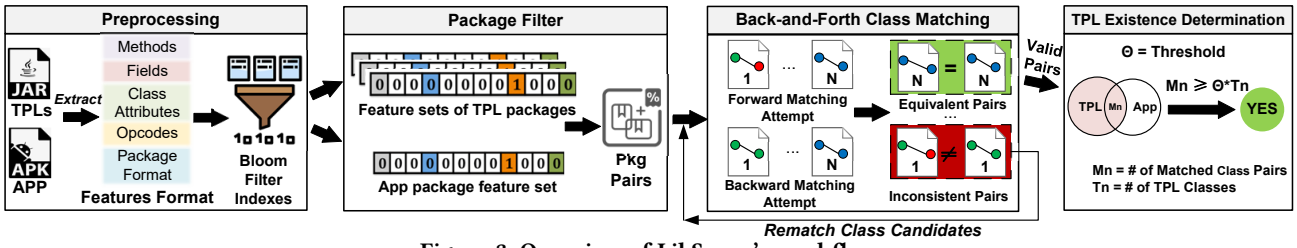

Figure 3: Overview of LibScope's workflow.

2.4 Scope of LibScope

The evolution of Android obfuscation techniques [50], coupled with the widespread adoption of commercial obfuscators [24–26], has driven the progression of TPL detection methods. They have transitioned from initial versions that focused on matching identifier names [51–53] to recent advancements capable of withstanding common obfuscation tools [21, 23]. We recognize that the arms race in code obfuscation and deobfuscation has transformed into an intensive tug-of-war; it is evident that no singular TPL detection tool can comprehensively address all Android obfuscation techniques. Therefore, the anti-obfuscation scope of LibScope remains consistent with recent works such as LibScan [23].

Table 1 lists specific obfuscation methods supported by three prominent obfuscators [24–26], and LibScope aims to effectively handle all of them. Leveraging fuzzy features, LibScope demonstrates robustness against identifier renaming. Moreover, our package filtering strategy can competently accommodate package flattening and repackaging obfuscation. Similar to LibScan [23], our approach mitigates the impact of dead code removal and code addition by implementing similarity thresholds. Notably, LibScope's features do not rely on control flow graphs (CFGs), making it resistant to obfuscation techniques and compiler optimizations that modify CFGs. Furthermore, LibScope's detection mechanism remains unaffected by obfuscation schemes targeting string constants and non-code-based obfuscation such as manifest transformations.

However, given that our approach is based on static analysis, we acknowledge that dynamic obfuscation techniques enacted at runtime, such as class encryption [54], app-level Dex encryption [30], and virtualization-based protection [55], fall outside the purview of our consideration. This assumption aligns with the constraints acknowledged by other similarity-based TPL detection tools.

3 Overview

In this section, we first introduce the workflow of LibScope. Then, we present an example to showcase the efficacy of the back-and-forth gaming methodology in achieving global matches.

System Workflow Figure 3 provides an overview of LibScope's workflow. LibScope first takes the app and TPLs as inputs to generate fuzzy feature profiles for both. Through comparison of these profiles, LibScope produces a report indicating the presence of TPLs within the app. Our approach involves refining existing fuzzy features, enhancing methods, fields, class attributes, method call relationships and opcodes to construct class candidates. Subsequently, package candidates are formed by aggregating class candidates within the same package. Then, a comparison process is initiated to narrow down possible matched package pairs based on package features. Following this, we apply the back-and-forth gaming algorithm to determine matched class pairs for these packages, which is a crucial step in ensuring the accuracy of TPL detection. At last, we determine the presence of the TPL in the app when the number of the matched class pairs exceeds a predefined threshold.

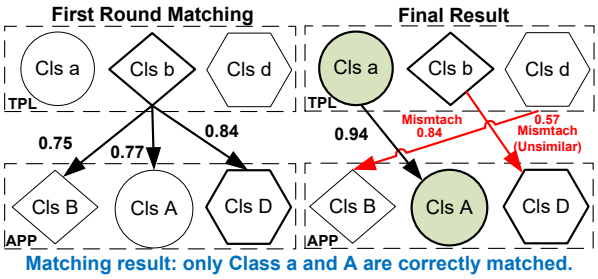

Figure 4: Local maxima of similarity matches.[1]

Localized Matching Result As a critical step in TPL detection, measuring class fuzzy features may produce multiple similar matching candidates. Simply selecting the pair of candidates with the highest similarity score as the best-matched class pair is a prevalent limitation observed in the majority of existing approaches. The running example depicted in Figure 4 illustrates three mutually paired classes between TPL and the app. The traditional locally optimal solution begins with searching for a best-matched pair for class b. Unfortunately, due to the influence of obfuscation and normalization of fuzzy features, TPL class b loses its maximum similarity with its intended app counterpart, class B. Consequently, class b and class D, which display the highest similarity score, form an erroneously matched pair. Even though class a and class A meet the maximum similarity condition and are successfully matched, the remaining class d and class B fail to match with each other owing to their low similarity score (0.57). At last, only 1/3 of the class candidates can be correctly matched. Please note that once two candidates cannot be correctly paired, the cascading effect of mismatch will further compromise the overall matching effectiveness.

Key Insight The localized matching, as depicted in Figure 4, often fail to identify the global solution, as they ignore neighboring classes during matching, therefore, they commit to matching candidates too early, which restricts their ability to explore better overall solutions later. Our key insight highlights the significance of not only considering individual class-pair similarities but also embracing a comprehensive assessment of neighboring classes within a package scope to achieve superior matching results. This insight motivates our adoption of the principles underlying model theory's back-and-forth game [31] to navigate towards the global matches.

LibScope Matching Result To optimize matching outcomes, our back-and-forth gaming involves a multi-round matching process to

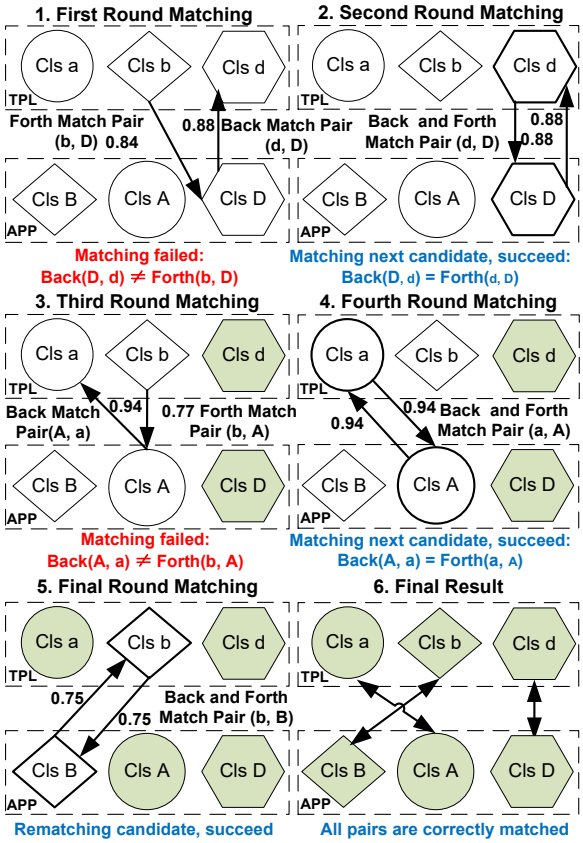

Figure 5: Global matches via back-and-forth gaming.

explore surrounding classes, each round consists of both forward and backward matching attempts. Figure 5 illustrates how our methodology rectifies Figure 4's mismatch.

Similar to Figure 4, our approach initiates by looking for a matched pair for the TPL class b. In the first round, the forward matching attempt compares the TPL class b with all app candidates, and then it selects the app class with the highest similarity score to form a temporarily matched pair (class b *vs.* class D). Please note that this result is not immediately regarded as the final outcome. In this round, we also need to execute a backward matching attempt to verify the result obtained from the forward matching. Specifically, we compare the temporarily-selected target (class D) with all TPL candidates. However, a different matched pair (class D *vs.* class d) emerges based on the ranking of similarity scores. If both forward and backward matching attempts yield an equivalent matched pair, we consider the result as a valid match. In cases of any inconsistency, the mismatched pairs are discarded, and the process advances to the next round. In this example, our algorithm proceeds with a different TPL candidate (class d) for the second round matching. Consequently, a correct pair (class d & class D) is identified through mutual agreement from two-direction matching attempts. This iterative process continues; although the third round fails, the fourth round successfully finds a correct pair (class a & class A). Eventually, in the final round, class b is matched to its correct counterpart. This iterative approach ensures the successful matching of all TPL classes to their respective app counterparts.

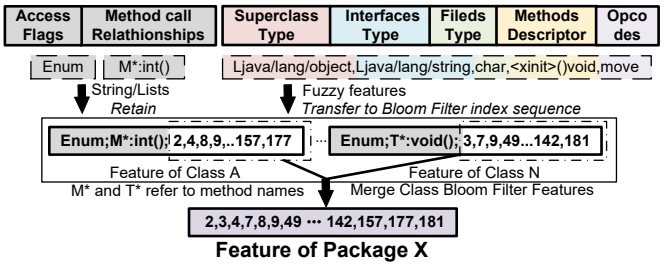

Figure 6: LibScope's package feature format.

The comparison between Figure 4 and Figure 5 visually demonstrates the novelty of our approach. By integrating contextual information from surrounding classes, LibScope addresses the mismatch inherent in the localized matching of fuzzy features, leading to significantly improved outcomes.

4 LibScope Design

This section begins with our feature generation. Then, we present the package filter mechanism that eliminates irrelevant TPLs and creates valid package pairs. Further, the discussion focuses on the class matching process, leveraging the back-and-forth algorithm to produce a global outcome for class matches.

4.1 Preprocessing: Fuzzy Feature Generation

In the context of obfuscation, certain TPL code features remain unchanged, while others undergo regular modifications. The essence of feature extraction lies in identifying invariant features and uncovering transformation patterns in variable features. Several studies have proposed rules for different features. Therefore, our focus is on refining and combining existing features to enhance the representation of comparison units. After evaluating various feature combinations in terms of effectiveness and efficiency, LibScope adopts the class as the fundamental unit for feature comparison and introduces features from the following categories: *methods*, *fields*, *class attributes*, *opcodes*, and *method call relationships*.

Method Features The functionality of a class is intrinsically linked to its methods. Therefore, we extract method features by abstracting parameters and return values of methods. Since obfuscation does not alter code semantics, we found that method call relationships typically persist after common Android obfuscation. This discovery is leveraged to refine our features, enhancing the accuracy of fuzzy descriptors without compromising obfuscation resilience. To this end, we record all caller and callee relationships using fuzzy method signatures for each method in a dictionary. At the beginning of the matching process, LibScope verifies if an app method shares both the same method signature and call relationships with a unique method from the TPL. Upon finding such a match, LibScope establishes a correspondence between the parameter and return types of the app and TPL methods (e.g., <a, Gson >), aiming to reveal the original representation of the obfuscated identifier. This approach enables the conversion of all features involving the obfuscated user-defined type *a* into specific representations, such as *Gson*, instead of the placeholder "X", thereby improving the feature precision.

Field Features Fields define the properties and data structures of class, providing insights into its characteristics. We extract all

fuzzy descriptors associated with field types, including symbols that indicate access attributes (e.g., static).

Class Attributes Certain class access flags, such as *abstract* and *static*, often remain unchanged after obfuscation [20], providing a clear criterion for determining whether two class candidates can be considered a potential match. Besides, capturing superclass type and class interfaces type, representing class inheritance information, effectively delineates the hierarchical relationships among classes.

Opcodes Opcodes reflect the code's structure, logical flow, and functional characteristics. Therefore, we collect all opcodes from each method in the class as part of the feature.

Format of Package Feature The package feature construction process is depicted in Figure 6. Initially, we combine various feature components of the class to create candidate features. Then, package features are generated based on the features of class candidates within the package. To realize this, we retain the original string of access flags and a dictionary of method call relationships as part of class feature. The former enables quick filtering of unrelated class candidates during comparison, while the latter can enhance our feature precision. The other feature components, including method descriptors, field types, superclass types, interface types and opcodes, are abstracted into *Bloom Filter* sequence indexes for later comparison. Finally, we merge *Bloom Filter* features of all class candidates within the same package to form package feature.

When attempting to represent a diverse range of candidates, using a more extensive set of features can be advantageous. However, finer features are more susceptible to strong obfuscation (e.g., control-flow randomization can significantly alter CFG features). More importantly, even with highly complex feature compositions, fuzzy features may inadvertently lead to the generation of more similar candidates, despite belonging to different classes. This can lead to potential mismatches unless additional constraints are imposed. So, in §4.2 and §4.3, we utilize package filter and surrounding class information to progressively identify the matched class pairs.

4.2 Package Filter

The package filter serves a dual purpose: 1) it excludes irrelevant package candidates and 2) narrows down the focus for the subsequent comparison of class candidates through the computation of similarity between the app package and TPL package to establish a potential package pair. We first measures the similarity between the app package (ap) and TPL package (tp) by calculating the overlap ratio of their bloom filter feature sets using Equation 1:

$$package\_sim1(ap, tp) = \frac{|BF(ap) \wedge BF(tp)|}{MIN|BF(tp), BF(ap)|} \tag{1}$$

Considering a scenario where packages ap and tp have bloom filter sequences 1 and 1, 2, 3, …, respectively, Equation 1 yields an overlap ratio of 100%. However, this metric fails to reflect the actual distinction between the two packages. To tackle this issue, LibScope utilizes Equation 2 in conjunction with the former equation, ensuring that the features of one package have sufficient common characteristics with those of its counterpart. In addition, we employ the entropy-based rules [21] to counteract repackaging and flattening obfuscation.

$$package\_sim2(ap, tp) = \frac{|BF(ap) \wedge BF(tp)|}{MAX|BF(tp), BF(ap)|} \tag{2}$$

The package pair (ap *vs.* tp) is designated for subsequent back-and-forth comparison only when the aforementioned two equations surpass their respective thresholds (θ_1 for Equation 1 and θ_2 for Equation 2) individually. To determine the optimal values, we performed a threshold tuning experiment, as described in Section 5.

4.3 Back-and-Forth Class Matching

After acquiring package match pairs, class pairing within these pairs is crucial for confirming the existence of TPLs. However, classes within the same package may appear similar due to shared inheritance or other factors. Moreover, the use of fuzzy features heightens the probability of different classes exhibiting apparent similarities, potentially misleading locally optimal matching solutions.

Principle of Back-and-Forth Gaming Our approach draws inspiration from model theory's back-and-forth game, akin to a player-opponent game [56] where the player proposes a theorem, and the opponent seeks a single counterexample to prompt the player to refine the theory. The ultimate goal of the game is for the player to find perfect strategy that the opponent cannot refute and thus win the game. The efficacy of the back-and-forth strategy stems from its ability to reevaluate the initially identified best match, allowing for optimization if a superior result emerges later in this round. We strive to elevate local matching results to a global matches by leveraging contextual information present in the surrounding classes and refine matching outputs through back-and-forth matching iterations. The neighboring classes refer to the unmatched classes excluding the currently targeted matching class. Contextual information is derived from other potential locally optimal results formed by these neighboring classes, which can broadens the scope of target class similarity from local to global, guiding us in forming global matches or discarding results for each matching iteration.

Formalization Next, we proceed to formalize the intuition to explain why our algorithm can achieve these results. In our context, *BaF* represents the back-and-forth comparison process, where A and T denote all app and TPL class candidates. The similarity of the pair (a, t) is abbreviated as $\mathrm{Sim}(a, t)$. The ultimate goal of the game in each round is to obtain a globally matched pair (a, t), where $a \in A$ and $t \in T$. Assuming the matching starts with t to find the best match a within A as usual, in our approach, we acknowledge that this best match represents a local outcome within all possible pairs (A, t) but cannot be considered as a global result directly. This flexibility allows for a better match if a has a different and superior match with candidates in set T. The global match pair is determined only when (a, t) obtains the maximum similarity among all possible pairs within (A, T). This process is formalized in Formula 3.

$$\begin{aligned} &\underset{A,T}{BaF}(a, t) \quad \leftrightarrow \\ &(\forall a_i \in A, a_i \neq a, Sim(a_i, t) \leq Sim(a, t)) \wedge \\ &(\nexists t_j \in T, t_j \neq t, Sim(a, t_j) > Sim(a, t)) \end{aligned} \tag{3}$$

In the first set of parentheses, a local solution, denoted as a for t, is selected from all app candidates based on the ranking of their fuzzy feature similarities. Within the second set of parentheses, an additional comparison is conducted within the context of a to

all candidates in T to identify a better match pair than (a, t). If no superior match pair exists, (a, t) stands as the best-matched pair globally within (A, T). Otherwise, it signifies that (a, t) represents only a local result, and the match relationship between these two candidates is disrupted, necessitating further matching attempts. By implementing this algorithm, LibScope aims to achieve a global result by considering the holistic context of all matching candidates, thereby ensuring more precise matches.

Algorithm 1 Back-and-forth Matching Algorithm

Input: A – *all app class in matched package*
T – *all TPL class in matched package*
Output: *MatchedPairs, Abbreviated as P*

```
1:  P = {}
2:  RefineFeature(A, T)
3:  WrongTargets = {}, Abbreviated as WT
4:  while MatchingNotEnd(A, T, P, WT) do
5:    t_j ← SelectTarget(T, WT, P)
6:    (a_i, t_j) ← GetLocalOptimal(A, t_j, P)
7:    (a_i, t_j') ← GetLocalOptimal(T, a_i, P)
8:    if t_j == t_j' then
9:      P.add(a_i, t_j)
10:     InitList(WT)
11:   else
12:     WT.add(t_j)
13:   end if
14: end while
15: Function GetLocalOptimal(candidates, target, P):
16:   ToMatch = {}, Abbreviated as TM
17:   ValidOnes ← Filter(candidates, target, P)
18:   TM.add(ValidOnes)
19:   BestCandidate ← GetBestMatch(TM, target)
20:   Return (BestCandidate, target)
```

Back-and-Forth Matching Algorithm The detailed back-and-forth class matching strategy is listed in Algorithm 1. The algorithm takes class candidates from matched package pairs as input. During the matching process, each candidate obtains a localized result by matching within two distinct scopes (i.e., all app and TPL candidates). However, until the algorithm determines this result to be the global one, it will continue to iterate and update the matching strategy. We initialize an empty tuple labeled as *MatchedPairs* (Line 1) to store all global matches (Line 9). This iterative process continues until all candidates from either the app or the TPL side have been successfully matched, or no more global results can be generated.

At the beginning of the first round of iteration, the function *RefineFeature* (Line 2) is employed to identify app and TPL methods that have the same method signature and share unique call relationships, allowing LibScope to recover fuzzy descriptors into their original specific representations, thereby increasing feature precision. Additionally, an empty list, termed *WrongTargets*, is initialized (Line 3) to accumulate the targets that cannot form pairs in this round. These wrong targets serve as inputs to the *SelectTarget* function. As the algorithm progresses, this list continues to collect erroneous targets (Line 12) until a global matches is formed. The *WrongTargets* list is reset to accommodate new erroneous information (Line 10). Lines 5 to 13 delineate a singular round of the back-and-forth class matching process, in which the function *SelectTarget* decides a target, then *GetLocalOptimal* provides the locally matched class pair for this target. This function will execute twice to obtain results from different matching scopes (A and T).

Lines 15 to 20 show the main process of function *GetLocalOptimal*. As target (such as t_j in Line 5) in this function needs to be compared with nearly all candidates within a certain matching scope (A or T), to increases the efficiency of obtaining global results and eliminates irrelevant candidates, we employ the following strategy to filter candidates and identify the valid ones. Firstly, candidates with same class name as input target will be considered first, as they are highly likely to belong to the identical class. Secondly, certain incorrect candidates will be directly eliminated based on class flags. Finally, matched candidates will not be considered again. All these valid candidates will be added to a set named *ToMatch*. In Line 19, the function *GetBestMatch* is used to find *BestCandidate* in *ToMatch* that is most similar to the target, and their similarity score also exceeds the minimum threshold θ_a. The pairwise class similarity calculation is expressed in the following Equation 4.

$$class\_sim(ac, tc) = \frac{|BF(ac) \wedge BF(tc)|}{|BF(ac) \vee BF(tc)|} \tag{4}$$

4.4 TPL Detection

After obtaining all valid matched class pairs through back-and-forth comparisons, we assess the probability of the TPL's presence within the app. This evaluation is formalized in Equation 5, where the presence similarity of the TPL is defined as the ratio of valid matched class pairs to the maximum possible matches, with the latter representing the total number of classes within the TPL.

$$existence\_sim(app, tpl) = \frac{|Valid\ Class\ Pairs|}{|Classes\ in\ TPL|} \tag{5}$$

Similar to the previous comparison process, if the outcome from this formula surpasses the threshold θ_b, we determine the presence of the TPL within the app. For details on the empirical study tuning of θ_a and θ_b, please refer to the Section 5.

5 Evaluation

5.1 Experiment Settings

Implementation and Testbed We employed Androguard [57] as the primary framework to extract fuzzy features and utilized dex2jar toolset [58] to convert jar files into the dex format. Our comparative experiments involved testing LibScope against four prominent tools: LibScout [19], LibID [32], LibLoom [21], and LibScan [23]. These tools were selected for their representation of state-of-the-art approaches in similarity-based TPL detection and for their public availability. Other tools, such as ATVHunter [17], were excluded due to lack of public access. All our experiments were executed on a server equipped with an Intel Xeon Gold 6338 CPU and 256GB of memory, operating on Ubuntu 20.04.

Dataset Collection Our controlled experiments necessitate datasets with established ground truth to accurately assess TPL detection outcomes. To fulfill this requirement, we utilized obfuscated datasets provided by LibLoom (*Dataset1*) and LibScan (*Dataset2*) as our

Table 2: Statistics of TPL detection benchmarks. (IR = identifier renaming, DCR = dead code removal, F = package flattening, R = repackaging, CFGR = control flow graph randomization, SE = string encryption.)

Dataset	Obfuscator	Obfuscation Type	Subset	# of Apps
LibLoom (Dataset1)	Dasho	IR DCR F CFGR SE	DOF	100
		IR DCR R CFGR SE	DOR	100
	Proguard	IR DCR	PGD	100
		IR DCR F	PGF	100
		IR DCR R	PGR	100
	Allatori	IR DCR CFGR R	ALR	200
LibScan (Dataset2)	Dasho	IR R	DOR2	159
		CFGR	DOC	79
		IR F	DOF2	79
		DCR	DOD	79
	Proguard	IR DCR	PGD2	152
	Allatori	IR DCR R	ALR2	188

benchmarks. The selection of these two datasets was deliberate for the following reasons. First, LibLoom and LibScan have invested substantial efforts in dataset construction, making them the most comprehensive public benchmarks for TPL detection. Second, given LibScope's resilience against commercial obfuscators [24–26], its capabilities closely align with the obfuscation types prevalent in these datasets. Finally, these datasets are publicly available online, unlike tools such as LibID, whose datasets remain unpublished.

Obfuscation Types Most of apps within these two datasets were obfuscated by three prominent popular obfuscators: Proguard [24], Allatori [25], and Dasho [26], each configured with distinct obfuscation settings. In *Dataset1*, the 700 APK files were categorized based on various package obfuscation techniques, including repackaging and package flattening. On the other hand, *Dataset2* comprises a total of 736 obfuscated apps explicitly identified in the LibScan repository. The obfuscation details for these two datasets are presented in Table 2. In reviewing the "Obfuscation Type" column, it becomes apparent that apps within *Dataset1* demonstrate a wider range of obfuscation techniques compared to those in *Dataset2*. Therefore, *Dataset1* stands as a benchmark showcasing strong obfuscation. Each abbreviation within the "Subset" column denotes a distinct subgroup of apps sharing identical obfuscation techniques. In the later evaluation, we will test each subset separately.

Large-Scale Test Cases We collected a total of 1,436 obfuscated apps with ground truth to measure the effectiveness of LibScope and peer tools. This number notably exceeds the scale of controlled experiments conducted in prior research, such as 700 apps for LibLoom and 1,046 for LibID. Besides, to assess LibScope's efficacy in vulnerability identification and its real-world scalability, a random sample of 5,000 apps per year from AndroZoo [59, 60] was selected from 2019 to 2024, totaling 30,000 apps as our *large-scale dataset*.

Thresholds Tuning As thresholds influence the detection capabilities of LibScope, we empirically tuned these parameters using a set of 73 F-Droid apps [61], distinct from those in Dataset1 and Dataset2. We first tuned θ_1 and θ_2 by varying their values from 0 to 1 in increments of 0.05, selecting the threshold combination that maximized the number of correct package match pairs across all 73 test apps, which yielded $\theta_1 = 0.8$ and $\theta_2 = 0.35$. We then applied the same procedure to optimize the TPL detection F1 score, resulting in $\theta_a = 0.75$ and $\theta_b = 0.6$. The subsequent experiments were conducted using these tuned values.

5.2 Effectiveness Evaluation

We evaluate the effectiveness of LibScope in both library detection and version detection. The former assesses the tool's ability to detect the presence of TPLs in the app, while the latter focuses on the tool's accuracy in identifying specific versions of the TPL once its category is correctly recognized. For instance, an accurate library detection entails identifying a library as "glide-4.12.0," while a discrepancy in detecting the precise version (e.g., detecting "glide-4.11.0") would indicate an accuracy issue in version detection.

In our comparative analysis, we evaluate both the library and version detection capabilities of LibScope in relation to its four counterparts: LibScout [19], LibID [32], LibLoom [21], and LibScan [23]. TPL detection results for each tool concerning the existence of 5,024 TPL profiles are presented in Table 3 for *Dataset1*, Table 4 for *Dataset2*, and Table 6 for *large-scale dataset*. Performance metrics within each subset (as per Table 2's subsets) are represented by *LP* (precision), *LR* (recall), and *LF1* (F1 score) for library detection, and *VP*, *VR*, *VF1* for these metrics in version detection.

Peer Tools' Performance on Dataset1 As an early representative tools, LibScout demonstrates high precision in analyzing Proguard-obfuscated apps, as evidenced in the "PGR" subset. However, it shows relatively lower recall in detecting all obfuscated apps. This disparity could be attributed to inconsistent changes in package structure caused by different obfuscators, which LibScout relies on to generate its features. In comparison, LibID achieves higher recall rates than LibScout but does not maintain precision beyond 35%. In its utilization of the Binary Integer Programming (BIP) algorithm, LibID operates under the assumption that obfuscators will not disrupt internal package hierarchy structures, a presumption that holds true only in specific instances. However, even when the package hierarchy remains intact, the outcomes generated by the BIP algorithm often encompass a multitude of erroneous matches, given its objective of maximizing matched pairs. These inherent flaws have significantly compromised its overall detection efficacy. LibScan employs method-opcode and call-chain-opcode features in setting a similarity threshold. However, its rigorous detection process falters in strongly obfuscated apps, impacting its overall performance. LibLoom, leveraging fuzzy features and translating TPL detection into a set inclusion problem, outperforms LibScout, LibID, and LibScan. Nevertheless, LibLoom also faces limitations, evident in its recall falling below 46% when confronted with Dasho and Proguard. This highlights the limited efficacy of localized matching in producing accurate global matches.

LibScope In contrast, LibScope demonstrates remarkable performance in both TPL detection and version identification, significantly outperforming its counterparts. In terms of library detection, LibScope achieves exceptional recall, surpassing LibScout and LibScan by over tenfold, and even outperforming the leading tool in related work, LibLoom, with a recall score of 69.5%. Moreover, its precision rate of 88.4% surpasses that of other similar tools. Despite the inherent challenge of identifying closely related versions of TPL

Table 3: TPL detection results using *Dataset1*. Scope^{I} = LibScope's features + localized matching, Scope^{II} = LibScope's features + Binary Integer Programming (BIP) algorithm.

		Scope^{I}	Scope^{II}	LibScope	LibScout	LibScan*	LibID	LibLoom
ALR	LP	84.0%	73.7%	**85.1%**	51.5%	40.8%	13.6%	82.3%
	LR	88.2%	82.0%	**96.1%**	4.7%	11.6%	61.2%	86.0%
	LF1	86.1%	77.6%	**90.3%**	8.7%	18.1%	22.2%	84.1%
	VP	76.9%	53.4%	**78.6%**	39.5%	28.1%	10.8%	66.0%
	VR	80.8%	59.4%	**88.7%**	3.6%	8.0%	48.6%	68.8%
	VF1	78.8%	56.2%	**83.3%**	6.6%	12.4%	17.6%	67.3%
DOF	LP	90.4%	78.5%	**92.4%**	36.4%	31.4%	13.7%	85.6%
	LR	44.3%	38.8%	**57.4%**	0.4%	3.5%	12.7%	45.6%
	LF1	59.4%	51.9%	**70.8%**	0.9%	6.3%	4.6%	59.5%
	VP	77.7%	59.7%	**82.4%**	36.4%	27.5%	10.4%	69.4%
	VR	38.0%	29.5%	**51.1%**	0.4%	3.1%	2.1%	36.9%
	VF1	51.1%	39.5%	**63.1%**	0.9%	5.5%	3.5%	48.2%
DOR	LP	84.8%	74.6%	**89.8%**	41.7%	30.2%	12.3%	82.3%
	LR	34.9%	41.3%	**55.1%**	0.5%	3.5%	3.1%	36.1%
	LF1	49.4%	53.2%	**68.2%**	1.1%	6.3%	4.9%	50.1%
	VP	71.3%	57.2%	**80.8%**	41.7%	26.4%	9.7%	63.8%
	VR	29.3%	31.7%	**49.5%**	0.5%	3.1%	2.4%	28.0%
	VF1	41.5%	40.8%	**61.4%**	1.1%	5.5%	3.9%	41.1%
PGD	LP	91.7%	87.4%	**93.2%**	82.2%	26.8%	33.2%	90.2%
	LR	51.6%	49.2%	**64.1%**	15.6%	2.8%	48.6%	43.3%
	LF1	66.0%	62.9%	**76.0%**	26.2%	5.1%	39.5%	58.5%
	VP	77.9%	67.7%	**81.9%**	74.6%	15.5%	26.3%	79.3%
	VR	43.8%	38.1%	**56.4%**	14.1%	1.6%	38.5%	38.0%
	VF1	56.1%	48.8%	**66.8%**	23.8%	3.0%	31.2%	51.4%
PGF	LP	88.7%	83.9%	**91.0%**	85.2%	26.6%	27.7%	85.4%
	LR	46.4%	38.2%	**59.6%**	11.2%	2.7%	36.5%	45.6%
	LF1	61.0%	52.5%	**72.0%**	19.8%	4.9%	31.5%	59.4%
	VP	71.2%	68.8%	**82.0%**	79.1%	16.0%	21.8%	66.6%
	VR	37.3%	31.4%	**53.7%**	10.4%	1.6%	28.6%	35.5%
	VF1	48.9%	43.1%	**64.9%**	18.4%	3.0%	24.7%	46.3%
PGR	LP	80.8%	83.2%	86.6%	**88.7%**	27.4%	13.7%	85.8%
	LR	37.8%	43.4%	**58.0%**	10.3%	2.8%	42.2%	36.9%
	LF1	51.5%	57.0%	**69.5%**	18.4%	5.1%	20.7%	51.6%
	VP	67.1%	68.1%	**77.0%**	76.4%	17.9%	10.3%	65.2%
	VR	31.4%	35.5%	**51.6%**	8.9%	1.9%	31.7%	28.1%
	VF1	42.7%	46.7%	**61.8%**	15.9%	3.4%	15.5%	39.3%
Total	LP	85.9%	77.9%	**88.4%**	73.4%	34.8%	27.2%	84.3%
	LR	55.9%	53.6%	**69.5%**	6.7%	5.5%	36.5%	54.2%
	LF1	67.7%	63.5%	**77.8%**	12.3%	9.5%	22.9%	65.9%
	VP	69.8%	59.2%	**79.8%**	64.3%	24.6%	13.1%	67.3%
	VR	48.7%	40.7%	**62.8%**	5.9%	3.9%	28.6%	43.3%
	VF1	57.3%	48.3%	**70.3%**	10.8%	6.7%	18.0%	52.7%

where subtle differences prevail among versions, particularly in obfuscated scenarios, LibScope exhibits commendable performance. Its precision rate of 79.8% and a recall score of 62.8% culminate in the highest F1 score of 70.3%. This performance significantly outstrips the corresponding values achieved by other tools, which are notably lower at 10.8%, 6.7%, 18.0%, and 52.7%, respectively.

LibScope's exceptional performance can be attributed to two key factors that complement each other. Firstly, LibScope's refined features enhance the distinctiveness of potential matches, increasing their diversity and thereby reducing the likelihood of erroneous matches. Secondly, owing to the use of fuzzy features in obfuscated apps, similar candidates tend to appear frequently. However, LibScope's back-and-forth comparison method adeptly addresses these similarities by leveraging contextual information from surrounding candidates. This strategy significantly reduces the occurrence of incorrect matches associated with these fuzzy features.

Stepwise Contribution Analysis LibScope refines fuzzy features and employs the back-and-forth gaming algorithm to enhance the matching process. To delineate the specific contributions of these steps, we conducted a separate experiment using refined features with the traditional localized matching algorithm, denoted as Scope^{I} in Table 3. Given that LibScope's fuzzy features draw from related works, we selected the top-performing tool from Table 3, namely LibLoom, as the baseline for comparison. The comparison between LibLoom and Scope^{I} allows us to gauge the advancements achieved through LibScope's refined fuzzy features. Simultaneously, the comparison between Scope^{I} and LibScope elucidates the novelty and efficacy of the proposed back-and-forth gaming algorithm. As LibID utilizes the Binary Integer Programming (BIP) algorithm to find the match pairs, to visually compare the differences between the BIP algorithm and the back-and-forth approach in terms of performance, we recorded the results using LibScope's features plus LibID's BIP algorithm as Scope^{II}. By comparing Scope^{II} with LibScope's result, it clearly shows the extent of improvement caused by the back-and-forth algorithm rather than features.

In summary, Scope^{I} demonstrates noteworthy advancements in library detection, specifically in the F1 score for library detection (*LF1*) that improved from 65.9% to 67.7%, and F1 score for version detection (*VF1*) that increased from 52.7% to 57.3% over LibLoom for total apps. These results indicate that the refined features have stronger discriminative capabilities, making LibScope better able to distinguish smaller version changes. Building upon the improvements achieved by Scope^{I}, LibScope attains an additional increase from 67.7% to 77.8% in *LF1* and 57.3% to 70.3% in *VF1*. This enhanced performance underscores the advantage of the back-and-forth algorithm in finding the correct match pairs through contextual information, thereby more accurately reflecting the true similarity between apps and TPLs.

Compared to Scope^{II}, LibScope improved the F1 score for TPL detection from 63.5% to 77.8% and increased the value for version identification from 48.3% to 70.3%, demonstrating that back-and-forth algorithm is more suitable for TPL detection compared to the BIP algorithm. The subpar performance of the BIP algorithm can be attributed to several factors. Firstly, the algorithm relies on heuristic constraints, yet identifying a set of constraints that are simultaneously effective, complementary, non-conflicting, and universally applicable across various scenarios, especially when confronted with different obfuscation technologies and their combinations, poses a significant challenge. Secondly, the algorithm's principle of maximizing matched pairs does not inherently translate to an increase in the number of correct matches. In practice, the algorithm employed by LibID may fail to adequately consider higher similarity between candidates as a crucial criterion for accurate matching. This oversight often results in the generation of erroneous matches, thereby undermining its effectiveness.

Performance Comparison with Dataset2 Table 4 exhibits the performance metrics of various tools on *Dataset2*. Given the relatively simpler obfuscation combinations present in *Dataset2*, all tools exhibit a noteworthy improvement in overall results compared to those presented in Table 3. LibID's performance varies significantly when encountering different obfuscation techniques, while LibScout still encounters challenges in effectively handling obfuscation applied by Allatori or Dasho. Notably, the dataset provided by LibScan positively impacts its performance, evident in **nine** times higher *LF1* and *VF1* scores compared to those observed in *Dataset1*.

Table 4: TPL detection results using *Dataset2*.

		LibScope	LibScout	LibScan	LibID	LibLoom
ALR2	LF1	**95.7%**	16.6%	90.1%	40.3%	94.0%
	VF1	**95.0%**	16.4%	87.4%	38.9%	79.0%
DOC	LF1	**98.5%**	7.8%	96.1%	1.3%	97.0%
	VF1	**98.5%**	5.5%	96.2%	0.6%	95.8%
DOF2	LF1	**91.2%**	6.3%	83.4%	0.6%	80.4%
	VF1	**89.4%**	4.8%	82.3%	0.6%	76.5%
DOR2	LF1	**94.5%**	30.0%	89.9%	30.3%	86.0%
	VF1	**93.0%**	29.0%	89.1%	29.8%	84.2%
DOD	LF1	**97.0%**	7.9%	82.8%	1.9%	93.0%
	VF1	**95.7%**	7.1%	82.8%	1.9%	89.9%
PGD2	LF1	**97.5%**	86.3%	95.4%	80.0%	96.1%
	VF1	**97.3%**	92.1%	94.3%	79.2%	94.6%
Total	LF1	**95.6%**	39.4%	90.4%	41.4%	91.4%
	VF1	**94.8%**	37.0%	89.0%	40.6%	85.1%

LibLoom's performance data on *Dataset2* are comparable to those of LibScan. Moreover, LibLoom showcases high consistency across the two datasets, demonstrating its robustness. Once again, LibScope emerges as the top-performing tool, achieving an *LF1* of 95.6% and *VF1* of 94.8% on *Dataset2*. Among other tools, the maximum *LF1* and *VF1* values stand at 91.4% and 89.0%, respectively. These results underscore the efficacy of the back-and-forth algorithm in significantly improving the accuracy of candidate matches, even when confronted with similar fuzzy features. This enhancement notably benefits both TPL detection and version identification.

5.3 Efficiency Evaluation

Our efficiency evaluation focuses on comparing the runtime performance of LibScope against peer tools. We report the average detection time for testing apps across the two datasets. and the results are presented in Table 5.

Based on the experimental results, LibScout demonstrates the highest detection efficiency. LibScout leverages the package hierarchy to construct a *Merkle Tree* structure. Its comparison process terminates early if substantial differences emerge between the original Merkle Tree and the new Merkle Tree after package obfuscation. In contrast, both LibLoom and LibScan adopt a two-stage detection method to filter unrelated candidates. However, the additional call-chain similarity comparison in LibScan introduces a time-consuming process, thereby increasing its overall overhead.

The BIP algorithm employed by LibID constitutes a classic NP-complete problem with high complexity, resulting in exponential growth in time costs, particularly when processing a large number of TPL feature profiles. As a consequence, both LibID and Scope^{II} exhibit significantly longer processing times compared to other TPL detection tools, underscoring the inefficiency of the BIP algorithm for library detection.

While our approach employs a package filter to screen TPL candidates, the comparison of richer fuzzy features and the application of the back-and-forth gaming algorithm both incur additional runtime overhead. As shown in Table 5, the average detection time for Scope^{I} is 33.1s, demonstrating an efficiency level comparable to that

Table 5: Average TPL detection time for each tool

Tool	Scope^{I}	Scope^{II}	LibScope	LibScout	LibScan	LibID	LibLoom
Avg.	33.1s	1648.3s	48.0s	10.2s	135.7s	1711.8s	32.6s

of LibLoom. The average detection time for LibScope is 48.0s, outperforming LibScan and LibID. LibScope requires multiple rounds of comparisons to obtain global results, so the disparity values between "LibScope" and "Scope^{I}" columns represent the running time of our proposed algorithm. Given its excellent effectiveness, LibScope's overhead is acceptable for real-world use.

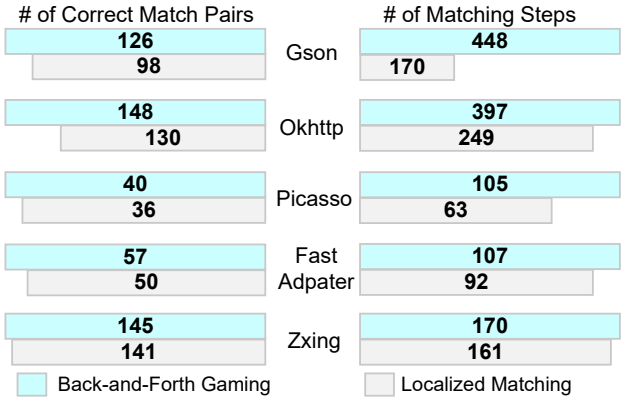

Figure 7: The five most prevalent TPLs detected solely by LibScope but missed by localized matching algorithms.

5.4 Back-and-Forth Matching Algorithm

To comprehensively assess the impact of our class matching methodology on TPL detection, we gathered data regarding the five most prevalent TPLs exclusively identified by the back-and-forth algorithm, but evading detection through localized matching. Instances of these TPLs are outlined in Figure 7. This figure compares the quantitative differences between our algorithm and the localized matching in terms of correct match pairs and matching steps.

Since the localized matching relies solely on the ranking of similarity scores as the criterion for comparison, we treat this process as a single matching step. Conversely, our back-and-forth strategy incorporates not only similarity scores but also the contextual information of matching candidates. As a result, our method requires additional matching steps to enhance the number of correctly matched pairs and improve the overall accuracy. For instance, in the case of TPL *gson*, our method required an additional 278 matching steps, resulting in a 28.6% increase in the matched pairs compared to the localized matching. Moreover, even a marginal increment in matched pairs can significantly influence the final detection outcome. In the case of identifying TPLs such as *zxing* and *picasso*, despite a mere 4 additional matched pairs compared to the localized matching, our method correctly identified these TPLs, whereas localized matching failed to do so.

The above findings piqued our interest in understanding the impact of varying matching steps on TPL detection. To analyze the number of matching steps necessary to establish a valid matched pair, we plotted the top 5 most frequently encountered TPLs along with the distribution of their back-and-forth matching steps. In Figure 8, the horizontal axis depicts the number of steps required

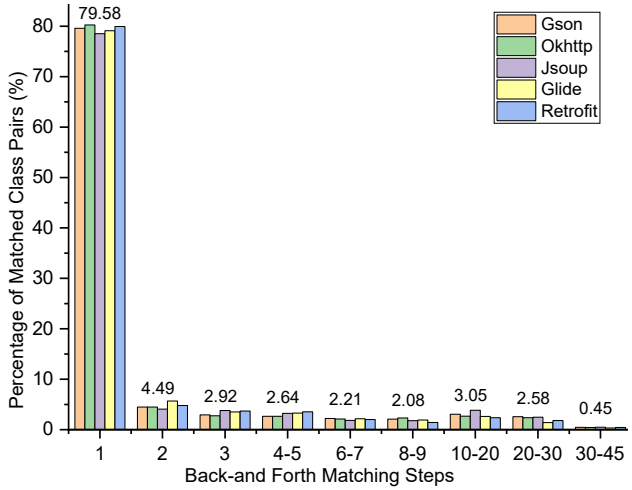

Figure 8: The distribution of back-and-forth matching steps for the most common TPLs.

to form a valid matched pair using the back-and-forth strategy, while the vertical axis denotes the proportion of matched class pairs. For these commonly used libraries, more than 20% of class candidates require multiple steps to identify a correct pair. Although this ratio tends to diminish as the number of steps increases, over 6% of classes still require more than 10 steps for accurate matching. Instances where more steps were required signify the complexity in establishing accurate matches. Our algorithm, by conducting multiple iterations of matching, adeptly incorporates surrounding information within same package as the target candidates. This iterative process gradually shifts local matches into global ones.

5.5 Real-world Application of LibScope

We undertake an evaluation of the scalability of LibScope by conducting a thorough examination of vulnerabilities across a large-scale dataset encompassing 30,000 real-world apps. Our analysis involves a comparative examination of the detection outcomes against contemporary advancements in the field, such as LibLoom and LibScan. To better demonstrate the security impact of refined features and our algorithm in real-world tasks, Scope^{I} is also considered as one of our comparison subjects. All the results have been manually verified as true positives rather than false positives.

In summary, LibScope detected 8,235 apps that utilize vulnerable TPLs. Table 6 presents the top 5 most frequently encountered TPLs with related vulnerabilities , as well as their detected numbers. Among them, LibScope unveil a total of 10,675 instances of TPLs with specific versions exhibiting vulnerabilities. However, without our algorithm, this number would sharply decrease to 9,348, as indicated by Scope^{I}'s results. The additional 1,327 detected vulnerable TPLs demonstrate that our algorithm can significantly enhance the vulnerability detection capabilities of the tool. Additionally, by refining fuzzy features, we identified at least 645 additional vulnerable TPLs compared to other tools, illustrating feature refinement can considerably enhance detection effectiveness.

From Table 6, it is evident that *Gson* stands out as the most frequently detected TPL among all three tools, registering a total of 1,425 occurrences across various versions as reported by LibScope. While *Okhttp*, *Retrofit* and *Jsoup* also exhibited a high frequency

Table 6: Detected vulnerabilities in top 5 most frequently encountered TPLs.

TPL	Vulnerabilities	LibScope	Scope^{I}	LibLoom	LibScan
Gson	CVE-2022-25647 1 more CVE...	1,425	1378	1,339	1,342
Okhttp	CVE-2020-15250 3 more CVE...	1,027	968	923	956
Retrofit	CVE-2020-15250 5 more CVE...	821	815	805	762
Jsoup	CVE-2023-26049 9 more CVE...	728	712	699	694
Log4j	CVE-2021-44228 69 more CVE...	443	409	390	393
Total	-	10,675	9,348	8,703	8,579

of occurrences, which aligns with our ground truth datasets. It is worth noting that *Okhttp* and *Retrofit* are affected by the same vulnerability, CVE-2020-15250, due to the presence of identical insecure dependency components in both. Surprisingly, *Log4j* has over 70 vulnerabilities, some vulnerabilities, such as CVE-2021-44228, can allow attackers to execute arbitrary code on users' devices.

6 Discussion & Conclusion

Threats to Validity The validity of LibScope is subject to the following threats: *1) Bias in Obfuscation Strategies.* Although we have considered commonly used obfuscation strategies, more advanced techniques, such as app packing, remain beyond our handling capabilities. Therefore, we recommend using unpacking tools [62, 63] to preprocessing packed apps before applying LibScope. Nevertheless, our evaluation covers three major obfuscators, which account for over 88% of apps [27], and LibScope demonstrates high accuracy against these tools. *2) Potential Impact of Advanced Optimizations.* Recently, R8 [64] optimizer introduced a series of advanced optimizations that can significantly alter code features [65], potentially affecting LibScope's effectiveness. For example, call-site optimization may alter method signatures, which are a crucial feature in forming LibScope class candidates. However, these optimizations are not enabled by default [66]. We plan to address the associated challenges as part of our future work.

Binary Library Detection Future work will focus on extending our approach to the detection of C/C++ libraries [67].

Conclusion Detecting third-party libraries is crucial for securing Android supply chains, yet remains challenging amidst pervasive code obfuscation. Existing methods predominantly rely on local similarity matches of fuzzy library features, overlooking broader contextual knowledge within adjacent classes. To address this, our research leverages surrounding library classes to enhance matching accuracy, transitioning from local to global matches. Our approach significantly surpasses other TPL detection methods.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable feedback and guidance in improving this paper. This work was sponsored by National Natural Science Foundation of China (No.62272351, No.62172308 and No.62172144).

References

[1] Chinenye Okafor, Taylor R Schorlemmer, Santiago Torres-Arias, and James C Davis. SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '22)*, 2022.

[2] Vikas Hassija, Vinay Chamola, Vatsal Gupta, Sarthak Jain, and Nadra Guizani. A Survey on Supply Chain Security: Application Areas, Security Threats, and Solution Architectures. *IEEE Internet of Things Journal*, 8(8), 2021.

[3] Adriana Sejfia and Max Schäfer. Practical Automated Detection of Malicious npm Packages. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*, 2022.

[4] Gede Artha Azriadi Prana, Abhishek Sharma, Lwin Khin Shar, Darius Foo, Andrew E Santosa, Asankhaya Sharma, and David Lo. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. *Empirical Software Engineering*, 26(59), 2021.

[5] Haoyu Wang and Yao Guo. Understanding Third-Party Libraries in Mobile App Analysis. In *Proceedings of the 39th International Conference on Software Engineering Companion*, 2017.

[6] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P '16)*, 2016.

[7] Tatsuhiko Yasumatsu, Takuya Watanabe, Fumihiro Kanei, Eitaro Shioji, Mitsuaki Akiyama, and Tatsuya Mori. Understanding the Responsiveness of Mobile App Developers to Software Library Updates. In *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy (CODASPY '19)*, 2019.

[8] Kaifa Zhao, Xian Zhan, Le Yu, Shiyao Zhou, Hao Zhou, Xiapu Luo, Haoyu Wang, and Yepang Liu. Demystifying Privacy Policy of Third-Party Libraries in Mobile Apps. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*, 2023.

[9] Snyk Open Source. Guide to Software Composition Analysis. https://snyk.io/series/open-source-security/software-composition-analysis-sca/, [online].

[10] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. Automatically Locating Malicious Packages in Piggybacked Android Apps. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft '17)*, 2017.

[11] Chongbin Tang, Sen Chen, Lingling Fan, Lihua Xu, Yang Liu, Zhushou Tang, and Liang Dou. A Large-Scale Empirical Study on Industrial Fake Apps. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP' 19)*, 2019.

[12] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li. Automated Poisoning Attacks and Defenses in Malware Detection Systems: An Adversarial Machine Learning Approach. *Computers & Security*, 73, 2018.

[13] Sen Chen, Lingling Fan, Chunyang Chen, Minhui Xue, Yang Liu, and Lihua Xu. GUI-Squatting Attack: Automated Generation of Android Phishing Apps. *IEEE Transactions on Dependable and Secure Computing*, 18(6), 2019.

[14] Takuya Watanabe, Mitsuaki Akiyama, Fumihiro Kanei, Eitaro Shioji, Yuta Takata, Bo Sun, Yuta Ishi, Toshiki Shibahara, Takeshi Yagi, and Tatsuya Mori. Understanding the Origins of Mobile App Vulnerabilities: A Large-Scale Measurement Study of Free and Paid Apps. In *Proceedings of 14th International Conference on Mining Software Repositories (MSR '17)*, 2017.

[15] The Hacker News. Urgent: Secret Backdoor Found in XZ Utils Library, Impacts Major Linux Distros. https://thehackernews.com/2024/03/urgent-secret-backdoor-found-in-xz.html, March 2024.

[16] Apache. Apache Log4j Security Vulnerabilities. https://logging.apache.org/log4j/2.x/security.html, 2023.

[17] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*, 2021.

[18] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review. *IEEE Transactions on Software Engineering*, 48(10), 2021.

[19] Michael Backes, Sven Bugiel, and Erik Derr. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, 2016.

[20] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. Detecting Third-Party Libraries in Android Applications with High Precision and Recall. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*, 2018.

[21] Jianjun Huang, Bo Xue, Jiasheng Jiang, Wei You, Bin Liang, Jingzheng Wu, and Yanjun Wu. Scalably Detecting Third-Party Android Libraries With Two-Stage Bloom Filtering. *IEEE Transactions on Software Engineering*, 49(4), 2022.

[22] Burton H Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7), 1970.

[23] Yafei Wu, Cong Sun, Dongrui Zeng, Gang Tan, Siqi Ma, and Peicheng Wang. LibScan: Towards More Precise Third-Party Library Identification for Android Applications. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security' 23)*, 2023.

[24] Guardsquare. ProGuard: Java Obfuscator and Android App Optimizer. https://www.guardsquare.com/proguard, 2023.

[25] Allatori. Allatori Java Obfuscator. https://allatori.com/, 2023.

[26] PreEmptive. Dasho Professional-grade Application Protection. https://www.preemptive.com/products/dasho/, [online].

[27] Yan Wang and Atanas Rountev. Who Changed You? Obfuscator Identification for Android. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft '17)*, 2017.

[28] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. Orlis: Obfuscation-Resilient Library Detection for Android. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft '18)*, 2018.

[29] Zifan Xie, Ming Wen, Haoxiang Jia, Xiaochen Guo, Xiaotong Huang, Deqing Zou, and Hai Jin. 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 '23)*, 2023.

[30] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. Automated Third-Party Library Detection for Android Applications: Are We There Yet? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*, 2020.

[31] Andrzej Ehrenfeucht. An Application of Games to the Completeness Problem for Formalized Theories. *Fundamenta Mathematicae*, 49(2), 1961.

[32] Jiexin Zhang, Alastair R Beresford, and Stephan A Kollmann. LibID: Reliable Identification of Obfuscated Third-Party Android Libraries. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*, 2019.

[33] Grigoris Ntousakis, Sotiris Ioannidis, and Nikos Vasilakis. Demo: Detecting Third-Party Library Problems with Combined Program Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*, page 2429–2431, 2021.

[34] Fabien Patrick Viertel, Fabian Kortum, Leif Wagner, and Kurt Schneider. Are Third-Party Libraries Secure? A Software Library Checker for Java. In *Proceedings of the 13th International Conference on Risks and Security of Internet and Systems (CRiSIS '18)*, 2019.

[35] Vipawan Jarukitpipat, Klinton Chhun, Wachirayana Wanprasert, Chaiyong Ragkhitwetsagul, Morakot Choetkiertikul, Thanwadee Sunetnanta, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, and Kenichi Matsumoto. V-Achilles: An Interactive Visualization of Transitive Security Vulnerabilities. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*, 2023.

[36] Jian Xu and Qianting Yuan. LibRoad: Rapid, Online, and Accurate Detection of TPLs on Android. *IEEE Transactions on Mobile Computing*, 21(1), 2020.

[37] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. LibDX: A Cross-Platform and Accurate System to Detect Third-Party Libraries in Binary Code. In *Proceedings of 27th International Conference on Software Analysis (SANER '20)*, pages 104–115, 2020.

[38] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, Scalable Detection of "piggybacked" Mobile Applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, 2013.

[39] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015.

[40] Xian Zhan, Tao Zhang, and Yutian Tang. A Comparative Study of Android Repackaged Apps Detection Techniques. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*, 2019.

[41] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In *Proceedings of the 13th annual international conference on mobile systems, applications, and services (MobiSys '15)*, 2015.

[42] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An Investigation into the Use of Common Libraries in Android Apps. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, 2016.

[43] Yue Duan, Lian Gao, Jie Hu, and Heng Yin. Automatic Generation of Non-intrusive Updates for Third-Party Libraries in Android Applications. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID '19)*, 2019.

[44] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 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 '17)*, 2017.

[45] Jie Huang, Nataniel Borges, Sven Bugiel, and Michael Backes. Up-To-Crash: Evaluating Third-Party Library Updatability on Android. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P '19)*, 2019.

[46] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, 2017.

[47] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*, 2014.

[48] Ralph C Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Proceedings of the 7th International Conference on the Theory and Applications of Cryptographic Techniques (CRYPTO '87)*, 1987.

[49] Raimundo Real and Juan M Vargas. The Probabilistic Basis of Jaccard's Index of Similarity. *Systematic Biology*, 45(3):380–385, 1996.

[50] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. In *Proceedings of the 14th International Conferenceon on Security and Privacy in Communication Systems (SecureComm '18)*, 2018.

[51] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*, 2015.

[52] Le Yu, Xiapu Luo, Chenxiong Qian, and Shuai Wang. Revisiting the Description-to-Behavior Fidelity in Android Applications. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER' 16)*, 2016.

[53] Le Yu, Xiapu Luo, Chenxiong Qian, Shuai Wang, and Hareton KN Leung. Enhancing the Description-to-Behavior Fidelity in Android Apps with Privacy Policy. *IEEE Transactions on Software Engineering*, 44(9), 2017.

[54] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on Android malware. *Computers & Security*, 51, 2015.

[55] Lei Xue, Yuxiao Yan, Luyi Yan, Muhui Jiang, Xiapu Luo, Dinghao Wu, and Yajin Zhou. Parema: An Unpacking Framework for Demystifying VM-Based Android Packers. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21)*, 2021.

[56] Yaniv David, Nimrod Partush, and Eran Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*, 2018.

[57] Androguard. Reverse Engineering and Pentesting for Android Applications. https://github.com/androguard/androguard, 2023.

[58] dex2jar. Tools to work with android .dex and java .class files. https://github.com/pxb1988/dex2jar, 2023.

[59] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*, 2016.

[60] Jordan Samhi, Tegawendé F Bissyandé, and Jacques Klein. AndroLibZoo: A Reliable Dataset of Libraries Based on Software Dependency Analysis. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR '24)*, 2024.

[61] F-Droid. Free and Open Source Android App Repository. https://f-droid.org, [online].

[62] Lei Xue, Hao Zhou, Xiapu Luo, Le Yu, Dinghao Wu, Yajin Zhou, and Xiaobo Ma. PackerGrind: An Adaptive Unpacking System for Android Apps. *IEEE Transactions on Software Engineering*, 48(2):551–570, 2022.

[63] Lei Xue, Hao Zhou, Xiapu Luo, Yajin Zhou, Yang Shi, Guofei Gu, Fengwei Zhang, and Man Ho Au. Happer: Unpacking Android Apps via a Hardware-Assisted Approach. In *2021 IEEE Symposium on Security and Privacy (S&P '21)*, 2021.

[64] Google. R8. https://r8.googlesource.com/r8/, 2024.

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

[66] Google. Shrink, obfuscate, and optimize your app. https://developer.android.com/build/shrink-code, 2024.

[67] Jiahui Wu, Zhengzi Xu, Wei Tang, Lyuye Zhang, Yueming Wu, Chengyue Liu, Kairan Sun, Lida Zhao, and Yang Liu. OSSFP: Precise and Scalable C/C++ Third-Party Library Detection using Fingerprinting Functions. In *Proceedings of the 45th International Conference on Software Engineering (ICSE' 23)*, 2023.