2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Shrunk, Yet Complete: Code Shrinking-Resilient Android Third-Party Library Detection

Jingkun Zhang[†,‡], Jingzheng Wu[†], Xiang Ling[†], Tianyue Luo[†], Bolin Zhou[†,‡], and Mutian Yang[§]
[†]*Institute of Software, Chinese Academy of Sciences, Beijing, China*
[‡]*University of Chinese Academy of Sciences, Beijing, China*
[§]*Beijing VuLab Technology Co.Ltd, Beijing, China*
{zhangjingkun23, zhoubolin22}@mails.ucas.ac.cn,
{jingzheng08, lingxiang, tianyue}@iscas.ac.cn, mutian@vulab.com.cn

***Abstract*—Managing third-party libraries is a costly and critical task for enterprises, essential for both vulnerability assessment and license compliance. Existing android software composition analysis tools focus on mitigating code obfuscation but neglect the impact of code optimization, which is deeply integrated into build pipelines and disrupts library structure.**

To tackle these challenges, we developed LibSleuth, a detection tool designed to be resilient to code shrinking and obfuscation. It is based on the observation that even after shrinking, the remaining code still retains functional completeness. LibSleuth adopts two novel strategies: (1) Method level functional module matching: We break down feature matching to method level and define a functional module as related methods that represent used functionality. This allows us to detect libraries based on functional module completeness to address code shrinking. (2) Context-enhanced multi-level filtering: To improve robustness against obfuscation and reduce the cost of pairing, LibSleuth leverages contextual relationships to enhance feature stability and adopts a coarse-to-fine progressive matching process.

We evaluated LibSleuth on datasets containing obfuscated and optimized Android apps. LibSleuth outperforms state-of-the-art academic and commercial tools in both scenarios. Under combined code shrinking and obfuscation, LibSleuth achieves an average 27.74% higher version level F1-score. Moreover, our analysis of 10,000 real world Android apps shows that 20.35% still depend on vulnerable library, demonstrating the practical utility of LibSleuth for downstream tasks.

***Index Terms*—Android, Third-Party Library, Software Composition Analysis, Code Obfuscation, Code Shrinking**

I. Introduction

Code reuse is one of the most common practices in software development, especially for enterprise developers. An industrial report [1] shows that 97% of apps integrate open source software to reduce development costs. However, the widespread adoption of code reuse also greatly expands the attack surface of software [14], [46]. According to an analysis [2], 82% of open source software components are considered “inherently risky.” As a result, establishing a reliable mechanism for managing the software supply chain has become an urgent and critical task for developer.

In the mobile ecosystem, the Android platform faces particularly severe risks from third-party libraries (TPLs). The Maven Central Repository [3], which hosts over 50 million open source TPLs, is the most widely used platform in the Android development ecosystem. Developers can easily integrate TPLs via Maven [5] or Gradle [6]. While this development model improves efficiency and lowers technical barriers, it also increases the likelihood that hidden security threats are introduced into software products. For example, the Log4j [7] remote code execution vulnerability [8] discovered in 2021 affected more than 8% of TPLs in the Maven ecosystem.

To mitigate such risks, the TPL detection technique in software composition analysis (SCA) play a crucial role [13], [45]. For application developers, security vendors, and app marketplace operators, accurate TPL detection is critical for proactive vulnerability and license management [4], [21], [22]. This work focuses on enhancing TPL detection techniques for Android apps, particularly in binary analysis scenarios where source code is unavailable.

In Android app development, code obfuscation and optimization are widely used to protect intellectual property and enhance app performance [9], [10]. However, these techniques also introduce challenges for TPL detection. Obfuscation techniques, such as renaming identifiers and flattening package structures, are designed to make reverse engineering more difficult. Code optimization, typically performed by the compiler, aims to improve runtime performance and reduce app size. For instance, compilers may apply optimizations like constant folding and function inlining, or remove unused methods, classes, and packages at higher levels, which is known as code shrinking. A study [11] shows that 41.1% of projects on the F-Droid platform [15] are built using R8 [16], a tool that supports obfuscation and optimization. Our further statistics reveal that among projects using R8, 82.0% apply obfuscation, 60.2% enable optimization, and 99.6% activate shrinking. These techniques modify observable code features at the source level after compilation, making it harder to detect TPLs in the binary, and thereby increasing the cost of security audits and compliance checks for organizations.

Android TPL detection tools have evolved in recent years to address challenges posed by code obfuscation and optimizations. Several commercial SCA solutions are available on the market, including AppSweep by Guardsquare [56], CodeArts Governance by Huawei [53], etc. However, our evaluation

Jingzheng Wu and Xiang Ling are the corresponding authors.

in Section 5.3 shows that these industrial tools offer limited effectiveness when applied to code transformation scenarios, particularly in the presence of code shrinking. Code shrinking disrupts the software structures and code patterns that these tools rely on, leading to lots of false negatives. In parallel, most academic approaches extract method level features and apply fuzzy hashing techniques to reduce the impact of feature changes, then aggregate these features to form class or package level representations for library matching. For example, LibLoom [19] addresses repackaging and package flattening by using Bloom filters [25] and entropy-based similarity. LibScan [18] uses class signatures, method opcodes, and call chain opcodes for multi level matching. LibHunter [11] models inlining and callsite optimization and uses opcode and string features to counter the impact of code optimizations. However, this bottom-up aggregation framework has a key limitation: if some methods are missing, the resulting higher level features become incomplete, leading to reduced detection accuracy. In summary, most tools focus on tolerating feature changes but rarely address the issue of feature loss.

To address the above limitations, we focus on how to handle the impact of code shrinking in Android TPL detection, while maintaining robustness against code obfuscation.

The first challenge we face is: how can we still identify TPLs when some features are missing due to code shrinking? Android apps are structured in three levels: package, class, and method. Traditional detection tools usually focus on matching features at the class or package level and use a bottom-up feature aggregation strategy (a class feature is built from features of its methods, and package features rely on their classes). These tools typically aim to detect the complete library. Traditional tools rely on a key assumption: the structure of the TPL remains complete in the app. However, code shrinking removes unused methods starting from the bottom method level and gradually removes unused classes and packages, which breaks the library's structure. As a result, the aggregated higher level features become incomplete, reducing the effectiveness of traditional detection tools. Therefore, detecting libraries after code shrinking has become a major challenge.

To overcome this limitation, we propose a new detection strategy (S1): shifting the feature matching unit down to the method level and changing the detection target from a complete library to a functional module. This strategy is based on the behavior of code shrinking [16]: methods are the smallest unit of removal, and any method that remains must have a complete call chain to ensure proper functionality. Specifically, by using methods as the basic unit for matching, we no longer need to aggregate features from the bottom up, and detection is not affected by the library's structure. Instead of identifying the entire library, we detect the completeness of the functional module. To achieve this, our approach requires that all methods invoked by any matched method must also be present in the matched set. Coincidentally similar method clusters are unlikely to satisfy this dependency-closure property, which effectively filters out false positives. In short, the detection logic is designed to work in reverse of the code shrinking process. While shrinking removes unused methods and retains functionality, we trace back from the remaining methods to identify the functional module they form.

Although method level detection can avoid the problem of broken library structures, it also brings new challenges. A typical app file usually contains between 10^4 to 10^5 methods, which is several times more than the number of classes. If we try to achieve precise matching by comparing fine-grained features one by one, the matching process becomes extremely complex and time-consuming. Furthermore, a single method contains less information compared to a class or a package, making it more fragile to code obfuscation. These factors create a trade-off between accuracy and efficiency, leading to **the second challenge**: How can we match a large number of methods quickly and accurately, even under code obfuscation?

Inspired by search engines' hierarchical retrieval mechanisms [26], we propose multi-granularity filtering with features of increasing granularity. This process begins with lightweight, coarse-grained matching to quickly narrow down candidates, followed by a fine-grained and time-consuming analysis to ensure precise detection. This design significantly reduces overall computational complexity. Regarding feature engineering, relying on a single type of feature is insufficient due to the diverse impact of code obfuscation. Therefore, it is necessary to integrate multiple features and their contextual relationships to improve the robustness of method representation and resistance to obfuscation. In response to the above challenge, we propose a solution strategy (S2): enhancing the robustness of features through contextual relationships and adopting a multi-granularity filtering strategy to reduce matching complexity.

To address the challenges posed by code shrinking and obfuscation, we engineered LibSleuth, a novel and robust detection tool. LibSleuth performs detection in two stages. Firstly, in the detection stage, LibSleuth progressively filters candidate methods using a four-step strategy: fuzzy signature based, semantic element based, instruction based, and class consistency based. It then constructs a functional module based on method dependency graphs to enhance resilience against code shrinking. Secondly, in the selection stage, LibSleuth finalizes matched TPLs and their versions by calculating aggregated similarity scores and resolving method conflicts across libraries using uniquely matched method ratios. We conducted comprehensive evaluations on real world Android apps, benchmarking LibSleuth against both academic state-of-the-art tools [11], [18], [19], [32] and commercial SCA tools developed by security vendors. The results show that LibSleuth outperforms existing tools, especially in version level detection, and remains effective even under code obfuscation and shrinking. Furthermore, we deployed LibSleuth in a real-world vulnerable library detection scenario by analyzing 10,000 Android apps from the AndroZoo dataset [20] and identified 2,517 instances of vulnerable library versions. In addition to closed source scenarios, we also conducted experiments in open source environments, further validating the effectiveness of LibSleuth in performing vulnerability detection tasks.

This work makes the following key contributions:

- We designed and developed LibSleuth, a robust Android TPL detection tool that is resilient to code obfuscation and code shrinking, and capable of precise version level identification. (https://github.com/tobbykun/LibSleuth-code)
- We carried out a comprehensive experimental evaluation, demonstrating LibSleuth's superior accuracy and robustness compared to existing academic state-of-the-art tools and available commercial SCA products.
- We performed real-world vulnerable TPL detection and identified the usage of known vulnerable TPL versions.

II. Background and Motivation

A. *Code Obfuscation and Optimization*

Code obfuscation and optimization are techniques that transform code without changing functionality [12], collectively termed code transformations. Obfuscation increases code entropy to hinder reverse engineering and access to sensitive logic, while optimization improves performance by simplifying code and reducing redundancy. Both techniques are integrated into the app build process through the toolchains.

Although these techniques are originally designed to improve software protection and performance, they also introduce substantial analysis costs and blind spots for security and compliance teams. Moreover, malicious developers may misuse these techniques to evade detection, posing challenges to tasks like binary composition analysis and malicious behavior detection [23], [37]. In the following sections, we introduce commonly used code transformation tools in Android development and describe the categories of transformation techniques.

1) ***Code Transformation Tools****:* Android code transformation tools fall into two types. The first group is the default components provided by the Android Gradle plugin [27]. They are easily configurable through project settings. The earliest one is ProGuard [28], which follows a four-stage pipeline: shrinking, optimizing, obfuscating, and pre-verifying. It supports techniques such as identifier renaming, repackaging, and peephole optimization. Since Gradle plugin version 3.4.0, R8 has replaced ProGuard as the default tool. It improves upon ProGuard with more effective shrinking.

The second group consists of commercial obfuscation tools that require additional installation but provide more advanced capabilities. For example, Allatori [29] is a second-generation obfuscator for Java apps, offering capabilities like string encryption, control flow obfuscation, and code watermarking to better protect intellectual property. DashO [30] is a third-generation Java obfuscator with more sophisticated control flow obfuscation and built-in tamper detection mechanisms.

2) ***Code Transformation Types****:* Based on the impact on different code elements, we categorize common code transformations into three primary types:

- **Alteration:** Reconstruct code elements at various levels without affecting functionality. For example, identifier renaming, control flow flattening, and repackaging. These reduce code readability and obstruct reverse engineering.
- **Insertion:** New code elements or functionalities are introduced into the program, which may consist of code snippets with practical purposes or unreachable branches. This transformation increases code complexity, commonly through techniques like dead code insertion and string encryption.
- **Elimination:** Redundant or unused code elements are removed. At the method level, this includes eliminating unused variables or unreachable basic blocks. At higher levels, it involves removing unused methods, classes, or entire packages. Such transformation poses a challenge for detection tools that rely on complete library features, as their detection targets may have been significantly simplified.

Notably, shrinking belongs to code optimization by definition. However, it is handled separately in R8.

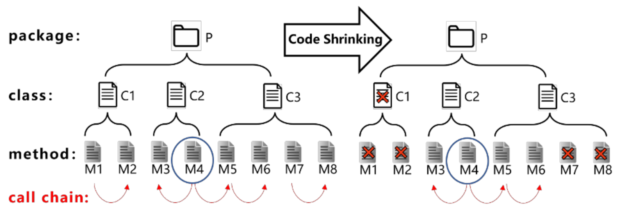


Fig. 1. An example of code shrinking

B. *Motivation*

Reliable SCA is vital for enterprises and regulatory bodies to manage vulnerabilities and compliance. However, code transformation techniques present significant and often underestimated challenges, with code shrinking being one of the most disruptive. As a fundamental step in code optimization, code shrinking eliminates unused code elements through reachability analysis. In practice, partial library importing can also be seen as a form of intentional code shrinking performed by developers. Both approaches essentially involve pruning parts of the code, resulting in irreversible damage to the software's structure. To ensure that the remaining code maintains complete functionality, code shrinking must carefully handle dependencies and method invocation relationships.

Taking R8 as an example, as shown in Figure 1, a simplified library L contains package-class-method hierarchies with intra- and inter-class method calls. If an app imports library L and calls method $M4$, R8 shrinks code by analyzing reachability from entry points and developer-specified Keep-Rules, removing unreachable methods (e.g., $M1$, $M2$, $M7$, $M8$), classes (e.g., $C1$), and empty packages. Most existing Android TPL detection tools rely on feature aggregation at the class/package level. These approaches assume the library structure remains complete. For example, features of class $C3$ derive from its methods $M5$–$M8$. After shrinking, the structure breaks: missing methods lead to incomplete class/package features, causing detection blind spots.

From the perspective of the optimizer, code shrinking operates at the method level. To align detection with code shrinking, the basic detection unit should be shifted from classes or packages down to methods. Since code shrinking removes methods but keeps their internal features intact, method level feature completeness is preserved. To verify that matched methods (M_m) represent a library residue, we use a key principle of shrinking: the remaining TPL must form functionally complete modules with preserved call dependencies. We can check whether M_m constitute a valid functional module as a reference for library detection.

We build a call graph G from M_m (as root nodes) by expanding along call edges, and define functional completeness as $Sim_{\text{mod}} = \frac{|M_m|}{|\text{Methods in } G|}$. If $Sim_{\text{mod}} = 1$, M_m forms a complete functional module. Under full-library matching, the library completeness is defined as $Sim_{\text{lib}} = \frac{|M_m|}{|\text{Methods in } L|}$. As shrinking causes $|M_m| \ll |\text{Methods in } L|$, a low similarity threshold T_{lib} maintains recall but risks false positives.

III. Approach

A. Overview

In this section, we present LibSleuth, a novel Android TPL detection tool with resilience to code obfuscation and shrinking. As shown in Figure 2, LibSleuth consists of two stages: detection and selection.

Detection stage: This stage analyzes apps against candidate TPLs and forwards likely matches to the selection stage. It is composed of two components: (1) Method matching component progressively filters candidate methods using different granularity code features, with the goal of establishing a one-to-one mapping between the methods in the TPL and those in the Android app. (2) Functional module matching component constructs a call graph via matched methods to represent the library functionality. The method matching rate of the functional module is then used to calculate the library similarity score that is provided to the selection stage.

Selection stage: When multiple TPLs are detected, the selection stage determines the final matched TPLs and their versions based on the detection results. First, if a library has multiple versions, the version with the highest similarity score is chosen as the best match. Furthermore, LibSleuth uses a greedy strategy to select the optimal TPL set by maximizing unique method matches, avoiding conflicts in method matching between multiple TPLs.

B. Detection Stage

The detection stage is divided into two components: method matching and functional module matching.

1) ***Method Matching****:* This component is the detailed design of S2, aimed at achieving efficient and accurate matching of large volumes of methods under code obfuscation. This component uses a four-step progressive filtering mechanism to pair methods from TPLs with the best matching methods in the app. The resulting method mappings are then passed to the functional module matching component. Each step is designed with features enhanced by contextual information.

Description based filtering. Initially, each TPL method considers all app methods as candidates, leading to tens of millions of pairs. Thus, coarse-grained filtering is needed to quickly narrow candidates while ensuring high recall.

Method signatures are typically composed of descriptive information such as identifiers (class names, method names, and parameter names), access modifiers, and parameter data types. Among these features, developer-defined identifiers are fragile and prone to obfuscation. Similarly, access modifiers can be affected by obfuscation. For example, the ProGuard option "$*/marking/private$" [35] attempts to mark elements as private whenever possible. To address this, there are two possible strategies: either avoid using fragile features or apply fuzzification to reduce their sensitivity. Based on this consideration, we select parameter data types as the basis for our method signature design, which is the most stable descriptive information. In particular, data types defined by the system (such as int, java.lang.Object, and android.view.ViewPager, etc.) are rarely modified during obfuscation. Therefore, LibSleuth constructs fuzzy method signatures using these parameter types. Furthermore, we replace all non-Java and non-Android basic types with a wildcard "X".

To enrich fuzzy signatures, we add inheritance context as a supplementary feature. We observed that class inheritance hierarchies tend to remain stable. Although intermediate classes may be renamed, the topmost system-defined class usually remains unchanged. Additionally, the number of inheritance levels is consistent. For example, in Figure 3, for the method $FragmentTransaction$, we trace upward from its declaring class $BackStackRecord$ through the inheritance chain until reaching the first system-defined class. The name of this superclass ($Ljava/lang/Object$) and the number of inheritance levels (2) are then used as additional features in the signature.

During feature extraction, LibSleuth builds an index that maps fuzzy method signatures to corresponding method sets. In the filtering phase, each method in a TPL can use its fuzzy signature to quickly retrieve a candidate set of matching methods from the app's index. This enables efficient and resilient matching even in the presence of code obfuscation.

Semantic element based filtering. After description based filtering, the average number of candidate matches for each method is reduced to a few dozen. Since method signatures only describe the external interface of a method, the second step needs to further filter candidates using internal information within the method. We refer to the APIs called by the method, the fields it accesses, and the strings it operates on as semantic elements. These internal features reflect the functional behavior of the method. However, semantic elements are not entirely stable and may be affected by obfuscation. For example, identifier renaming can change parts of the descriptive attributes of semantic elements, and string encryption can hide the plain-text content of strings.

Typically, obfuscation usually does not remove semantic elements or change the purpose of how they are used. Therefore, similar to the approach used in constructing fuzzy method signatures, LibSleuth extracts robust descriptive information from semantic elements to build element signatures, and also applies fuzzy processing to the data type features. To better use the contextual information of semantic elements, LibSleuth analyzes the parameters that have data flow relationships with

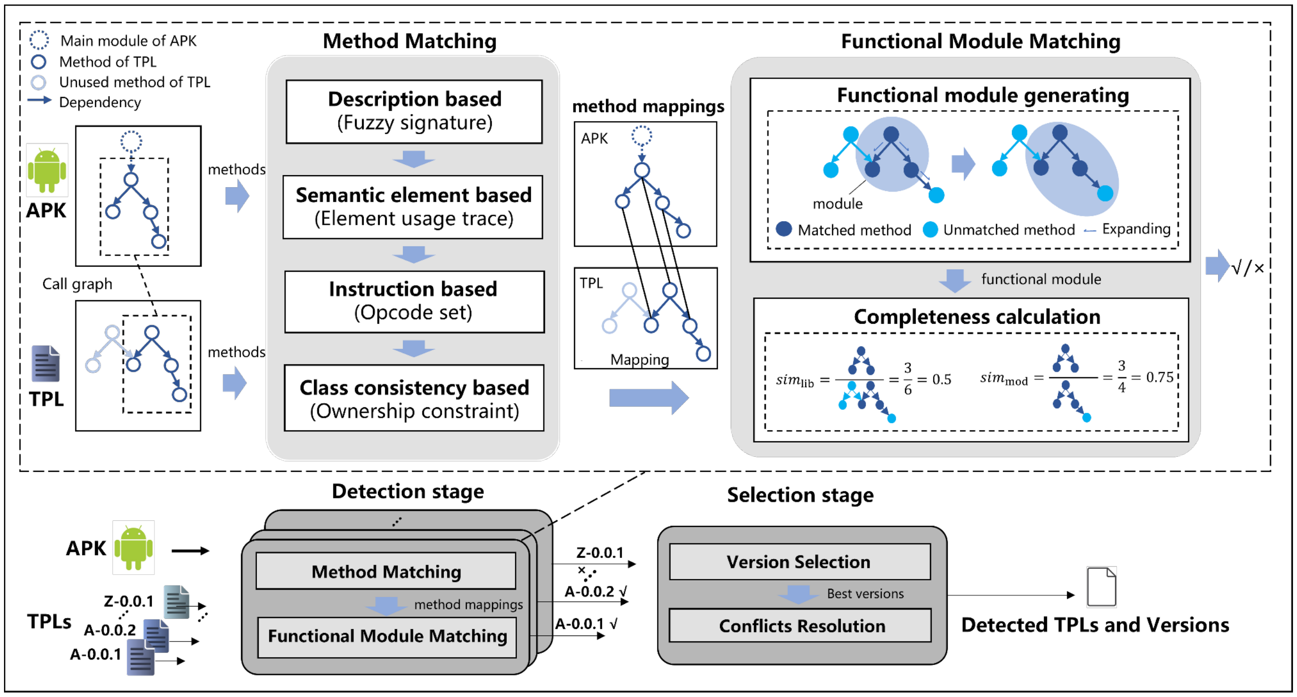
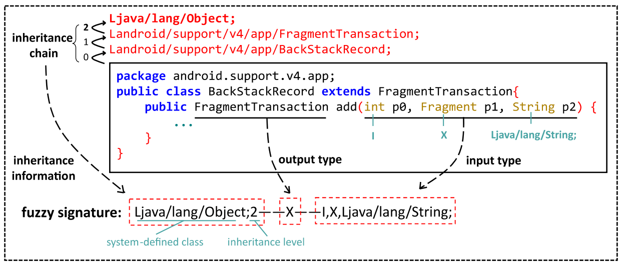


Fig. 2. Workflow of LibSleuth. The workflow consists of two main stages. In the detection stage, it uses a multi-granularity method matching component and a functional module matching component to identify potential libraries. In the selection stage, it determines the final libraries and their versions based on similarity scores while resolving conflicts in method matching.


```
Ljava/lang/Object;
Landroid/support/v4/app/FragmentTransaction;
Landroid/support/v4/app/BackStackRecord;
```

```
package android.support.v4.app;
public class BackStackRecord extends FragmentTransaction{
    public FragmentTransaction add(int p0, Fragment p1, String p2) {
        ...
    }
}
```

Fig. 3. An example of fuzzy method signature

these elements and includes them in the element signatures to enhance their robustness. We refer to this context information obtained from data flow analysis as the usage trace of an element. Finally, an element signature includes the fuzzy description and associated parameters.

During the filtering process, LibSleuth compares all element signatures between the method to be matched and each of its candidate methods. For each element, LibSleuth selects the candidate element with the largest intersection of the associated parameter set among all descriptive information matches as the best match. If a matching element cannot be found, that candidate method is filtered out.

Instruction based filtering. This step performs filtering based on more detailed code semantics. Instructions are the smallest unit in code logic and carry specific behavioral information of methods, making them the most fine-grained features. Unlike regular text matching, code execution is non-linear, and the sequence of instructions may not have a fixed order. Therefore, similar to the approach used in LibScan [18], LibSleuth extracts the entire opcode set of a method as fine-grained method features. However, since LibSleuth focuses on method level matching, using only the global opcode set of a method is still not sufficient for effective filtering.

To address this, LibSleuth adds opcode sets of basic blocks as local information, enhancing the robustness of instruction features through the combination of global and local views. In addition, to address the impact of control flow obfuscation such as splitting or merging basic blocks, LibSleuth extracts the opcode set from each basic block and also gathers opcodes from neighboring basic blocks in the control flow graph, forming a context opcode set. Both types of basic block opcode sets are used for local similarity calculations.

The specific process of this step is as follows: (1) Fuzzifying instruction: Simplify instructions by removing registers, operands, and addresses, retaining only opcodes. (2) Feature extraction: Use the full method opcode set as a global feature and each basic block's opcode set as a local feature. (3) Global similarity calculation: Compute the overlap ratio of complete opcode sets between the target and candidate methods. (4) Local similarity calculation: First, matching is performed using the original opcode sets of basic blocks. If no suitable match is found, the context opcode sets are used to find the best match. A weighted score based on block size and the similarity of the best match represents the local similarity. (5) Final similarity calculation: The overall instruction similarity is calculated by combining global and local similarity scores with weighted averaging. Candidate methods with a similarity score below a predefined threshold are filtered out.

Class consistency based filtering. To obtain a final one-to-one mapping and to avoid cases where methods from the same class in a TPL are matched to different classes in the Android app, LibSleuth performs this final filtering step. Although code shrinking may alter the structure of a library and cause some methods in a class to be missing, the class to which a method belongs does not change. Therefore, LibSleuth uses the ownership relationship between classes and methods to verify all method matching results. It retains only those candidates that meet the following condition: All successfully matched methods from the same class in the TPL must be matched to methods within the same class in the Android app.

*2) **Functional Module Matching**:* The functional module matching component is the design of S1. Its purpose is to accurately determine the matching of libraries in presence of code shrinking. After obtaining the one-to-one method mapping, one common approach is to determine whether a TPL is present by calculating the method matching rate of the library Sim_{TPL}. However, when code shrinking is applied, only a small number of library methods may remain in the app. If we simply lower the matching threshold to reduce false negatives, it can easily lead to false positives. To address this, LibSleuth changes the matching target from the entire library to a code module that ensures complete functionality.

Based on the one-to-one method mapping, LibSleuth treats all successfully matched methods in the TPL as root nodes and constructs a call graph using the dependency relationships among methods. The resulting call graph represents a module that can ensure the functionality of all matched methods is complete. LibSleuth then calculates the method matching rate within this function module Sim_{mod} and uses it to determine library matching in a more flexible way. This approach helps reduce the impact of code shrinking. As shown in Figure 2, if we calculate the method matching rate based on the entire library, it would be 3/6 = 0.5. But within the function module, the matching rate becomes 3/4 = 0.75.

Ultimately, both the method matching rate across the whole library and the matching rate within the function module are used to determine whether a library is matched. The library is considered to be matched, only when $Sim_{\text{lib}} > T_{\text{lib}}$ and $Sim_{\text{mod}} > T_{\text{mod}}$, where T_{lib} is the library similarity threshold and T_{mod} is the functional module similarity threshold.

C. Selection Stage

The goal of the Selection stage is to determine the final matched results from multiple successfully matched library candidates. This stage consists of two components: Version Selection and Conflict Resolution.

*1) **Version Selection**:* When multiple versions of the same library are matched, it is necessary to select the most appropriate version. LibSleuth calculates a library level similarity score by summing the similarity score of all successfully matched methods, as obtained during the instruction based filtering step. The version with the highest overall similarity score is selected as the best match for that library.

*2) **Conflict Resolution**:* In practice, basic methods with similar functionality may exist in different libraries. To prevent assigning the same method to multiple libraries, LibSleuth introduces the uniquely matched method: A method is considered uniquely matched if it forms an exclusive mapping between a library and an app method. LibSleuth adopts a greedy algorithm to determine library matches in a step-by-step manner. At each step, given the current set of matched libraries, LibSleuth selects the next library whose uniquely matched method ratio is the highest and exceeds a predefined threshold. This continues until all matches are finalized.

TABLE I
CATEGORIZATION AND STATISTICS OF DATASETS

Dataset	Category		apps	TP	TPLs
AS1	Non-obfs		225	1656	454
	Proguard		169	997	
	Allatori		210	1532	
	DashO	cfr	88	491	
		fp-ir	88	491	
		dcr	88	491	
		cfr-fp-ir-dcr	88	491	
	Total		1049	6866	
AS2	D8		51	323	146
	R8	shrink	51	323	
		shrink-orlis	51	323	
		shrink-opt	51	323	
	Total		204	1292	
VDD	Closed Source		10000	None	35
	Open Source		231	325	134

IV. Experiments

To comprehensively evaluate the effectiveness and applicability of LibSleuth, we designed the following questions:

- RQ1: How effective is LibSleuth in detecting TPLs under code obfuscation scenarios?
- RQ2: How effective is LibSleuth in detecting TPLs under code shrinking scenarios?
- RQ3: How does LibSleuth compare to commercial tools?
- RQ4: What is the contribution of each component of LibSleuth to its overall performance?
- RQ5: Can LibSleuth effectively identify TPLs with known N-day vulnerabilities in real-world apps?

A. Experimental Setup

*1) **Environment and Implementation**:* Experiments were run on Ubuntu 22.04.3 with AMD EPYC 7543 (128 CPUs). The pipeline of feature extraction and detection was developed in Python. TPLs were converted to DEX via D8 [36] from JAR or AAR, and analyzed statically using Androguard-4.1.2 [38].

*2) **Datasets and Metrics**:* For evaluating detection effectiveness, we reused two benchmark datasets, AS1 and AS2, released by LibScan [18]. We found and corrected issues in the datasets, including misnamed files and inaccurate ground truth. Details of the datasets are shown in Table I.

- AS1 is used to evaluate the tool's resistance to obfuscation. It is composed of datasets from Orlis [31] and Atvhunter [17], including a total of 1,049 Android apps. This dataset contains original apps without obfuscation, as well as apps obfuscated using ProGuard, Allatori, and DashO. In DashO obfuscated data, there are four types of obfuscation configurations: control-flow randomization (cfr), package flattening + identifier renaming (pf-ir), dead code removal (dcr) and all three options enabled (cfr-pf-ir-dcr). Among them, 110 apps were randomly selected for threshold tuning, while the remaining 939 apps were used for evaluation. After our updates, AS1 covers 454 unique TPLs, with 6,866 app-TPL mapping relationships in total.
- AS2 is used to evaluate the tool's resistance to optimization. It contains 204 Android apps, compiled using the D8 compiler and R8 compiler in Android Studio v3.5.1. R8 was configured with three optimization strategies: code shrinking only, code shrinking with Orlis-configuration [31] obfuscation, and code shrinking with code optimization. After updates, AS2 includes 146 unique TPLs, with a total of 1,292 dependency mappings.
- To validate LibSleuth's practical utility in identifying vulnerable TPLs, we built two distinct real world vulnerability detection datasets (VDD).
  Closed Source VDD. We selected the latest 10,000 Android apps (as of February 20, 2025), sourced from Google Play Store [39], AppChina [42], or F-Droid [15] in the AndroZoo dataset [20]. Vulnerable TPLs were selected based on CVEs from the OSV database [40]. We then selected libraries from the top 100 most used Maven libraries that have known vulnerabilities. Based on usage frequency of specific versions, we manually selected 21 vulnerable versions from 11 libraries, corresponding to 77 CVEs directly. Additionally, 14 non vulnerable library versions were included to ensure tool to differentiate between vulnerable and benign version.
  Open Source VDD. To provide a evaluation with verifiable ground truth for both precision and recall, we constructed a dataset from all open source and Gradle-based projects available on F-Droid. For each project, we obtained both its source code and compiled APK. By parsing $build.gradle$ files, we extracted declared direct dependencies and resolved indirect dependencies. Combining this data with the OSV vulnerability database allowed us to build a ground truth for vulnerable library dependencies within these projects. We focused on the 40 most frequently used TPLs (134 versions) and selected 231 projects that depend on these TPL versions.

Detection performance was measured using precision, recall, and F1-score:

$$Precision = \frac{TP}{TP+FP} \qquad Recall = \frac{TP}{TP+FN}$$
$$F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

For library level detection, TP (True Positive) means a TPL present in the app and correctly identified by the tool. FP (False Positive) means a TPL identified by the tool but not present. FN (False Negative) means a TPL that exists in the app but was not detected by the tool. The same evaluation logic applies to version level detection. Notably, some tools may report multiple versions of a library with equal similarity. Following common practice, we consider the detection a true positive if the correct version is included in the results; otherwise, all detected versions are counted as false positives.

TABLE II
TUNED THRESHOLDS

Dataset	LibSleuth		LibPecker		LibLoom		LibScan		LibHunter	
	T_{lib}	T_{mod}	T_{lib}	T_{pkg}	T_{pkg}	T_{lib}	T_{cls}	T_{lib}	T_{m}	T_{sim}
AS1	0.7	0.6	0.7	0.5	0.05	0.9	0.7	0.85	0.95	0.1
AS2	0.25	0.55	0.05	0.1	0.85	0.2	0.05	0.5	0.7	0.2

*3) **Thresholds Tuning**:* LibSleuth relies on two thresholds to identify TPLs: the method match ratio of the library Sim_{lib} and that of the functional module Sim_{mod}. A lower library threshold T_{lib} increases the risk of misidentifying randomly matched method clusters as potential matches. Raising the functional module threshold T_{mod} can help reduce false positives. However, setting it too high may cause false negatives.

Thresholds were tuned via grid search using F1-score. The tuning was conducted on two benchmark datasets respectively. For AS1, we selected 110 apps listed by LibScan for tuning. For AS2, we randomly selected 20 apps (10% of the dataset) for tuning. For fair comparison, we also fine-tuned several academic state-of-the-art open source tools on the same datasets. Specifically, we included LibPecker [32], LibLoom [19], LibScan [18], and LibHunter [11]. The results are summarized in Table II.

B. Effectiveness of LibSleuth

To evaluate the effectiveness of LibSleuth in detecting TPLs and its robustness against code obfuscation and optimization techniques, we conducted comprehensive experiments on the AS1 and AS2 datasets. The evaluation was performed at both the library level and the version level, and the results were compared with several academic state-of-the-art tools.

*1) **Robustness against Obfuscation**:* We tested LibSleuth on the AS1 dataset with apps obfuscated by various tools and configurations. Table III shows the detection results grouped by obfuscation tool. The results clearly show that LibSleuth consistently achieves high detection accuracy at both the library and version levels, outperforming all baseline tools in terms of F1-score. Notably, compared to the non-obfuscated data, LibSleuth shows minimal degradation, indicating strong resilience to code obfuscation. Furthermore, its performance remains stable across both library and version granularities, suggesting its ability to accurately distinguish between different versions of the same library. Among the obfuscation tools, Allatori and DashO have a more significant negative impact on detection performance than ProGuard, likely due to their use of more advanced techniques such as block splitting. The performance of baseline tools varies depending on their feature selection and matching strategies. For example,

TABLE III
COMPARISON OF DETECTION PERFORMANCE ACROSS DIFFERENT OBFUSCATION TOOLS

Detection level	Obfuscation tool	LibSleuth			LibPecker			LibLoom			LibScan			LibHunter		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Library	Non-obfs	**0.9954**	0.9921	**0.9937**	0.8042	**0.9967**	0.8902	0.9914	0.9854	0.9884	0.9696	0.9901	0.9797	0.9088	0.9954	0.9501
	Proguard	0.9922	0.9944	**0.9933**	0.7778	0.9966	0.8737	**0.9966**	0.9899	0.9932	0.9811	0.9888	0.9849	0.9498	**0.9978**	0.9732
	Allatori	**0.9920**	**0.9820**	**0.9870**	0.7529	0.7828	0.7676	0.9758	0.8972	0.9349	0.9700	0.9295	0.9493	0.9354	0.8016	0.8633
	DashO	**0.9953**	**0.9714**	**0.9832**	0.6831	0.5934	0.6351	0.9927	0.9209	0.9555	0.9856	0.9503	0.9676	0.9426	0.7458	0.8327
	Avg.	**0.9937**	**0.9850**	**0.9893**	0.7545	0.8424	0.7917	0.9891	0.9484	0.9680	0.9766	0.9647	0.9704	0.9342	0.8852	0.9048
Version	Non-obfs	**0.994**	**0.9907**	**0.9923**	0.776	0.9716	0.8629	0.9834	0.9782	0.9808	0.9565	0.9894	0.9727	0.8811	0.9854	0.9303
	Proguard	0.9899	**0.9922**	**0.9910**	0.7481	0.9720	0.8455	**0.9910**	0.9843	0.9876	0.9628	0.9877	0.9751	0.9194	0.9843	0.9507
	Allatori	**0.9755**	**0.9769**	**0.9762**	0.6491	0.7282	0.7107	0.9571	0.8814	0.9177	0.9527	0.9267	0.9395	0.7853	0.6966	0.7383
	DashO	**0.9914**	**0.9676**	**0.9794**	0.6202	0.5396	0.5771	0.9887	0.9175	0.9518	0.9581	0.9238	0.9406	0.8700	0.6932	0.7716
	Avg.	**0.9877**	**0.9819**	**0.9847**	0.7096	0.8029	0.7491	0.9801	0.9404	0.9595	0.9575	0.9569	0.9570	0.8640	0.8399	0.8477

TABLE IV
COMPARISON OF DETECTION PERFORMANCE ACROSS DIFFERENT DASHO OBFUSCATION TYPES

Detection level	Obfuscation type	LibSleuth			LibPecker			LibLoom			LibScan			LibHunter		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Library	Non-obfs	**0.9954**	0.9921	**0.9937**	0.8042	**0.9967**	0.8902	0.9914	0.9854	0.9884	0.9696	0.9901	0.9797	0.9088	0.9954	0.9501
	cfr	0.9954	**0.9954**	**0.9954**	0.7332	0.7986	0.7645	**0.9977**	0.9931	0.9954	0.9907	0.9771	0.9839	0.9694	0.9428	0.9559
	pf-ir	0.9954	**0.9977**	**0.9965**	0.7127	0.7208	0.7167	**1.0000**	0.881	0.9367	0.9907	0.9771	0.9839	0.9751	0.9840	0.9795
	dcr	0.9925	0.9130	0.9511	0.713	0.7162	0.7146	**1.0000**	0.9085	0.9521	0.9974	0.8650	0.9265	0.9703	**0.9703**	**0.9703**
	cfr-pf-ir-dcr	**0.9962**	**0.9746**	**0.9853**	0.6124	0.4066	0.4887	0.9848	0.9127	0.9474	0.9771	0.9634	0.9702	0.8769	0.4751	0.6163
Version	Non-obfs	**0.9940**	**0.9907**	**0.9923**	0.7760	0.9716	0.8629	0.9834	0.9782	0.9808	0.9565	0.9894	0.9727	0.8811	0.9854	0.9303
	cfr	**0.9931**	**0.9931**	**0.9931**	0.6485	0.7094	0.6776	0.9931	0.9888	0.9908	0.9606	0.9474	0.9540	0.9038	0.8810	0.8923
	pf-ir	**0.9954**	**0.9977**	**0.9965**	0.6267	0.6339	0.6303	0.9948	0.8764	0.9319	0.9906	0.9474	0.9540	0.9095	0.9199	0.9147
	dcr	0.9900	**0.9108**	**0.9488**	0.6264	0.6293	0.6278	**0.9950**	0.9039	0.9473	0.9631	0.8352	0.8946	0.9016	0.9016	0.9016
	cfr-pf-ir-dcr	**0.9894**	**0.9681**	**0.9786**	0.5932	0.3944	0.4738	0.9818	0.9108	0.9450	0.9542	0.9408	0.9475	0.7925	0.4376	0.5639

LibHunter is notably affected by DashO, possibly due to its reliance on string-based features, which are highly susceptible to string encryption. LibPecker achieves the lowest precision, as it only employs coarse-grained features that lack sufficient discriminative capability. In contrast, LibLoom and LibScan demonstrate relatively better resilience under obfuscation.

To further investigate, we analyzed the effects of different DashO obfuscation configurations, as shown in Table IV. Across most configuration types, LibSleuth exhibits the best detection performance. It remains largely unaffected by control flow randomization, package flattening, and identifier renaming. Although dead code elimination reduces recall for all tools in version-level, LibSleuth still yields the fewest false negatives and maintains high precision. Notably, LibLoom shows a significant drop in recall under the "pf-ir" configuration. This may be attributed to its package level matching strategy, which remains fragile to package flattening despite some built-in mitigation measures. Additionally, we observed that combining multiple obfuscation techniques tends to amplify their impact on LibPecker and LibHunter. Nevertheless, LibSleuth remains the most stable and accurate among all baselines.

Answering RQ1: LibSleuth is highly effective at detecting TPLs under code obfuscation, outperforming other tools with minimal performance drop. Its fine-grained, context-aware matching makes it resilient to various obfuscation techniques and capable of distinguishing library versions accurately.

*2) **Robustness against Optimization**:* To evaluate the robustness of LibSleuth under code optimization, we conducted experiments on the AS2 dataset. Table V presents the detection results of LibSleuth and baseline tools across different optimization levels. LibSleuth shows stronger resilience against code optimization than other tools, especially when facing code shrinking. Similar to the DashO-dcr results observed in AS1, all tools show increased false negatives after code shrinking. Except for LibSleuth and LibHunter, most tools experience a substantial drop in recall. Notably, LibSleuth maintains high precision even while achieving better recall, which we attribute to its functional module matching strategy. For cases involving both code shrinking and obfuscation, the performance of most tools is similar to their performance under code shrinking alone, indicating that many tools have already developed resilience against common obfuscation.

Interestingly, unlike LibHunter, LibSleuth does not implement mitigation strategies specifically tailored for optimization. Nevertheless, it achieves strong detection performance in combined code shrinking and optimization scenarios. This underscores the need for stable, robust features resilient to unknown transformations. When comparing detection performance at the library and version levels, we observe that all baseline tools suffer significant degradation at the version level. This is primarily due to reduced feature availability caused by code shrinking, which makes distinguishing between library versions more challenging. In contrast, LibSleuth exhibits a clear advantage at the version level. This demonstrates that its method level feature matching, along with context-enhanced characteristics, can effectively capture fine-grained differences required for version level distinction.

Answering RQ2: LibSleuth also performs well under code shrinking and optimization, maintaining high precision and recall. Its robust feature matching allows it to detect libraries reliably, even when code has been shrunk.

C. Comparison with Commercial SCA Tools

To benchmark LibSleuth against industry solutions, we evaluated four commercial binary-level SCA tools: AppSweep (Guardsquare) [56], Scantist SCA [55], T-Sec BSCA (Ten-

TABLE V
COMPARISON OF DETECTION PERFORMANCE ACROSS DIFFERENT OPTIMIZATION TYPES

Detection level	Optimization type	LibSleuth			LibPecker			LibLoom			LibScan			LibHunter		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Library	D8	**0.8862**	0.9863	**0.9336**	0.8567	0.9829	0.9155	0.8814	0.9418	0.9106	0.7838	**0.9932**	0.8762	0.7721	0.9863	0.8662
	R8-shrink	0.9883	0.8699	**0.9253**	0.9315	0.6886	0.7984	**0.9951**	0.6986	0.8209	0.9405	0.8116	0.8713	0.9444	**0.8733**	0.9075
	R8-shrink-orlis	0.9845	0.8699	**0.9237**	0.8675	0.6952	0.7719	**0.9946**	0.6267	0.7689	0.9402	0.8082	0.8692	0.9444	**0.8733**	0.9075
	R8-shrink-opt	**1.0000**	0.6062	0.7548	0.9938	0.5446	0.7036	**1.0000**	0.3493	0.5177	**1.0000**	0.2637	0.4173	0.9636	**0.8151**	**0.8832**
	Avg.	0.9648	0.8331	0.8844	0.9124	0.7303	0.7974	**0.9678**	0.6541	0.7545	0.9161	0.7192	0.7585	0.9061	**0.8870**	**0.8911**
Version	D8	0.7908	0.8801	0.8331	0.7840	**0.9075**	0.8412	**0.8386**	**0.9075**	**0.8717**	0.5488	0.7705	0.6410	0.5990	0.8185	0.6918
	R8-shrink	**0.8571**	**0.7603**	**0.8058**	0.4023	0.3527	0.3759	0.8119	0.6062	0.6941	0.5160	0.4966	0.5061	0.6085	0.5856	0.5968
	R8-shrink-orlis	**0.8500**	**0.7568**	**0.8007**	0.3711	0.3235	0.3467	0.7805	0.5479	0.6438	0.5179	0.4966	0.5077	0.6064	0.5856	0.5958
	R8-shrink-opt	**0.7688**	**0.4897**	**0.5983**	0.3280	0.2123	0.2578	0.6667	0.2671	0.3814	0.5765	0.1678	0.2599	0.4982	0.4760	0.4868
	Avg.	**0.8167**	**0.7217**	**0.7595**	0.4714	0.4495	0.4554	0.7744	0.5822	0.6478	0.5398	0.4829	0.4785	0.5780	0.6164	0.5928

TABLE VI
ABLATION EXPERIMENT ON AS2

Variants	Precision	Recall	F1	Cost time(s)
$LibSleuth$	**0.8167**	0.7217	**0.7595**	**1346**
$LibSleuth_{\text{sig}}$	0.8049	0.6764	0.727	2464
$LibSleuth_{\text{ele}}$	0.7784	**0.7363**	0.7546	1395
$LibSleuth_{\text{ins}}$	0.7659	0.6781	0.7122	1504
$LibSleuth_{\text{cls}}$	0.5785	0.5309	0.5516	1548
$LibSleuth_{\text{mod}}$	0.7753	0.7106	0.7381	1561

TABLE VII
PERFORMANCE OF COMMERCIAL SCA TOOLS

Library metrics	LibSleuth	AppSweep	Scantist	BSCA	CodeArts
Precision	0.992	0.492	0.371	0.667	0.517
Recall	0.904	0.637	0.089	0.055	0.116

cent) [54], and CodeArts Governance (Huawei) [53]. We tested 30 APKs randomly sampled from the AS1 and AS2 datasets, covering both original and transformed variants. Since each tool uses its own TPL database, we assessed their accuracy based on known ground truth labels. Results are shown in Table VII. The analysis shows that LibSleuth consistently has the best performance across all scenarios. Among the four commercial tools, only CodeArts supports version level detection, but it rarely reports the correct version. The other tools only provide detection at the library level. Scantist SCA failed to detect most Android TPLs. Overall precision across all tools is low. BSCA achieves slightly higher precision, but its detection is limited to library $com.google.android.support$. Due to differences in each tool's internal TPL dataset, recall metrics are not directly comparable. However, across different variants, we observe that all tools show a degree of resilience to code obfuscation, with AppSweep performing the best in this regard. In contrast, all tools experience a sharp performance drop when facing optimization, especially code shrinking. In particular, CodeArts fails completely under this condition.

Answering RQ3: LibSleuth outperforms all evaluated commercial SCA tools and is the only solution that support version level detection. While existing tools show partial resilience to obfuscation, none of them handle code shrinking effectively.

D. Ablation Study

To evaluate the contribution of each component within LibSleuth, we selectively removed or modified specific components and analyzed the impact on accuracy and runtime. Our design focuses on the method matching component and functional module matching component in the detection stage. We designed several LibSleuth variants:

- $LibSleuth_{\text{sig}}$: In the description based filtering step, class inheritance information is excluded from signatures.
- $LibSleuth_{\text{ele}}$: The semantic element filter is removed.
- $LibSleuth_{\text{ins}}$: In the instruction based filtering step, only global instruction features are used without local features.
- $LibSleuth_{\text{cls}}$: The class consistency based filtering step is removed. Method pairs are selected by instruction similarity.
- $LibSleuth_{\text{mod}}$: The functional module matching stage is removed, and detection relies on the method match ratio of library level.

We evaluated these variants on the AS2 dataset, and the results are summarized in Table VI. Overall, all variants exhibited lower F1-scores and higher runtime compared to the original version of LibSleuth. In $LibSleuth_{\text{sig}}$, both detection performance and efficiency significantly declined, highlighting the importance of class inheritance information in enabling stable and efficient large-scale filtering. Although $LibSleuth_{\text{ele}}$ slightly improved recall, it caused a drop in precision and led to increased computational cost due to a larger number of candidates progressing to subsequent matching steps. $LibSleuth_{\text{ins}}$ showed reduced performance, as the lack of fine-grained information hindered accurate method mapping and weakened functional module matching. Among all variants, $LibSleuth_{\text{cls}}$ showed the most significant performance degradation. This highlights the critical role of maintaining the ownership relationship between classes and methods in avoiding incorrect method pairings. Finally, $LibSleuth_{\text{mod}}$ resulted in a higher number of false positives and false negatives, confirming the effectiveness of functional module matching.

Answering RQ4: Each component of LibSleuth contributes to improving detection performance and efficiency, demonstrating the soundness and effectiveness of the system design.

E. N-day Vulnerability Detection

To assess LibSleuth's practical applicability in real-world scenarios, we applied it to N-day vulnerability detection.

TABLE VIII
DETECTION PERFORMANCE ON OPEN SOURCE VDD

Detection level	metrics	LibSleuth	LibScan	LibHunter
Library	Precision	**0.922**	0.400	0.825
	Recall	**0.837**	0.772	0.811
	F1	**0.877**	0.527	0.818
Version	Precision	**0.866**	0.354	0.721
	Recall	**0.794**	0.687	0.723
	F1	**0.829**	0.467	0.722

Closed source VDD. We analyzed 10,000 recent apps from AndroZoo to detect vulnerable libraries. The results indicate that 20.35% of the apps were flagged as using vulnerable libraries. A total of 2517 vulnerable versions were detected. Among these, $gson$, $commons-io$, and $httpclient$ were the most frequently detected vulnerable libraries. For validation, we randomly sampled 500 flagged apps for manual verification. Ultimately, we identified 84 false positives out of the 609 detected versions. Estimated precision exceeds 86%, showing strong real-world reliability. Additionally, 9.93% of flagged apps were also tagged as malware in AndroZoo, suggesting a potential correlation between vulnerable libraries and app risk.

Open source VDD. To enable a fully controlled evaluation with ground-truth labels for both precision and recall, we conducted experiments on the Open source VDD, which includes 231 projects and 134 known vulnerable TPL versions. We compared LibSleuth against two academic baselines, LibScan and LibHunter. As shown in Table VIII, LibSleuth achieved the highest precision and recall at the version level.

Answering RQ5: LibSleuth can effectively identify N-day vulnerabilities in both closed source and open source real-world scenarios. In the closed source setting, it achieves an estimated precision above 86%. In the open source setting, it outperforms existing tools in both precision and recall.

V. DISCUSSION

A. Limitations and Future Work

LibSleuth has several limitations that need further attention. False negatives often result from inaccurate method matching, impacting functional module construction. Though robust features are used, they may weaken as optimization evolves. False positives mainly arise from small libraries that are hard to distinguish, and from libraries with similar code segments. Currently, LibSleuth lacks defenses against optimizations like inlining, reflection, and parameter removal. Future work includes exploring finer-grained resilient features and learning-based matching [43], [44]. Recent research [47]–[50] shows the potential of deep learning in function level code clone detection. Compared to rule-based methods, learning-based approaches may offer greater robustness against unknown or evolving obfuscation techniques.

B. Threats to Validity

Ground truth issues. When updating dataset AS2, we only verified the original apps. After code shrinking, the ground truth might have changed. However, due to the absence of mapping files, we could not update the ground truth accordingly, which may affect the accuracy of results. A mitigation strategy is to construct a new benchmark by building open source projects with R8 and preserving the mapping files.

Threshold configuration. Tuned thresholds may not generalize to real-world data with different distributions. Future work could explore adaptive thresholding methods that automatically adjust based on contextual code features, reducing the need for manual tuning and improving robustness.

VI. RELATED WORK

Several Android TPL detection works have been proposed.

Academic Tools. LibPecker [32] constructs class dependency signatures and employs adaptive similarity thresholds to tolerate customized or incomplete code, but is sensitive to package hierarchy changes. LibID [33] uses class dependency graphs and generates features via locality-sensitive hashing for matching. ATVHunter [17] adopts a coarse-to-fine detection strategy using control flow and opcode features, combined with fuzzy hashing, to improve detection under obfuscation. LibLoom [19] formulates TPL detection as a set containment problem and proposes a two-stage Bloom filtering approach and an entropy-based similarity metric to address challenges such as package flattening. LibScan [18] performs multi-level matching based on class signatures, method opcodes, and call chain opcodes. It maintains high accuracy under obfuscation techniques, but suffers under aggressive code shrinking. LibHunter [11] addresses the impact of compiler optimizations such as inlining and callsite optimization. It combines class signature matching, opcode-based method matching, and cross-inlined method analysis, but it does not fully address advanced obfuscation techniques like string encryption. LibAttention [51] pre-trains a Transformer on Smali, compresses class-level structure and type tokens into embeddings, and fine-tunes for TPL presence. It targets R8 but yields limited to library-level granularity. LibMD [52] extracts dependencies across class, function, and basic block scales, combining class reference syntax augmentation, cross-scale function mapping, and control flow reconstruction of basic blocks. However, it omits code optimization issues. Unlike tools relying on class/package features, LibSleuth uses method level matching and functional modules for greater resilience and flexibility. In addition, in the C/C++ domain, ModX [34] also performs detection at the module level. It employs a modular approach that aggregates functionalities into semantically-based modules. However, ModX requires pre-modularized of libraries, which limits its applicability to real-world, customized usage.

Commercial Tools. AppSweep (Guardsquare) [56] is a commercial mobile app security tool developed by the creators of ProGuard. Scantist SCA [55], derived from NTU's lab, provides SCA services. T-Sec BSCA [54] is Tencent's binary analysis tool supported by Keen Security Lab [58]. CodeArts Governance [53] is Huawei's open source management platform. We also applied for BlackDuck SCA [57] access but were unable to obtain research licensing.

VII. Conclusion

We developed LibSleuth, an Android TPL detector resilient to code shrinking and robust against code obfuscation. LibSleuth identifies TPLs by performing method level feature matching and leveraging the completeness of the functional module. To achieve accurate and efficient method matching, it adopts a multi-step filtering strategy and context-enhanced features. Testing shows LibSleuth handles obfuscation and shrinking effectively, outperforming academic and commercial tools at version-level detection. Furthermore, in real-world scenarios, we successfully identified the usage of vulnerable library versions. LibSleuth provides development and security teams accurate visibility they need to effectively manage security risks, reducing audit costs.

VIII. Acknowledge

This paper is supported by the the Strategic Priority Research Program of the Chinese Academy of Sciences under No. XDA0320401, the National Natural Science Foundation of China under No. 62202457. This paper is also supported by YuanTu Large Research Infrastructure.

References

[1] 'Open Source Trends - OSSRA Report', 2025. [Online]. Available: https://www.blackduck.com/blog/open-source-trends-ossra-report.html.

[2] 'Report Finds 82% of Open Source Software Components Inherently Risky', 2023. [Online]. Available: https://venturebeat.com/security/report-finds-82-of-open-source-software-components-inherently-risky.

[3] 'Maven Repository', 2025. [Online]. Available: https://mvnrepository.com.

[4] C. J. Li, H. Yu, K. Chen, X. J. Zhao, Y. Han, and A. P. Li, "A survey of IoT threat intelligence knowledge graph," Journal of Cybersecurity, vol. 2, no. 2, pp. 18–35, 2024. [Online]. Available: https://doi.org/10.20172/j.issn.2097-3136.240202

[5] 'Apache Maven', 2025. [Online]. Available: https://maven.apache.org.

[6] 'Gradle', 2025. [Online]. Available: https://gradle.org.

[7] 'Apache Log4j 2.x Documentation', 2025. [Online]. Available: https://logging.apache.org/log4j/2.x/index.html.

[8] 'Understanding the Impact of Apache Log4j', 2021. [Online]. Available: https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html.

[9] S. Dong et al., 'Understanding android obfuscation techniques: A large-scale investigation in the wild', in Security and privacy in communication networks: 14th international conference, secureComm 2018, Singapore, Singapore, August 8-10, 2018, proceedings, part i, 2018, pp. 172–192.

[10] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, 'A large scale investigation of obfuscation use in google play', in Proceedings of the 34th annual computer security applications conference, 2018, pp. 222–235.

[11] Z. Xie, M. Wen, T. Li, Y. Zhu, Q. Hou, and H. Jin, 'How Does Code Optimization Impact Third-party Library Detection for Android Applications?', in Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 1919–1931.

[12] G. You, G. Kim, S.-J. Cho, and H. Han, 'A Comparative Study on Optimization, Obfuscation, and Deobfuscation tools in Android', J. Internet Serv. Inf. Secur., vol. 11, no. 1, pp. 2–15, 2021.

[13] L. Zhao et al., 'Software composition analysis for vulnerability detection: An empirical study on Java projects', in Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 960–972.

[14] K. Ren, K. Yang, H. T. Shen, F. Lin, and D. K. Shen, "A survey of cybersecurity for intelligent connected vehicles," Journal of Cybersecurity, vol. 2, no. 6, pp. 16–35, 2024. [Online]. Available: https://doi.org/10.20172/j.issn.2097-3136.240602

[15] 'F-Droid - Free and Open Source Android App Repository', 2025. [Online]. Available: https://f-droid.org.

[16] 'Shrink, Obfuscate, and Optimize Your App', 2025. [Online]. Available: https://developer.android.com/build/shrink-code.

[17] X. Zhan et al., 'Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications', in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 1695–1707.

[18] Y. Wu, C. Sun, D. Zeng, G. Tan, S. Ma, and P. Wang, 'LibScan: Towards more precise Third-Party library identification for android applications', in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 3385–3402.

[19] J. Huang et al., 'Scalably detecting third-party android libraries with two-stage bloom filtering', IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 2272–2284, 2022.

[20] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, 'AndroZoo: Collecting Millions of Android Apps for the Research Community', in Proceedings of the 13th International Conference on Mining Software Repositories, Austin, Texas, 2016, pp. 468–471.

[21] Z. Y. Sun, J. Z. Wu, X. Ling, Y. L. Wei, T. Y. Luo, and Y. J. Wu, "Research on key technologies of SBOM in software supply chain," Ruan Jian Xue Bao/Journal of Software, vol. 36, no. 6, pp. 2604–2642, 2025 (in Chinese). [Online]. Available: http://www.jos.org.cn/1000-9825/7308.htm

[22] X. Cui et al., "An empirical study of license conflict in free and open source software," in 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), IEEE, 2023, pp. 495–505.

[23] M. Hammad, J. Garcia, and S. Malek, 'A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products', in Proceedings of the 40th international conference on software engineering, 2018, pp. 421–431.

[24] 'ssdeep - Fuzzy Hashing Program', 2018. [Online]. Available: https://ssdeep-project.github.io/ssdeep/index.html.

[25] B. H. Bloom, 'Space/time trade-offs in hash coding with allowable errors', Communications of the ACM, vol. 13, no. 7, pp. 422–426, 1970.

[26] R. Levinson and G. Ellis, 'Multilevel hierarchical retrieval', Knowledge-Based Systems, vol. 5, no. 3, pp. 233–244, 1992.

[27] 'Gradle Build Overview', 2025. [Online]. Available: https://developer.android.com/build/gradle-build-overview.

[28] 'ProGuard', 2025. [Online]. Available: https://www.guardsquare.com/proguard.

[29] 'Allatori Java Obfuscator', 2025. [Online]. Available: https://www.allatori.com.

[30] 'DashO Java Obfuscator', 2025. [Online]. Available: https://www.preemptive.com/products/dasho/overview.

[31] Y. Wang, H. Wu, H. Zhang, and A. Rountev, 'Orlis: Obfuscation-resilient library detection for android', in Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, 2018, pp. 13–23.

[32] Y. Zhang et al., 'Detecting third-party libraries in android applications with high precision and recall', in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 141–152.

[33] J. Zhang, A. R. Beresford, and S. A. Kollmann, 'Libid: reliable identification of obfuscated third-party android libraries', in Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, pp. 55–65.

[34] C. Yang, Z. Xu, H. Chen, Y. Liu, X. Gong, and B. Liu, 'ModX: binary level partially imported third-party library detection via program modularization and semantic matching', in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1393–1405.

[35] 'ProGuard Manual: Optimizations', 2025. [Online]. Available: https://www.guardsquare.com/manual/configuration/optimizations.

[36] 'D8 - Android's Dex Compiler', 2025. [Online]. Available: https://developer.android.com/tools/d8.

[37] X. Ling et al., "A wolf in sheep's clothing: Practical black-box adversarial attacks for evading learning-based windows malware detection in the wild," in 33rd USENIX Security Symposium (USENIX Security 24), 2024, pp. 7393–7410.

[38] 'Androguard - Reverse Engineering and Analysis of Android Applications', 2025. [Online]. Available: https://github.com/androguard/androguard.

[39] 'Google Play Store', 2025. [Online]. Available: https://play.google.com.

[40] 'OSV - Open Source Vulnerability Database', 2025. [Online]. Available: https://osv.dev.
[41] 'Common Vulnerabilities and Exposures (CVE)', 2025. [Online]. Available: https://www.cve.org.
[42] 'AppChina', 2025. [Online]. Available: http://m.appchina.com.
[43] S. Feng, W. Suo, Y. Wu, D. Zou, Y. Liu, and H. Jin, 'Machine learning is all you need: A simple token-based approach for effective code clone detection', in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.
[44] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, 'How machine learning is solving the binary function similarity problem', in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 2099–2116.
[45] S. Almanee, A. Ünal, M. Payer, and J. Garcia, 'Too quiet in the library: An empirical study of security updates in android apps' native code', in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 1347–1359.
[46] X. Zhan et al., 'Research on third-party libraries in android apps: A taxonomy and systematic literature review', IEEE Transactions on Software Engineering, vol. 48, no. 10, pp. 4181–4213, 2021.
[47] S. H. H. Ding, B. C. M. Fung, and P. Charland, 'Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization', in 2019 ieee symposium on security and privacy (sp), 2019, pp. 472–489.
[48] H. Wang et al., 'Jtrans: Jump-aware transformer for binary code similarity detection', in Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 1–13.
[49] Y. Zhang, Y. Liu, G. Cheng, and B. Ou, 'GTrans: Graph Transformer-Based Obfuscation-resilient Binary Code Similarity Detection'.
[50] L. Jiang et al., 'Binaryai: binary software composition analysis via intelligent binary source code matching', in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.
[51] D. Pan et al., 'Pay Your Attention on Lib! Android Third-Party Library Detection via Feature Language Model', in 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2025, pp. 217–228.
[52] Z. Zhang, S. Luo, Y. Lu, and L. Pan, 'Obfuscation-resilient detection of Android third-party libraries using multi-scale code dependency fusion', Information Fusion, vol. 117, p. 102908, 2025.
[53] 'Huawei Cloud CodeArts Governance', Huawei Cloud, 2025. [Online]. Available: https://www.huaweicloud.com/intl/en-us/product/codeartsgovernance.html.
[54] 'BSCA Open API", Tencent Cloud, 2025. [Online]. Available: https://cloud.tencent.com/product/bsca.
[55] 'SCA – Software Composition Analysis', Scantist, 2025. [Online]. Available: https://scantist.com/products/sca.
[56] 'AppSweep Mobile Application Security Testing', Guardsquare, 2025. [Online]. Available: https://www.guardsquare.com/appsweep-mobile-application-security-testing.
[57] 'Software Composition Analysis Tools', Black Duck, 2025. [Online]. Available: https://www.blackduck.com/software-composition-analysis-tools.html.
[58] 'KeenLab – Tencent Security', KeenLab, Tencent, 2025. [Online]. Available: https://keenlab.tencent.com/zh/index.html.