Version-level Third-Party Library Detection in Android Applications via Class Structural Similarity

Bolin Zhou
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
zhoubolin22@mails.ucas.ac.cn

Jingzheng Wu
Institute of Software, Chinese Academy of Sciences
Key Laboratory of System Software (Chinese Academy of Sciences)
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
jingzheng08@iscas.ac.cn

Xiang Ling*
Institute of Software, Chinese Academy of Sciences
Key Laboratory of System Software (Chinese Academy of Sciences)
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
lingxiang@iscas.ac.cn

Tianyue Luo
Institute of Software, Chinese Academy of Sciences
tianyue@iscas.ac.cn

Jingkun Zhang
Institute of Software, Chinese Academy of Sciences
zhangjingkun23@mails.ucas.ac.cn

Abstract

Android applications (apps) integrate reusable and well-tested third-party libraries (TPLs) to enhance functionality and shorten development cycles. However, recent research reveals that TPLs have become the largest attack surface for Android apps, where the use of insecure TPLs can compromise both developer and user interests. To mitigate such threats, researchers have proposed various tools to detect TPLs used by apps, supporting further security analyses such as vulnerable TPLs identification.

Although existing tools achieve notable library-level TPL detection performance in the presence of obfuscation, they struggle with version-level TPL detection due to a lack of sensitivity to differences between versions. This limitation results in a high version-level false positive rate, significantly increasing the manual workload for security analysts. To resolve this issue, we propose SAD, a TPL detection tool with high version-level detection performance. SAD generates a candidate app class list for each TPL class based on the feature of nodes in class dependency graphs (CDGs). It then identifies the unique corresponding app class for each TPL class by performing class matching based on the similarity of their class summaries. Finally, SAD identifies TPL versions by evaluating the structural similarity of the sub-graph formed by matched classes within the CDGs of the TPL and the app. Extensive evaluation on three datasets demonstrates the effectiveness of SAD and its components. SAD achieves F1 scores of 97.64% and 84.82% for library-level and version-level detection on obfuscated apps, respectively, surpassing existing state-of-the-art tools. The version-level false positives reported by the best tool is 1.61 times that of SAD. We further evaluate the degree to which TPLs identified by detection tools correspond to actual TPL classes. Experimental results show that SAD achieves a class-level F1 score of 94.12%, 11% higher than the best tool, demonstrating the reliability of SAD and better supporting downstream tasks that rely on specific code.

CCS Concepts

• **Software and its engineering → Software libraries and repositories**; Software reverse engineering; • **Security and privacy →** *Software security engineering*.

Keywords

Third-party Library, Android Application, Version Identification

ACM Reference Format:
Bolin Zhou, Jingzheng Wu, Xiang Ling, Tianyue Luo, and Jingkun Zhang. 2025. Version-level Third-Party Library Detection in Android Applications via Class Structural Similarity. In *Proceedings of The 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Third-party libraries (TPLs) in Android offer a wide range of pre-implemented functionalities, enabling developers to avoid reinventing the wheel [33]. This greatly simplifies the development process for Android apps and significantly shortens the app development and delivery cycle [32]. However, research [26, 29, 37] indicates that TPLs have become the largest attack surface within apps, and the integration of numerous TPLs introduces various security and compliance issues. Specifically, when TPLs containing vulnerabilities, malicious code, or license conflicts with other TPLs are integrated into apps, they may harm developers' interests and jeopardize user privacy and security. Many studies have been proposed to evaluate and mitigate these threats. For instance, some

* ✉ Xiang Ling is the corresponding author.

EASE 2025, 17–20 June, 2025, Istanbul, Türkiye

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

researchers propose reducing risks posed by TPLs through permission downgrading [16, 17] or process isolation [29, 44]. Wu et al. [38] employ function summaries to detect flaws in TPLs. Other works focus on extracting TPLs to identify and analyze malicious behaviors [4, 6, 7, 23], analyzing fraudulent activities in ad TPLs [12, 13], examining privacy leakage issues associated with TPLs [26, 30], and identifying and analyzing vulnerable TPLs [3, 35].

A **prerequisite task** for aforementioned works is detecting TPLs in apps. The TPL detection supports downstream reliable and effective security and compliance analysis by identifying the TPLs and their versions used in apps [3, 19, 34, 39, 40, 42]. Due to the importance of TPL detection, it has become a core technology in many commercial software composition analysis (SCA) products [8, 15, 24]. Nevertheless, although existing advanced TPL detection tools [19, 39, 40, 45] achieve library-level F1 scores exceeding 95%, their performance in version-level detection, which is crucial for many downstream tasks, remains suboptimal. This can be attributed primarily to the following two reasons. **R#1: Reliance on features susceptible to obfuscation.** Existing tools rely on code features that are vulnerable to obfuscation, making it more challenging to detect TPLs in obfuscated apps. This leads to more false negatives and degrades detection recall at both the library- and version-level. **R#2: Unable to capture differences across versions.** Existing TPL detection tools independently process each TPL without capturing the differences between different versions of TPLs, which leads to scenarios where, at the version-level, different versions are detected as being within the app, particularly when the differences are minimal, resulting in a significant number of version-level false positives. Notably, although existing studies [39, 40] have evaluated the version-level detection performance of TPL detection tools, we argue that their precision calculation methods are overly relaxed and fail to effectively reflect a more realistic version-level detection performance.

In order to address the aforementioned issues, we propose SAD, which utilizes class structural similairty and summaries of class functionality for the TPL detection. Specifically, to deal with issue **R#1**, SAD first leverages the similarity of node features, integrating CDG structural information, to construct a candidate app class list for each TPL class. It then generates class functionality summaries based on field operations for class matching. Regarding issue **R#2**, SAD extracts fine-grained features of different TPL versions, filters erroneous versions based on the number of high-confidence matched classes, and conducts cross-version comparisons to identify subtle differences, ultimately determining the precise version.

Specifically, SAD first extracts class-level features and inter-class relationships from the input TPL and app to construct CDGs, and then generates feature for each class node through the operation of neighbor feature aggregation. Then, SAD generates a candidate app class list for every TPL class by calculating the similarity between the feature of nodes in the CDGs of the TPL and app. Subsequently, SAD performs reliable class matching based on the candidate lists of TPL classes. SAD categorizes classes into two types: *stateful classes* and *stateless classes*, referring to classes with and without non-static fields, respectively. Given a pair of classes from the TPL and the app, SAD requires their types to be consistent, categorizing them into stateful or stateless pairs. For stateful pairs, SAD establishes correspondences between the methods and fields of TPL and app classes, then summarizes class functionality by translating lengthy method code into field operation representations, leveraging intra-class method invocation sequences. Class matching is then conducted based on the similarity of these summaries. For stateless pairs, SAD matches methods using field read/write operations and method opcodes, and performs class matching based on opcodes matching ratio. Finally, SAD determines the TPL version based on the structural similarity of the sub-graph formed by the matched nodes within the CDGs associated with the TPL and the app.

We evaluated the effectiveness of SAD on three datasets composed of 1123 apps, and 562 TPLs. The results show that SAD outperforms existing tools on obfuscated dataset at both the library- and version-level [39], achieving F1 scores of 97.64% and 95.35%, respectively. Particularly, SAD scores 84.82% in version†-level that we propose, which is 10% higher than the best TPL detection tool. Overall, our contributions are as follows:

- We propose a version-level TPL detection tool SAD, utilizing class structural similarity between TPLs and apps for TPL detection.
- We propose a stricter calculation method for version-level false positives. This adjustment reduces the overestimation of F1 scores and provides a more realistic measurement of tool effectiveness.
- We evaluate the TPL detection performance of different tools on three datasets. SAD achieves 97.64% accuracy at the librar-level, comparable to state-of-the-art tools. For version-level detection, it achieves the highest F1 score of 84.82% among all tools.

2 Background and Related Work

2.1 Code Obfuscation

Code obfuscation techniques are widely employed to protect app code from threats such as piracy and reverse engineering. By applying semantically equivalent transformations, these techniques make code more difficult to understand and analyze, thereby achieving the goal of safeguarding apps. However, the widespread use of obfuscation techniques poses significant challenges to TPL detection task, making external security audits and research more difficult. As a result, researchers have proposed various methods for detecting obfuscation to guide further analysis. IREA [20] is a tool that employs pattern matching and API recognition to statically detect obfuscations, including identifier renaming, reflection, and data encryption. Mirzaei et al. [27] introduced AndrODet, which defines heuristic rules for statically detecting three types of obfuscation: identifier renaming, string encryption, and control flow obfuscation. Jiang et al. [9] extract control flow graphs (CFGs) and leverage a hybrid model combining graph convolutional network and long short-term memory network to detect string encryption, identifier renaming, and control flow obfuscation at the function level. In addition to obfuscation detection, some studies have focused on directly deobfuscating obfuscated apps. DeGuard [5] is a tool designed to address identifier renaming obfuscation by learning a probabilistic model from thousands of unobfuscated apps and applying the model to restore identifier names in obfuscated apps. Yoo et al. [41] proposed a dynamic code extraction-based approach to retrieve decrypted strings from apps affected by string encryption obfuscation.

Given the limited scope of existing obfuscation detection and deobfuscation methods, which restrict their ability to serve as preprocessing steps for TPL detection, an increasing number of TPL detection tools have been designed to account for the impact of code obfuscation [19, 39, 40, 42]. Notably, as technology advances, novel obfuscation techniques continue to emerge [2], making it highly challenging to develop a tool that is resilient to all obfuscation techniques. Therefore, we primarily focus on commonly used obfuscation techniques in widely adopted obfuscation tools [1, 14, 28].

2.2 TPL Detection Tools

Existing TPL detection tools can be mainly categorized into two types based on their methodologies [43]: machine learning-based [21, 25], and similarity comparison-based [19, 39, 40, 42, 45]. Tools based on machine learning can be further categorized into classification-based and clustering-based tools. Classification-based tools are primarily designed to distinguish between ad TPLs and non-ad TPLs, but their applicability is limited. Clustering-based tools utilize a large number of apps as input to extract code features for clustering purposes, grouping similar TPLs together. The underlying principle is that popular TPLs are utilized by numerous apps. However, the limitation of such methods lies in the necessity of acquiring a substantial number of apps, which consequently restricts detection to widely used TPLs, thereby neglecting newer or less common ones. The majority of tools are feature similarity comparison-based, which do not require a large number of apps. Instead, they necessitate the construction of a feature database for TPLs, extracting features from the input apps and comparing them with those in the database to identify TPLs within the apps. This method requires pairwise comparisons, which is often time-consuming, yet it offers superior detection performance.

LibScan [39] builds potential class correspondences through fingerprint code features and then detects in-app TPLs using two fine-grained stages of method opcodes similarity and call chain opcode similarity, achieving excellent efficiency and effectiveness. LibHunter [40], building upon the foundation of LibScan, takes into account the effects of optimizations.It utilizes an enhanced class signature matching approach to address call site optimizations for constructing class correspondence relationships. Subsequently, method matching is performed using the opcodes and strings of the methods, and method inlining optimizations are processed through simulating method inlining strategies. It has demonstrated outstanding performance on optimized apps. LIBLOOM [19] converts TPL detection into a set inclusion problem, using two-stage bloom filter to extract candidate TPLs and compute similarity scores for detection, and employs a novel entropy-based metric to specifically handle apps obfuscated by repackaging and package flattening, significantly improving scalability while ensuring effectiveness. LibPecker [45] matches TPLs by generating signatures based on class dependencies for both TPL classes and app classes and introduces adaptive class similarity thresholds and weighted class similarity scores when computing TPL similarities, which makes it more resilient to obfuscations.

Although these tools have been proven to be resilient to obfuscations, the evaluation metrics at the version-level are overly relaxed, leading to overestimated performance of TPL detection tools. As a result, their performance in downstream tasks reliant on specific TPL versions is suboptimal. This motivates the development of a TPL detection tool with superior performance under more accurate and stricter version-level detection metric, thereby supporting a broader range of downstream tasks.

3 Methodology

In this section, we describe the functionality of the various modules of SAD. As shown in Figure 1, the overall framework of SAD consists of three modules. In the preprocessing module, SAD first parses the app and the TPL to extract their CDGs. It then constructs a list of candidate app classes for each TPL class based on the feature of nodes in the CDGs associated with the TPL and the app. Subsequently, SAD performs class matching by iterating over the candidate list for each TPL class using the similarity of class functionality summary. Finally, SAD validates the structural similairty of the sub-graph formed by the matched classes to detect TPLs.

3.1 Pre-processing

SAD parses the input app and TPL to extract class features and dependency relationships, subsequently constructing their respective CDGs. It then generates a list of candidate app classes for each TPL class based on the similarity of node features between the app and TPL CDGs. By leveraging rich class-level structural information and the resilience of CDGs to obfuscation, SAD minimizes potential omissions in identifying class correspondences during candidate list generation.

Table 1: Types of CDG edges and features of CDG nodes.

Element	Node/Edge Type	Feature
Node	static inner class	***static***
	abstract class	***abstract***
	other class	***default***
	interface	***interface***
Edge	extends	***extends***
	implements	***implements***

3.1.1 Class Dependency Graph. Class dependency relationships, due to their robustness compared to package structures and their incorporation of semantic and structural information of the code, are utilized by many TPL detection tools [42, 45]. However, existing tools, in an effort to enhance the feature distinguishability among classes, integrate more granular method and field information within the utilized class dependency relationships, which consequently diminishes their resilience to obfuscation. To address this issue, SAD constructs CDGs for the app and TPL, wherein classes serve as nodes, as illustrated in Table 1. CDG is a directed graph in which types of edge include **inheritance** and **implementation** relationships and types of node include class and interface. Within a complete functional module, these two types of dependencies exhibit resilience against most of obfuscation techniques. The nodes in the CDG are characterized by class modifiers; to enhance resilience against obfuscation, we only consider four types of modifiers: default, abstract, static, and interface.

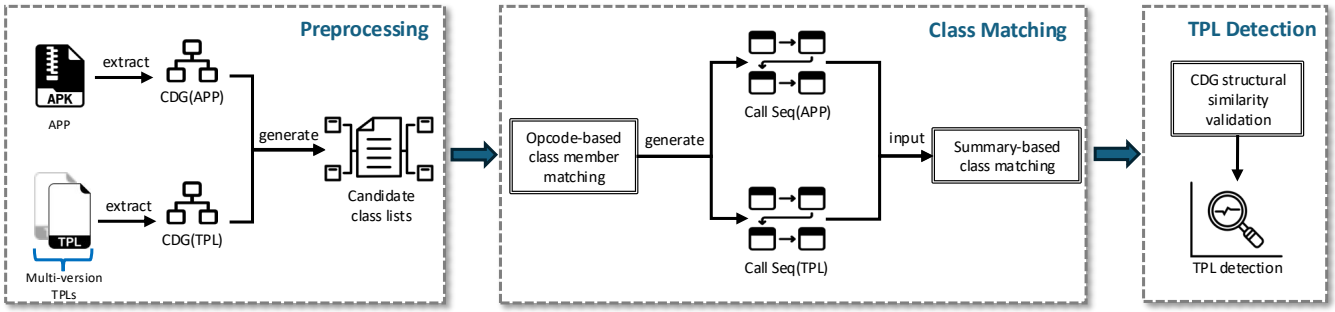

Figure 1: The overview of SAD, which includes three main modules, preprocessing, class matching and TPL detection.

3.1.2 Candidate Class List Generation. Inspired by the message passing concept in graph neural networks, SAD fully leverages the structural information of CDGs to generate a candidate app class list for each TPL class. As shown in Algorithm 1, SAD generates features for each node in the CDG (lines 4-9), then calculates the similarity of node features between the app and TPL CDGs (lines 14-17), considering the app class n_{app} as a candidate for the TPL class n_{lib} when the similarity score of the pair $\langle n_{app}, n_{lib} \rangle$ exceeds the threshold T_c. Specifically, SAD generates a feature for each node n through the following procedure: (1) iteratively aggregating the features of the neighboring nodes of n; (2) sorting the aggregated features, concatenating them with the feature of n; and (3) applying locality-sensitive hashing (LSH) to generate the node feature. The feature generating process for nodes in the CDGs of the app and TPL iterates $\texttt{Diameter}(G_{lib})$ times. The key **insight#1** here is that the feature propagation between the most distant nodes in the TPL's CDG requires exactly as many iterations as the diameter of G_{lib}. Fewer iterations may fail to propagate features to all nodes in the CDG, underutilizing the structural information, while more iterations introduce additional computational overhead. This configuration effectively mitigates the impact of graph size differences between the TPL and the app, ensuring a consistent feature generation process and enhancing the reliability of candidate lists generated based on node features.

It is worth noting that SAD aggregates only the features of the dependent nodes for each node, meaning that the propagation of features occurs in a direction opposite to that of the edges in CDG. The **insight#2** here is that the class dependencies within the app flow from the main program to TPLs. Therefore, the direction of feature propagation described above can mitigate the influence of the main program on feature generation. SAD calculates the cosine similarity of features for each node pair $\langle n_{app}, n_{lib} \rangle$ in the CDGs of app and TPL. Potential candidates are established for pairs with similarity exceeding the threshold T_c. The output of preprocessing module of SAD are the candidate app class lists for all TPL classes $\mathcal{M}$ and CDGs of the app and TPL.

3.2 Class Matching

Based on the candidate class lists $\mathcal{M}$, SAD further conducts fine-grained pairwise matching between app class and TPL class to eliminate false positive candidates. Class matching is the process of examining the syntactic and semantic consistency between each candidate app class c_a and TPL class c_l in $\mathcal{M}$.

SAD categorizes classes into *stateful classes* and *stateless classes* based on the presence of non-static fields. Stateful classes contain non-static fields, allowing changes in field values to directly reflect the object's state; conversely, stateless classes lack fields, making state changes less perceptible. Based on this distinction, SAD processes stateful and stateless classes separately, leveraging code syntactics and semantics for effective class matching. SAD first verifies whether the pair $\langle c_a, c_l \rangle$ exhibits consistent categories, terminating the matching process if the categories differ.

Slice-based Member Matching. To mitigate the impact of fine-grained code obfuscation (e.g., control flow flattening), SAD performs taint analysis using the method parameters as the source and the method exit as the sink. SAD slices the instructions based on data flows from source to sink, extracting the opcodes of the sliced instructions as the method's functional representation. For methods without parameters, SAD extracts the opcodes of all instructions. For

Algorithm 1: Candidate Class List Generation Algorithm

Input: $CDG_{app}(V_{app}, E_{app})$, $CDG_{lib}(V_{lib}, E_{lib})$, T_c.
Output: The TPL candidate nodes of app node $\mathcal{M}$.

1 **Function** GenNodesFeature(G, *miter*):
2 $L \leftarrow \{v : \texttt{LSH}(v) \mid \forall v \in V\}$
3 **for** *iter* $\leftarrow 1 :$ *miter* **do**
4 **for** *each node* $u \in V$ **do**
5 $\mathcal{S} \leftarrow \emptyset$ ▷ Initialized with empty **multiset**.
6 **for** *each neighbor* v *that* $(u, v) \in E$ **do**
7 $\mathcal{S} \leftarrow \mathcal{S} \cup L[v]$ ▷ Add label of v.
8 $l \leftarrow L[u] \oplus \texttt{Sort}(\mathcal{S})$ ▷ $\oplus$: concatenate.
9 $L[u] \leftarrow \texttt{LSH}(l)$
10 Return L
11 $\mathcal{M} \leftarrow \{\}$
12 $\mathcal{L}_{app} = \texttt{GenNodesFeature}(CDG_{app}, \textbf{Diameter}(CDG_{lib}))$
13 $\mathcal{L}_{lib} = \texttt{GenNodesFeature}(CDG_{lib}, \textbf{Diameter}(CDG_{lib}))$
14 **for** $c_l \in \mathcal{L}_{lib}$ **do**
15 **for** $c_a \in \mathcal{L}_{app}$ **do**
16 **if** $\texttt{Sim}(\mathcal{L}_{lib}[c_l], \mathcal{L}_{app}[c_a]) > T_c$ **then**
17 $\mathcal{M}[c_l] \leftarrow \mathcal{M}[c_l] \cup c_a$
18 Return $\mathcal{M}$

class pair $\langle c_a, c_l \rangle$, SAD calculates the opcodes overlap rate between the method $m_a \in c_a$ in the app and the method $m_l \in c_l$ in the TPL using the following formula:

$$overlap_{m_a,m_l} = \frac{|Op(m_a) \cap Op(m_l)|}{|Op(m_l)|}, \quad (1)$$

where $Op(\cdot)$ denotes the set of opcodes associated with the input method, and $|\cdot|$ represents the cardinality of the set. SAD extracts all method pairs that satisfy $Overlap_{m_a,m_l} > T$ and share the same method fuzzy signature [39], prioritizing the establishment of correspondences between pairs with the highest overlap rate, while ensuring that each method is matched at most once. Note that this process does not guarantee the successful matching of all methods in c_a and c_l. In the presence of method additions or deletions, which may result from obfuscations, the correspondence may diminish. SAD approach disregards the unmatched methods, thereby focusing on the matched methods and mitigating the impact of such obfuscations.

For stateless pairs, SAD calculates the proportion R_m of matched methods among all methods in c_l, and the proportion R_o of opcodes of matched methods to the total opcodes in c_l to determine the confidence score of matching:

$$CMSc_a, c_l = \frac{R_m + R_o}{2}, \quad (2)$$

Class c_a and class c_l are considered matched when $CMS(c_a, c_l) > T$. The class matching confidence is determined by integrating both ratios, thus mitigating the uneven distribution of opcodes within methods due to obfuscation. For instance, if a obfuscator inserts a string decryption method in a simple app class, which constitutes a large proportion of the opcodes in the class, but no matching method exists in the corresponding TPL class, considering opcode matching alone could result in a false negative.

Stateful classes can be more complex, and similar functionalities or operation patterns may lead to similar opcode sets, but the fields involved may differ, resulting in cases of similar syntax but different semantics. To further eliminate false positives, SAD matches fields using field read and write operations from the matched method pairs, provided that $CMS(c_a, c_l) > T$. Based on the type and frequency of field-related operations, SAD performs field matching. If the match fails, it indicates a false positive in method matching, and the method match is discarded. This step allows SAD to leverage field information to eliminate method matching false positives while enabling field matching, thereby providing a foundation for deeper semantic analysis.

Functionality Summary-based Class Matching. Existing research [39] utilize the opcodes of call chains to further mitigate false positives in method matching. However, the extensive code introduced by call chains may weaken the distinction between the opcode sets of app method and TPL method, and the generation and traversal of call graphs incur substantial runtime overhead. In contrast, SAD eliminates false positives in class matching by generating method call sequences within classes to construct diverse contexts for semantic consistency verification. Subsequently, SAD traverses these sequences to generate summaries representing class functionality, using similarity measures to assess the semantic consistency between the two classes $\langle c_a, c_l \rangle$. The method call sequence $CS_c = \{m_1, m_2, ..., m_n\}$ of a class c is an ordered list composed of n methods within the class. SAD simulates actual usage scenarios to generate method call sequences CS_{c_l} of TPL class c_l. Initially, SAD simulates object instantiation by randomly selecting a constructor `<init>` of c_l to add to the sequence. Furthermore, SAD randomly selects methods in the c_l, excluding constructors, to include in the call sequence, repeating this process S times to simulate object usage. Based on the matched methods, SAD constructs the method call sequence CS_{c_a} for the corresponding app class c_a.

To mitigate the impact of noise introduced by obfuscation, SAD distills code of each method call sequence into field operations that reflects the state changes of the object created and used through the method call sequence. Specifically, SAD identifies three types of field operations through static code analysis: *initialization*, *assignment*, and *method invocation*. For each field operation, SAD extracts used key elements, including the names and fuzzy types of matched fields, as well as the positions and fuzzy types of method parameters. Therefore, each method call sequence CS forms a corresponding field operation sequence FOS.

Since classes retain the same semantics despite obfuscation, the field operation sequence FOS should also exhibit similarity before and after obfuscation (**insight#3**). Therefore, SAD first replaces field names in the field operation sequence with unique number to eliminate the impact of identifier obfuscation, then computes locality sensitive hashing of field operations to generate class functional summaries FS. To address potential disruptions in field operation order caused by control flow obfuscation, SAD applies the Hungarian algorithm [36] ($\mathcal{H}(\cdot)$) to maximize matches between the summaries of app class FS_{c_a} and TPL class FS_{c_l}. The proportion of these maximum matches $|\mathcal{H}(FS_{c_a}, FS_{c_l})|$ to the total number of summaries $|FS_{c_l}|$ is used as the semantic similarity score for the method call sequences CS_{c_a} and CS_{c_l}. To avoid false positives caused by random method selection, SAD generates K method call sequences to construct diverse code context, and the average of the semantic matching scores of all call sequences is used as the class matching confidence score CMS.

$$CMS_{c_a,c_l} = \frac{1}{K} \sum_{i=1}^{K} \frac{|\mathcal{H}(FS^i_{c_a}, FS^i_{c_l})|}{|FS^i_{c_l}|} \quad (3)$$

SAD regards pairs with matching confidence scores exceeding T as matches. Specifically, nodes with confidence score CMS_{c_a,c_l} greater than T_h ($T_h > T$) are identified as **high-confidence matches** and are excluded from subsequent matching processes.

3.3 TPL Detection

To capture the differences between different versions of TPLs, SAD performs an analysis across all versions of the given TPL. Since the class matching module considers only intra-class information, it can incorrectly yield high matching confidence scores for structurally dissimilar classes in scenarios such as code reuse or code cloning. To address this issue, SAD leverages the structural similarity between CDGs to verify whether matched classes remain structurally consistent. However, identifying a small graph (TPL) within a larger graph (app) is fundamentally a subgraph isomorphism problem, which is one of NP problems and computationally expensive [11, 31], and obfuscation may compromise isomorphism.

We observe that when an app utilizes the TPL class c_l, to ensure functional integrity, the classes on which c_l depends must also be incorporated into the app. This dependency relationship recursively extends until encountering dependency-free classes (i.e., terminal nodes with an out-degree of 0 in the CDG), naturally forming a functional module of the TPL. Therefore, to eliminate potential false positives, SAD inspects whether the matched classes of TPL form at least one path P_{lib} to the terminal nodes in the CDG. If P_{lib} exists, SAD then examines whether the corresponding set of app nodes $N_{app}^{P_{lib}}$ also forms an identical path P_{app} in the app's CDG, where an identical path denotes passing through edges with identical features in the same order. If P_{lib} and P_{app} exist and $P_{lib} = P_{app}$, SAD considers that the CDGs of the app and TPL have similar structures.

After preliminarily verifying structural similarity, SAD calculates a confidence score, $Score_{app,lib}$, to assess the likelihood that a given TPL is integrated in the app, based on the class matching results between the app and the TPL. For the sake of clarity in notation, we define the set of high-confidence matched pairs as $HM_{app,lib} = \{(c_a, c_l) | T_h \leq CMS_{c_a,c_l}\}$, and the set of other matched pairs as $M_{app,lib} = \{(c_a, c_l) | T \leq CMS_{c_a,c_l} < T_h\}$. SAD uses the following formula to calculate the confidence score for TPL detection:

$$Score_{app,lib} = \frac{|M_{app,lib}| + \alpha \times |HM_{app,lib}|}{|\mathcal{M}|} \tag{4}$$

where $|\mathcal{M}|$ denotes the number of TPL classes annotated with candidate app classes (§3.1.2). α represents the weight of high-confidence matches, and we set it to 1.5. When $Score_{app,lib} > T_G$, SAD considers that lib is used by app. Since slight differences among versions may result in scores exceeding T_G, SAD prioritizes selecting the TPL version with the highest score as the final result. If multiple versions share the same highest score, SAD further selects the version with the largest number of high-confidence matched classes in the TPL's CDG as the final result. In certain extreme cases, the differences between various versions of TPL may be minimal (for instance, merely a change in a version-identifying string). Therefore, SAD extracts a set of literal to construct feature set $\mathbb{I}_{lib_{v_i}}$ from the matched classes of different versions lib_{v_i}, and then extracts a set of literal features $\mathbb{I}_{app}$ from the matched classes of the app. Finally, SAD calculates the intersection of the literal features of the TPL version lib_{v_i} and the app, and selecting the version with the largest intersection as the final version.

4 Experiments

In this section, we conduct experiments which are designed to answer the following four research questions.

- **RQ1 (Effectiveness)**: Can SAD achieve a higher F1 score compared to state-of-the-art TPL detection tools?
- **RQ2 (Reliability)**: Does SAD outperform existing TPL detection tools in class-level detection?
- **RQ3 (Contribution of Components)**: Do the main components of SAD contribute significantly to its performance?
- **RQ4 (Efficiency)**: How does SAD's efficiency compare with state-of-the-art TPL detection tools?

4.1 Experimental Setup

Datasets. We utilize the dataset constructed by LibScan [39] to evaluate the performance of various TPL detection tools. This dataset comprises 1,231 apps (#app and #Tuning) and 562 TPL versions as the TPL database, as presented in Table 2. To better compare the effectiveness of TPL detection tools across different types of apps, we divide it into three subsets: the unobfuscated dataset D_1, the obfuscated dataset D_2, and the optimized dataset D_3. The obfuscated dataset is generated using three obfuscation tools—Allatori, DashO, and ProGuard—with different configurations. Specifically, DashO employs four distinct obfuscation levels to produce corresponding obfuscated apps, including control flow randomization (cfr), package flattening and identifier renaming (pf-ir), dead code removal (dcr), and a combination of the three obfuscations (cfr-pf-ir-dcr). Furthermore, we identified errors in the ground truth of the LibScan dataset. After manual review and correction, the total ground truth for D_1 and D_2 increased from 5,956 to 6,168. The optimized dataset D_3 consists of 51 apps compiled with D8 and 51×3 apps compiled with R8 under different optimization configurations. R8-shrink-opt enables optimization on top of code shrinking, while R8-shrink-orlis applies Orlis's ProGuard configuration to perform repackaging and renaming obfuscation in addition to code shrinking. The 453 TPL versions in D_1 and D_2 encompass 236 distinct TPLs, with each TPL averaging ~2 versions. In D_3, the 109 TPL versions correspond to 59 unique TPLs, demonstrating a version-to-TPL ratio of 1.85 versions per TPL.

Table 2: Statistic of dataset used to evaluate SAD.

Dataset	Category	#apps	#Tuning	#libs
D_1	Non-obfus	203	22	453
D_2	Allatori	188	22	
	DashO-cfr	79	9	
	DashO-pf-ir	79	9	
	DashO-dcr	79	9	
	DashO-cfr-pf-ir-dcr	159	22	
	ProGuard	152	17	
D_3	D8-compiled	46	5	109
	R8-shrink	46	5	
	R8-shrink-opt	46	5	
	R8-shrink-orlis	46	5	

Metrics. We evaluate the performance of TPL detection tools by calculating their F1 score. To quantify library-level detection performance, we adopt the same approach as LibScan to count the *true positives* (TP_l), *false positives* (FP_l), and *false negatives* (FN_l). For version-level detection, LibScan counts TP_v, FP_v and FN_v, ensuring that $TP_l + FP_l = TP_v + FP_v$. This reflects LibScan's relaxed false positive statistical method at the version-level. For instance, if a tool reports `gson-2.4` and `gson-2.5` in an app, while the actual version is `gson-2.6`, LibScan records this as both a false negative and a false positive at the version-level. In a similar case, if a tool detects `gson-2.4`, `gson-2.5`, and `gson-2.6` in an app, LibScan considers it a true positive and disregards the incorrect versions. We argue that this statistical approach introduces bias in the assessment of version-level detection performance for TPL detection tools, as a

Table 3: Detection performance (%) of TPL detection tools (SAD, LibPecker, LibHunter, LibScan, LIBLOOM) on datasets D_1 and D_2 at library-level, version-level, and version-level†.

Dataset	Tools	Library-level			Version-level			Version-level†		
		Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
D_1	LibPecker	80.86	**99.54**	89.23	78.82	97.03	86.98	64.73	97.03	77.65
	LibHunter	80.26	93.93	86.56	80.03	93.66	86.31	61.96	93.66	74.59
	LibScan	96.96	98.88	97.91	96.83	**98.75**	97.78	68.50	**98.75**	80.89
	LIBLOOM	**99.00**	98.28	**98.64**	95.74	95.05	95.40	67.92	95.05	79.23
	SAD	97.91	99.14	98.52	**97.26**	98.48	**97.87**	**79.03**	98.48	**87.69**
D_2	LibPecker	73.41	71.49	72.44	66.89	65.14	66.01	56.30	65.14	60.40
	LibHunter	80.41	59.89	68.65	79.00	58.84	67.45	61.81	58.84	60.29
	LibScan	98.03	95.02	96.50	**96.50**	93.54	95.00	64.05	93.54	76.04
	LIBLOOM	**98.74**	92.58	95.56	95.86	89.88	92.77	66.49	89.88	76.43
	SAD	96.66	**98.65**	**97.64**	94.39	**96.33**	**95.35**	**75.76**	**96.33**	**84.82**

bold: the highest value, underlined: the second-ranked value.

tool could simply report all gson versions to easily achieve a false positive rate (FPR) of 0 at the version-level.

To more accurately assess the version-level detection performance of TPL detection tools, we propose a new method for counting true positives ($TP_{v^\dagger}$), false positives ($FP_{v^\dagger}$), and false negatives ($FN_{v^\dagger}$) at the version-level, referred to as **version-level†** to distinguish it from LibScan's approach. Specifically, if a tool correctly identifies the actual version among reported versions of the gson TPL, we count it as one $TP_{v^\dagger}$, while the remaining incorrect versions are counted as $FP_{v^\dagger}$. If the correct version is not included in the reported list, we count one $FN_{v^\dagger}$ and classify all reported versions as $FP_{v^\dagger}$. Consequently, version-level† detection satisfies the condition $TP_{v^\dagger} + FP_{v^\dagger} \geq TP_v + FP_v$.

4.2 Thresholds Tuning

To avoid biases introduced by empirical threshold settings, we extract 10% of apps from different datasets to construct validation datasets (shown in Table 2 #Tuning) for thresholds tuning of SAD and the baseline tools (i.e. LibScan, LIBLOOM, LibHunter, and LibPecker). Following LibScan's approach [39], we perform thresholds tuning separately on the obfuscated datasets D_1, D_2 and the optimized dataset D_3.

Table 4: Thresholds tuning results of different TPL detection tools.

Tools	Obfuscation		Optimization	
LibPecker	$T_{lib} = 0.7$	$T_{pkg} = 0.5$	$T_{lib} = 0.1$	$T_{pkg} = 0.05$
LibHunter	$T_{mtd} = 0.3$	$T_{lib} = 0.95$	$T_{mtd} = 0.9$	$T_{lib} = 0.2$
LibScan	$T_{class} = 0.7$	$T_{lib} = 0.85$	$T_{class} = 0.7$	$T_{lib} = 0.1$
LIBLOOM	$T_{pkg} = 0.05$	$T_{lib} = 0.9$	$T_{pkg} = 0.95$	$T_{lib} = 0.15$
SAD	$T = 0.7$	$T_G = 0.85$	$T = 0.8$	$T_G = 0.5$

For the threshold T_c of the candidate list construction, we initially constructs the ground truth of class correspondences between the apps and TPLs in the validation set, after which we employs the candidate class list construction algorithm to check whether the candidates of TPL classes contains the corresponding app classes to compute *false negative rate* (FNR) and *false positive rate* (FPR). Starting from 0, T_c is gradually increased to 1.0 in steps of 0.05. Eventually, a threshold of 0.85 is set to balance the FNR and FPR. The threshold T_h indicates high-confidence matches, which is fixed at 0.9, invariant to the dataset. For the class matching threshold T and the CDG matching threshold T_G, we perform thresholds tuning on the validation set using grid search, with each threshold ranging from 0 to 1.0, incremented by steps of 0.05. Subsequently, the thresholds yielding the highest F1 score are selected for subsequent experiments. Since the baseline tools also involve two thresholds, we apply the same grid search procedure to choose the best thresholds. The thresholds tuning results are presented in Table 4.

4.3 RQ1: Effectiveness of SAD

We utilized D_1, D_2 and D_3 to evaluate the performance of SAD in detecting TPLs and their versions in unobfuscated apps, obfuscated apps, and optimized apps. The evaluations are conducted at the library-level, version-level, and version-level†, and results are compared against four state-of-the-art TPL detection tools: LibScan [39], LIBLOOM [19], LibHunter [40], and LibPecker [45]. Table 3 presents the detailed detection performance of SAD and baseline tools on datasets D_1 and D_2. Due to the stricter statistical method of false positives (§4.1), all tools exhibit a notable performance decline at the version-level†. We find that SAD achieves detection performance comparable to state-of-the-art TPL detection tools at both the library- and version-level, while significantly outperforming the baselines at the version-level†. Specifically, SAD achieves an average improvement in F1 scores at the version-level† of 12.39% and 25.91% on the unobfuscated dataset D_1 and obfuscated dataset D_2, respectively, reaching 87.69% and 84.82%. This improvement is primarily attributed to SAD's ability to significantly reduce the number of false positive versions while maintaining a low false negative rate, demonstrating its effectiveness. As described in §4.1, the relaxed version-level FP evaluation inflates the performance of detection tools, making it appear comparable to library-level performance. However, a comparison between version-level and version-level† FPs reveals that advanced tools such as LibScan and

LIBLOOM report a substantial number of FP versions, leading to significant performance discrepancies. In contrast, SAD achieves FPs at only 58.70% and 68.04% of those reported by LibScan and LIBLOOM, respectively, while exhibiting the low false negative rate, resulting in superior performance at version-level†.

Table 5: The F1 score (%) of TPL detection tools on different obfuscation level (D_1 and D_2).

Level	Type	LibPecker	LibHunter	LibScan	LIBLOOM	SAD
Lib	Non-obfus	89.23	86.56	97.91	**98.64**	98.52
	Allatori	75.26	44.05	94.78	93.33	**96.74**
	DashO-cfr	76.67	85.19	98.27	**99.43**	98.97
	DashO-pf-ir	71.90	85.90	98.27	93.56	**98.98**
	DashO-dcr	71.69	84.64	92.53	95.09	**95.17**
	DashO-A	49.04	44.43	97.12	94.69	**97.24**
	ProGuard	87.64	88.38	98.49	**99.16**	98.45
Ver†	Non-obfus	77.65	74.59	80.89	79.23	**87.63**
	Allatori	60.78	38.39	75.62	72.76	**84.24**
	DashO-cfr	62.08	73.08	76.95	80.72	**86.35**
	DashO-pf-ir	57.56	73.78	76.31	75.73	**87.11**
	DashO-dcr	57.32	72.44	58.40	76.91	**85.22**
	DashO-A	43.69	43.56	75.16	76.53	**82.09**
	ProGuard	75.64	75.55	80.27	79.89	**86.85**

Lib: Library-level, Ver†: Version-level†, DashO-A: DashO-cfr-pf-ir-dcr.

Resilience to Obfuscations. Comparing the detection performance of different TPL detection tools on the obfuscated dataset D_2 in Table 3, we find that LibPecker reports the highest FPs, while LibHunter reports the most FNs, resulting in poorer F1 score at library-level and version-level† for both tools. The primary reason is that LibPecker uses coarse-grained class dependency signatures for matching, which fail to effectively distinguish between classes from different TPLs under obfuscation, leading to numerous FPs. In contrast, LibHunter's fine-grained features depend on strings, which cannot be successfully matched under string encryption obfuscation, resulting in a high number of FNs. In comparison, SAD, LibScan, and LIBLOOM exhibit higher resistance to obfuscation. SAD achieves the highest F1 score at the library-level, version-level, and version-level†. Notably, by effectively capturing fine-grained differences across versions to eliminate FPs, SAD improves the F1 score at version-level† by 10.98% over the best existing tool, LIBLOOM, reaching 84.82%, making it the most effective tool. Furthermore, LibScan and LIBLOOM exhibit approximately 15-fold and 11-fold increases in FPs at version-level† compared to version-level, which are obscured by the more relaxed FP statistical criteria of version-level. Consequently, their performance deteriorates most significantly at version-level†, indicating that obfuscation has a substantial impact on their version identification capabilities.

Under the DashO-cfr configuration, LIBLOOM exhibits superior F1 performance compared to SAD at the library-level. However, at the version-level†, it underperforms relative to SAD. This indicates that while coarse-grained features utilized by LIBLOOM are unaffected by control flow randomization obfuscation, they struggle to distinguish TPL versions. In contrast, SAD achieves a more effective balance between obfuscation resilience and version identification performance. Table 5 presents the F1 scores of TPL detection tools under different obfuscation levels, highlighting the varying resilience of these tools to different obfuscation configurations. Overall, Allatori and DashO-A are the most impactful obfuscation configurations, resulting in the lowest F1 scores for LibPecker, LibHunter, LIBLOOM, and SAD. In contrast, LibScan achieves the highest F1 score under DashO-A and the lowest under DashO-dcr. This discrepancy is due to the significant impact of dead code elimination on the opcode features utilized by LibScan, while the DashO-A configuration applies weaker dead code elimination on apps. Notably, SAD consistently achieves significantly higher F1 scores than baseline tools across all obfuscation configurations at the version-level†, demonstrating SAD's resilience against obfuscation in version identification.

Table 6: The F1 score (%) of TPL detection tools on different optimization level (D_3).

Level	Type	LibPecker	LibHunter	LibScan	LIBLOOM	SAD
Lib	D8	71.79	65.77	75.28	**84.07**	71.58
	shrink	77.09	74.79	69.06	**77.82**	69.18
	shrink-opt	**77.42**	57.76	15.14	61.48	47.16
	shrink-orlis	70.10	**75.21**	68.59	72.79	68.94
	Overall	73.78	68.84	62.83	**75.16**	$66.04^{\downarrow 9.12}$
Ver†	D8	63.71	61.40	23.17	**71.49**	59.96
	shrink	49.22	**67.57**	36.71	54.82	46.71
	shrink-opt	40.13	35.21	13.53	**44.85**	28.57
	shrink-orlis	42.68	**67.93**	37.04	52.10	51.83
	Overall	50.10	**59.54**	27.73	57.54	$49.31^{\downarrow 10.23}$

Lib: Library-level, Ver†: Version-level†.

Resilience to Optimizations. Although SAD is primarily designed for obfuscated apps, a certain proportion of real-world apps undergo optimization [40]. Therefore, we further evaluate the resilience of SAD against optimization using the optimized dataset D_3 constructed by [39]. Table 6 presents the F1 scores of various TPL detection tools under different optimization levels. LibHunter, designed for optimized apps, achieves the highest F1 score at the version-level†, whereas LIBLOOM performs best at the library-level, reflecting its high resilience to optimization due to the use of coarse-grained features for library-level detection. Surprisingly, LibScan performs the worst at both the library- and version-level†, primarily due to its reliance on fine-grained opcode features, which fail to handle optimization techniques, particularly under the shrink-opt configuration.

Overall, SAD experiences a decline in F1 scores, with decreases of 9.12 and 10.23 compared to the best-performing tools at the library- and version-level†, respectively. Further analysis reveals that the relatively low F1 score of SAD is mainly attributed to a high number of FNs. Manual inspection identifies two primary causes: ① incorrect TPL versions in the ground truth. Since the correct versions are not included in the TPL database and cannot be accessed, SAD's class matching module fails due to version discrepancies, ultimately affecting TPL detection. ② Excessive method reduction and removal, leading to a high number of method matching failures, which similarly impact TPL detection performance.

4.4 RQ2: Class-level Performance

Although SAD demonstrates high performance at the library-level and version-level† (§4.3) on obfuscated apps, some downstream tasks, such as TPL removal and isolation [34], require fine-grained identification of TPL code, thus imposing demands on class-level detection performance of TPL detection tools. Moreover, due to the complexity of the obfuscation and optimization, the actual range of codes relied upon by TPL detection tools when reporting TPLs remains unknown, leading to a lack of understanding regarding the reliability of tools. Therefore, we establish a one-to-one mapping between each class in the TPL and those in the app to construct the ground truth for evaluating class-level detection performance of TPL detection tools. Since the class mapping files of apps are unavailable, we manually analysis the decompiled code of both the apps and TPLs to construct the ground truth. Given the substantial workload of manual analysis, we randomly selected 20 pairs of $\langle app, lib \rangle$ from the intersection of the TPs reported by different tools in datasets D_1, D_2, and D_3 for evaluation. As LibPecker does not explicitly establish mapping between classes, we do not evaluate its class-level detection performance.

Table 7: Class-level detection performance of different TPL detection tools.

Metric	LibPecker	LibHunter	LIBLOOM	LibScan	SAD
Precision	-	93.06	91.67	**98.59**	96.70
Recall	-	69.79	68.75	72.92	**91.67**
F1	-	79.76	78.57	83.83	**94.12**

As shown in Table 7, LIBLOOM, which performs well in detection on obfuscated and optimized datasets, exhibits the poorest class-level F1. We find that LIBLOOM's method of hashing extracted features and using bloom filters for TPL detection inevitably leads to hash collisions, resulting in more false positives and false negatives at the class-level, such as incorrectly matching TPL classes with app interfaces. The class-level detection performance reflects the contribution of TPL classes to the tool's library- and version-level† detection results. A TPL detection tool with a low class-level F1 score is likely to report unreliable library- and version-level† TPs, as the identified TPL classes are not actually part of the TPL. This may lead to unpredictable detection results for different apps using the same TPL. SAD's class-level F1 score outperforms baseline tools, reflecting its superior reliability. The reason lies in SAD's fine-grained handling of stateful classes, which eliminates syntactically similar but semantically distinct candidate app classes by leveraging field-related operations and class functionality summaries (§3.2). Moreover, the class-level recall of SAD significantly outperforms the baseline tool, enabling it to better support downstream tasks that rely on specific TPL code.

4.5 RQ3: Contribution of Components

To evaluate the effectiveness of SAD in generating candidate lists for TPL classes by using structural information of CDGs, we replaced the candidate app class list generation algorithm with a signature matching algorithm to implement SAD-f. This algorithm generates class signature set for each class by extracting fuzzy signatures of its members [39], then generating a candidate app class list for the TPL class based on the overlap rate of the signature set. Furthermore, since many existing tools focus on opcodes for class matching, we aim to assess whether the class summary-based class matching step in SAD significantly improves detection performance, determining its necessity. Therefore, we perform method matching based on the opcodes of methods and verify the reliability of method matches by checking field read and write operations to eliminate false positives. We then compute the ratio of opcodes in matched methods to the total opcodes of the TPL class as the class matching confidence score. This baseline removes the step of the SAD class matching module that generate method call sequences and class functionality summaries for semantic matching, which is denoted as SAD-s.

As shown in Table 8, SAD markedly outperforms two baselines, achieving an average F1 score improvement of 22.86% and 25.32% at library-level and version-level† on three datasets, respectively, thereby demonstrating the effectiveness of the candidate class list generation algorithm and semantic matching step in class matching. By leveraging the structural information of the CDGs, SAD identifies candidate app classes for each TPL class, reducing the overhead of fine-grained class matching while effectively capturing structurally similar classes. In contrast, SAD-f, which adopts a more relaxed method and suffers from higher false positive rates. SAD-s, after removing the semantic matching step, fails to distinguish between different TPLs due to its inability to exploit semantic-level information, leading to a notable decline in both library- and version-level† detection performance.

Table 8: Effectiveness of SAD and baselines on D_1, D_2 and D_3.

Dataset	Tools	Library-level			Version-level†		
		P	R	F1	P	R	F1
D_1	SAD-f	84.00	96.91	90.00	60.82	80.34	69.23
	SAD-s	75.89	90.40	82.51	53.32	75.50	62.50
	SAD	**97.85**	**99.08**	**98.46**	**78.97**	**98.42**	**87.63**
D_2	SAD-f	85.63	75.74	80.38	59.10	59.41	59.25
	SAD-s	77.29	66.00	71.20	46.69	50.05	48.31
	SAD	**96.66**	**98.65**	**97.64**	**75.76**	**96.33**	**84.82**
D_3	SAD-f	64.72	30.85	41.78	47.94	23.83	31.84
	SAD-s	**84.69**	12.13	21.23	**70.69**	11.99	20.50
	SAD	71.24	**61.55**	**66.04**	51.60	**47.22**	**49.31**

P: Precision, R: Recall.

4.6 RQ4: Efficiency of SAD

We compare the detection time of SAD with other TPL detection tools on datasets D_1 and D_2. As shown in Table 9, we record the total processing time for each app by the TPL detection tools, and we compute the first quartile (Q1), median, third quartile (Q3), and mean of the detection times. We find that SAD is slower than LibHunter, LIBLOOM and LibScan. The reason for LIBLOOM's superior speed is its use of a two-stage bloom filter, which provides highly scalable TPL detection by converting extracted features to hash values, thereby significantly accelerating the matching phase. LibScan, on the other hand, is somewhat slower than LIBLOOM due to the time spent in opcode extraction and matching.

SAD is relatively slower due to two primary reasons. First, SAD spends considerable time in the candidate app class list generation step, with the feature propagation and matching of node features in graph structures being the main time-consuming phases. Second, SAD's precise version-level detection introduces additional time overhead by generating the class summaries of call sequences. Despite LibHunter, LibScan and LIBLOOM having less detection times compared to SAD, they exhibit inferior performance in version-level† detection performance. Considering the significance of TPL detection, we deem it reasonable for SAD to achieve better performance at the expense of certain efficiency.

Table 9: Average detection efficiency (s) of different tools on D_1 and D_2 (453 TPLs in total).

	LibPecker	LibHunter	LIBLOOM	LibScan	SAD
Q1	2,270.04	35.00	2.00	45.00	15.39
mean	2,442.74	63.95	5.00	46.05	129.78
median	2,546.37	49.00	2.56	49.00	109.68
Q3	2,779.70	75.00	3.98	55.00	192.24

5 Discussion

Limitations and Future Work. SAD cannot handle all types of obfuscations. For advanced obfuscation techniques such as reflective invocation, Dex file encryption, code virtualization, and others [2], the reliability of SAD's class matching step will be compromised, resulting in TPL detection failures. Additionally, like previous approaches, SAD tunes its thresholds separately on different datasets for evaluation. However, in real-world scenarios where some apps with mixed obfuscation and optimization statuses coexist, it remains challenging for SAD to identify an optimal threshold setting to achieve high performance.

LibHunter [40] is specifically designed for optimized apps and achieves better TPL detection performance on optimized datasets compared to SAD, but performs worse on obfuscated datasets. In real-world scenarios, it is often unclear whether the input app has undergone obfuscation or optimization. Therefore, it is necessary for TPL detection tools to cover obfuscated and optimized apps. However, as noted in the types of optimization listed by LibHunter [40], the code changes introduced by optimization may conflict with the principles of TPL detection tools to handle obfuscation. For example, to address dead code removal, SAD tolerates cases where the code in an app's class is less than that in the corresponding TPL class, whereas method inlining optimization inflates the code within app classes. Therefore, finding a balance between handling obfuscation and optimization is inherently challenging, and we leave it to future work to address.

Although we believe that SAD achieves a balance between effectiveness and efficiency, there remains room for improvement in detection efficiency compared to state-of-the-art baseline tools. The outstanding scalability of LIBLOOM stems from its reliance solely on class-level structural features and its efficient detection mechanism based on bloom filter design, making it suitable for high-throughput scenarios. However, as shown in §4.4, the hashing of features inevitably leads to collisions, compromising the class-level detection performance of LIBLOOM and reducing reliability. In contrast, both LibScan and LibHunter employ multi-process implementations to fully utilize CPU resources for accelerating the detection process, offering an effective approach to enhance the efficiency of SAD.

Threats to validity. SAD leverages the structural information of CDGs to generate candidate class lists and assesses structural similarity of matched nodes during TPL detection to identify specific versions. This design exhibits resilience to code obfuscation, as class dependency relationships within a complete functional module are typically unaffected by obfuscation. The class matching module, based on class summary similarity, extracts parameter-relevant instruction slices for member matching and generates class summaries using field operations, mitigating the impact of obfuscations like control flow flattening. However, in optimized apps employing method inlining, the inlined methods (callees) may fail to match, and the inflated code at the unique invocation site (caller) could lead to member matching failures, thereby affecting class matching results. Furthermore, SAD demonstrates limited resilience to more advanced obfuscation techniques, such as method overloading and reflection invocations [2]. Nonetheless, the applicability of these advanced obfuscation techniques is constrained, as they impose significant overhead to maintain app functionality and performance.

Furthermore, SAD is intentionally crafted to identify Java libraries (JAR) as well as Android libraries (AAR). Consequently, its generalizability to native libraries within apps may be limited, as native libraries may employ complex obfuscation techniques that SAD does not account for (e.g., code virtualization [18]). However, code virtualization inevitably incurs the overhead of virtual machine interpretation and execution, making it typically used to protect critical functions. Existing research on deobfuscating virtualized code [10, 22] may contribute to the detection of native libraries.

6 Conclusion

We proposed SAD, a class structural similarity-based version-level TPL detection tool. SAD generates candidate app class lists for TPL classes using the feature similarity of nodes in CDGs associated with the app and TPL and then performs class matching based on the similarity of class functionality summary. Finally, SAD achieves version-level TPL detection by identifying structural similarity between the sub-graphs formed by matched classes within the app and TPL CDG. Experimental results show that SAD outperforms baseline tools in both library-level and version-level† detection, achieving F1 scores of 97.64% and 84.82% on obfuscated dataset, respectively, demonstrating its effectiveness. Additionally, the superior performance of SAD in class-level detection underscores its reliability, making it well-suited for downstream tasks that depend on specific TPL code. The source code of the SAD and the experimental results are available at https://zenodo.org/records/15238860.

Acknowledgments

This paper is supported by the National Natural Science Foundation of China under No.62202457 and the Open Source Community Software Bill of Materials (SBOM) Platform under No.E3GX310201. This paper is also supported by YuanTu Large Research Infrastructure.

References

[1] 2005-2024. *ALLATORI JAVA OBFUSCATOR*. https://allatori.com/

[2] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. 2020. Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX* 11 (2020), 100403. https://doi.org/10.1016/j.softx.2020.100403

[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) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 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 *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 1021–1036. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bhoraskar

[5] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 343–355. https://doi.org/10.1145/2976749.2978422

[6] Gong Chen, Wei Meng, and John Copeland. 2019. Revisiting Mobile Advertising Threats with MAdLife. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 207–217. https://doi.org/10.1145/3308558.3313549

[7] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In *2016 IEEE Symposium on Security and Privacy (SP)*. 357–376. https://doi.org/10.1109/SP.2016.29

[8] Huawei Cloud. 2025. *CodeArts Governance*. https://console.huaweicloud.com/devsecurity/

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

[10] Kevin Coogan, Gen Lu, and Saumya Debray. 2011. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '11)*. Association for Computing Machinery, New York, NY, USA, 275–284. https://doi.org/10.1145/2046707.2046739

[11] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento, et al. 2001. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*. Citeseer, 149–159.

[12] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F. Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. 2018. FraudDroid: automated ad fraud detection for Android apps. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/3236024.3236045

[13] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Guoai Xu, and Shaodong Zhang. 2018. How do Mobile Apps Violate the Behavioral Policy of Advertisement Libraries?. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications* (Tempe, Arizona, USA) *(HotMobile '18)*. Association for Computing Machinery, New York, NY, USA, 75–80. https://doi.org/10.1145/3177102.3177113

[14] Guardsquare. 2016-2024. *ProGuard*. https://www.guardsquare.com/proguard

[15] Guardsquare. 2025. AppSweep. https://platform.guardsquare.com/

[16] Mahmoud Hammad, Hamid Bagheri, and Sam Malek. 2017. Determination and Enforcement of Least-Privilege Architecture in Android. In *2017 IEEE International Conference on Software Architecture (ICSA)*. 59–68. https://doi.org/10.1109/ICSA.2017.18

[17] Mahmoud Hammad, Hamid Bagheri, and Sam Malek. 2019. DelDroid: An automated approach for determination and enforcement of least-privilege architecture in android. *Journal of Systems and Software* 149 (2019), 83–100. https://doi.org/10.1016/j.jss.2018.11.049

[18] Zhongkai He, Guixin Ye, Lu Yuan, Zhanyong Tang, Xiaofeng Wang, Jie Ren, Wei Wang, Jianfeng Yang, Dingyi Fang, and Zheng Wang. 2019. Exploiting Binary-Level Code Virtualization to Protect Android Applications Against App Repackaging. *IEEE Access* 7 (2019), 115062–115074. https://doi.org/10.1109/ACCESS.2019.2921417

[19] Jianjun Huang, Bo Xue, Jiasheng Jiang, Wei You, Bin Liang, Jingzheng Wu, and Yanjun Wu. 2023. Scalably Detecting Third-Party Android Libraries With Two-Stage Bloom Filtering. *IEEE Transactions on Software Engineering* (Apr 2023), 2272–2284. https://doi.org/10.1109/tse.2022.3215628

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

[21] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: scalable and precise third-party library detection in android markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/icse.2017.38

[22] Mingyue Liang, Zhoujun Li, Qiang Zeng, and Zhejun Fang. 2018. Deobfuscation of Virtualization-Obfuscated Code Through Symbolic Execution and Compilation Optimization. In *Information and Communications Security*, Sihan Qing, Chris Mitchell, Liqun Chen, and Dongmei Liu (Eds.). Springer International Publishing, Cham, 313–324.

[23] Tianming Liu, Haoyu Wang, Li Li, Xiapu Luo, Feng Dong, Yao Guo, Liu Wang, Tegawendé Bissyandé, and Jacques Klein. 2020. MadDroid: Characterizing and Detecting Devious Ad Contents for Android Apps. In *Proceedings of The Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. Association for Computing Machinery, New York, NY, USA, 1715–1726. https://doi.org/10.1145/3366423.3380242

[24] Scantist Pte. Ltd. 2024. *Scantist Software Composition Analysis*. https://scantist.io/

[25] 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 Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 653–656. https://doi.org/10.1145/2889160.2889178

[26] Wei Meng, Ren Ding, Simon P Chung, Steven Han, and Wenke Lee. 2016. The Price of Free: Privacy Leakage in Personalized Mobile In-Apps Ads.. In *NDSS*. 1–15.

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

[28] PreEmptive. 2024. *Java Security and Android Obfuscation with DashO*. https://www.preemptive.com/products/dasho/

[29] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. 2012. AdSplit: Separating Smartphone Advertising from Applications. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 553–567. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/shekhar

[30] Yutian Tang, Xian Zhan, Hao Zhou, Xiapu Luo, Zhou Xu, Yajin Zhou, and Qiben Yan. 2019. Demystifying Application Performance Management Libraries for Android. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 682–685. https://doi.org/10.1109/ASE.2019.00069

[31] Julian R Ullmann. 2011. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *Journal of Experimental Algorithmics (JEA)* 15 (2011), 1–1.

[32] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A measurement study of google play. *ACM SIGMETRICS Performance Evaluation Review* (Jun 2014), 221–233. https://doi.org/10.1145/2637364.2592003

[33] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. 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* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/2771783.2771795

[34] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. 2018. ORLIS: obfuscation-resilient library detection for Android. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. https://doi.org/10.1145/3197231.3197248

[35] Takuya Watanabe, Mitsuaki Akiyama, Fumihiro Kanei, Eitaro Shioji, Yuta Takata, Bo Sun, Yuta Ishi, Toshiki Shibahara, Takeshi Yagi, and Tatsuya Mori. 2017. Understanding the Origins of Mobile App Vulnerabilities: A Large-Scale Measurement Study of Free and Paid Apps. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 14–24. https://doi.org/10.1109/MSR.2017.23

[36] Wikipedia. 2024. *Hungarian algorithm*. https://en.wikipedia.org/wiki/Hungarian_algorithm

[37] Stan Wisseman. 2016. Third-party libraries are one of the most insecure parts of an application. https://techbeacon.com/security/third-party-libraries-are-one-most-insecure-parts-application

[38] Rongxin Wu, Yuxuan He, Jiafeng Huang, Chengpeng Wang, Wensheng Tang, Qingkai Shi, Xiao Xiao, and Charles Zhang. 2024. LibAlchemy: A Two-Layer Persistent Summary Design for Taming Third-Party Libraries in Static Bug-Finding Systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 105, 13 pages. https://doi.org/10.1145/3597503.3639132

[39] 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 23)*. USENIX Association, Anaheim, CA, 3385–3402. https://www.usenix.org/conference/usenixsecurity23/presentation/wu-yafei

[40] Zifan Xie, Ming Wen, Tinghan Li, Yiding Zhu, Qinsheng Hou, and Hai Jin. 2024. How Does Code Optimization Impact Third-party Library Detection for Android Applications?. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association

for Computing Machinery, New York, NY, USA, 1919–1931. https://doi.org/10.1145/3691620.3695554

[41] WooJong Yoo, Myeongju Ji, M Kang, and Jeong Hyun Yi. 2016. String deobfuscation scheme based on dynamic code extraction for mobile malwares. *IT Convergence Practice* 4, 2 (2016), 1–8.

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

[43] Xian Zhan, 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 Transactions on Software Engineering* 48, 10 (2022), 4181–4213. https://doi.org/10.1109/TSE.2021.3114381

[44] Xiao Zhang, Amit Ahlawat, and Wenliang Du. 2013. AFrame: isolating advertisements from mobile applications in Android. In *Proceedings of the 29th Annual Computer Security Applications Conference* (New Orleans, Louisiana, USA) *(ACSAC '13)*. Association for Computing Machinery, New York, NY, USA, 9–18. https://doi.org/10.1145/2523649.2523652

[45] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. 2018. Detecting third-party libraries in Android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 141–152. https://doi.org/10.1109/SANER.2018.8330204

Received 31 January 2025; accepted 22 March 2025