Quokka

ACADEMIC PAPER

LIBRA

Library Identification
in Obfuscated
Android Apps

Libra: Library Identification in Obfuscated Android Apps

David A. Tomassi$^{1[0009-0007-8434-7122]}$, Kenechukwu Nwodo$^{2[0009-0000-7208-909X]\star}$, and Mohamed Elsabagh$^{1[0000-0002-5320-4985]}$

^{1} Quokka, McLean, VA, USA
{dtomassi, melsabagh}@quokka.io
^{2} Virginia Tech, Arlington, VA, USA
nwodok@vt.edu

Abstract. In the Android apps ecosystem, third-party libraries play a crucial role in providing common services and features. However, these libraries introduce complex dependencies that can impact stability, performance, and security. Therefore, detecting libraries used in Android apps is critical for understanding functionality, compliance, and security risks. Existing library identification approaches face challenges when obfuscation is applied to apps, leading to performance degradation. In this study, we propose *Libra*, a novel solution for library identification in obfuscated Android apps. *Libra* leverages method headers and bodies, encodes instructions compactly, and employs piecewise fuzzy hashing for effective detection of libraries in obfuscated apps. Our two-phase approach achieves high F1 scores of 88% for non-obfuscated and $50-87\%$ for obfuscated apps, surpassing previous works by significant margins. Extensive evaluations demonstrate *Libra*'s effectiveness and robustness against various obfuscation techniques.

Keywords: Library Identification · Obfuscation · SBOM · Android

1 Introduction

With three billion active devices, Android has become the dominant mobile platform with over three and a half million apps on the Google Play Store alone [10, 11]. These apps cater to the needs of billions of users in an ever-evolving landscape. To accelerate development and enhance user experience, apps often incorporate various third-party libraries to leverage their prebuilt functionalities [35, 44]. While these third-party libraries offer considerable development advantages, they introduce complex dependencies into the apps that can significantly impact stability, performance, and security.

Detecting libraries used in Android apps has become a critical pursuit for developers, security analysts, and researchers alike [28]. Identifying the libraries

* This work was done as part of an internship at Quokka.

that make up an app allows for a deeper understanding of the app's functionality, licensing compliance, and potential security risks. Moreover, tracking these dependencies aids in the timely integration of updates, ensuring the apps stay current with the latest feature enhancements and security patches.[3]

Various library identification approaches were introduced in the recent years, including clustering techniques [24, 26, 29, 42], learning-based techniques [25, 27], and similarity-based techniques [15, 19–21, 31, 34, 36, 37, 41, 43]. A variety of different app and library features are used by these approaches, ranging from package and class hierarchies to GUI resources and layout files. These techniques operate with the same end goal, that is to identify the libraries (names and versions) used by an app given the app published binary package.

Most of these tools have been developed with obfuscation in mind and select features that have resiliency to obfuscation techniques. Yet, it has been shown [38, 39, 42] that when obfuscation is applied to apps the performance of the state-of-the-art tools degrades significantly. Obfuscation is not a new reality for software and has been used to hide malicious software such as the SolarWinds attack [8] where the attackers used multiple obfuscation layers to hide the malicious software from detection. This exemplifies the need for identification techniques with increased resilience to the various obfuscation techniques that can be applied to Android apps, including identifier renaming, code shrinking and removal, control-flow randomization, package flattening, among others.

To this end, we propose *Libra* in this study as a novel solution to identify library names and versions in an Android app package with higher resilience to obfuscation than the state of the art. By examining the current state-of-the-art techniques, we shed light on some of the overlooked challenges that arise when analyzing obfuscated apps and discuss how *Libra* tackles these challenges to achieve higher detection power than the state of the art.

Libra is designed around three novel ideas: (1) leveraging both method headers and bodies to enhance robustness to obfuscation, (2) encoding method instructions into a compact representation to mitigate learning bias of instruction sequences, and (3) employing piecewise fuzzy hashing for effective adaptation to changes introduced by obfuscators. *Libra* employs a two-phase approach for library identification. In the learning phase, it extracts packages, encodes methods, and generates signatures. In the detection phase, it extracts library candidates, follows the same procedure in learning for method encoding and signature generation, pairs library candidates and actual libraries to shrink the search space, then applies a two-component weighted similarity computation to arrive at a final similarity score between a library candidate and a library.

We performed an extensive evaluation using multiple state-of-the-art Android library identification tools on various obfuscated benchmarks. For each tool we look at its capabilities and its resilience to different obfuscation techniques and highlight how it compares to *Libra*. Our experiments reveal that *Libra* achieves

[3] The process of identifying components used in a software is generally known as creating a Software Bill of Materials (SBOM). See https://www.cisa.gov/sbom for more information about the SBOM concept and standards.

a high F1 score of 88% for non-obfuscated apps, surpassing prior works by a margin ranging from 7% to 540%. For obfuscated apps, *Libra* achieves F1 scores ranging from 50% to 87%, achieving a substantial improvement over previous approaches from 7% and up to 1386% in certain cases.

To summarize, the contributions of this work are:

– We introduce *Libra*, a novel approach to library identification using fuzzy method signatures of hashed instructions.
– We provide a characterization of the state-of-the-art tools and highlight challenges unique to identifying libraries in obfuscated apps.
– We demonstrate the effectiveness of *Libra* by extensively evaluating it against recent Android library identification tools on multiple datasets with various degrees of obfuscation.

2 Background

2.1 Android Third Party Libraries

An Android app is packaged into an Android Package file (APK) which contains the app's Dalvik Executable (DEX) bytecode files, resource files, and assets. The bytecode is organized into package hierarchies, e.g. `com/example`, where each package in the hierarchy may contain one or more implementation units (a class file) and other subpackages. The APK contains both the app's own bytecode as well as the bytecode for all the third-party libraries (and their transitive dependencies) on which the app depends.

Several recent studies have shown that almost all Android apps use third-party libraries [9,35,39,44]. These libraries are used to leverage existing functionalities and enable various services, such as advertisements, maps, and social networks [35,44]. However, despite the widespread usage of libraries in Android apps, concerns have been raised about the security impact of depending on third-party libraries. Multiple studies revealed that apps often use outdated libraries [14,16,35,44]. A recent study [9] of apps on Google Play has shown that 98% used libraries, with an average of 20 libraries per app. Alarmingly, nearly half of the apps used a library that suffered from a high-risk vulnerability.

These vulnerable libraries pose significant challenges for developers and end users. The scope of a vulnerability in a library does not only impact the library, but also extends to its dependencies and other apps and libraries depending on it. Therefore, it is paramount that libraries packaged with an app are identified in order to allow for quick remediation and confinement of potential vulnerabilities.

2.2 Library Detection and Obfuscated Apps

Android app developers use obfuscation techniques to mask the structure, data, and resources of their apps to hinder reverse engineering attempts. Bad actors also use obfuscation to hide malicious code. Obfuscation typically takes place during the build or post-build processes where the obfuscators operate on the bytecode in an APK. Given an obfuscated app APK, the line between what bytecode is app code vs. library code is often blurred due to the various transformations that occur during the obfuscation process.

Android obfuscators such as ProGuard [6], DashO [2], and Allatori [1] are among the most popular and studied obfuscators in the literature. Several studies [15, 28, 33] have analyzed the configurations of these obfuscators and summarized their distinct obfuscation techniques. Pertinent to this study are techniques that apply transformations to the bytecode of an app, such as identifier renaming (transforming package/class/method names into random non-meaningful strings), code addition (adding redundant or bloating code to increase analysis cost), code removal (eliminating unused classes and methods while retaining the functionality of the app), package flattening/repackaging (consolidating multiple packages into one), control flow randomization (shuffling the app's basic blocks while maintaining functionality), among others. Some of these techniques overlap and may be categorized as optimization or code shrinking techniques.

To detect libraries integrated in an app APK, researchers have gone through a number of techniques. Initial efforts to library detection involved utilizing a whitelist of common library package names [29]. However, whitelisting disallows the identification of library versions and does not perform well when obfuscation techniques such as package renaming are applied, leading to low precision and recall. This has led to the development of other detection techniques, such as clustering and similarity comparisons. Clustering techniques [24, 26, 32, 42] pack app packages and library packages together and use a threshold for the cluster size to determine if a cluster of packages can be identified as a library. Clustering can be an exhaustive process, especially given the overwhelming number of library artifacts on the market.[4] To strike a better balance between performance and detection power, techniques based on similarity comparisons were introduced [15, 38] where they identify and compare certain features of library candidates from an app against a prebuilt library features database. *Libra* falls under this category. We discuss related work in more depth in §7.

3 Overview and Key Challenges

3.1 Motiviating Example

By examining the state-of-the-art techniques, we observed that obfuscators that perform code shrinking are particularly difficult to handle. Code shrinking removes unused classes, fields, methods, and instructions that do not have an impact on the functionality of an app. This shrinks the size of libraries in the app and leaves less bytecode to operate on to calculate similarity.

We encountered an instance of this with a library called `com.github.gabriele-mariotti.changeloglib:changelog:2.1.0` within the SensorsSandbox app [13]. A comparison of the package structures between the unobfuscated and obfuscated

[4] At the time of this writing, the Maven Central repository [5] had over 11 million indexed library packages.

```
android
com
it
└── gmariotti
    └── changelibs
        └── library
            ├── internal
            │   ├── 10 classes
            ├── parser
            │   ├── 3 classes
            └── view
                └── 4 classes
```

(a) Before code shrinking.

```
android
com
it
└── gmariotti
    └── changelibs
        └── library
            ├── internal
            │   ├── 7 classes
            ├── parser
            │   ├── 3 classes
            └── view
                └── 2 classes
```

(b) After code shrinking.

Fig. 1. Library package structure for the SensorsSandbox app without (left) and with code shrinking (right). The removal of a few classes causes the library to be missed by all prior solutions examined in this work.

versions of the library is depicted in Fig. 1. Three classes from the library's `internal` subpackage and two classes from the `view` subpackage were removed due to code shrinkage during compilation process. Overall, code shrinking resulted in a decrease of 36.9% in the number of instructions within the library.

Despite the small size of the app, the fact it had only this third-party library dependency, and the low degree of shrinking, all recent tools failed at identifying the library in the app APK when code shrinking was applied. Specifically, LibScout [15], ATVHunter [37], and Libloom [21], were able to detect the library *without* code shrinking, but not once it was applied, despite being obfuscation aware. On the other hand, LibScout [15], ATVHunter [37], Orlis [34], and others, do not have any mechanism to account for this common obfuscation.

Code shrinking is one example of the challenges encountered when identifying libraries in obfuscated apps. In the following, we highlight multiple key challenges to library identification in obfuscated apps and how *Libra* tackles them.

3.2 Challenges to Library Identification

C1: Multiple Root Packages. In some cases, a library may have multiple root packages, e.g., `com/foo` and `com/bar`, which presents a challenge for library identification techniques since they need to be able to accurately associate both packages with the same library. However, if there are no interactions between these packages, traditional approaches using method calls, inheritance, and field access/writes to create class and library relations may struggle.

To address this, *Libra* identifies all root packages in a library by looking for first-level code units, and ensuring that a root package subsumes all its sub packages. This is done with a bottom-up approach to ensure the root package has first-level code units. This allows *Libra* to independently evaluate each package in an app against all root packages of a library to accurately identify the library. This also allows *Libra* to effectively manage transitive dependencies by treating them as multiple root packages.

C2: Shared Root Packages. Libraries may have a common root package, either intentionally or due to obfuscation techniques such as package flattening. This causes an enlarged pool of classes, making it difficult to distinguish between the libraries under this shared root package as their classes in the APK, despite being under the same root package, belong to different libraries.

In order to address this challenge, *Libra* introduces a two-component similarity measure where the first component represents the number of matched methods within the library candidate in the app, and the second component represents the number of matched methods in the actual library. When multiple libraries are present under the same root package, the library-candidate component naturally yields a lower value. Conversely, the library ratio component remains unaffected by the number of libraries within the library candidate. Incorporating these two components together allows *Libra* to accurately detect libraries sharing a root package as the similarity measure adjusts per each candidate under the shared root package.

C3: Code Shrinking. A standard step in the building of an app APK is Code Shrinking, where the compiler or obfuscator removes code deemed not required at runtime, such as unused classes, fields, methods, and instructions, from the app [12]. This process permanently removes code artifacts from the app, potentially diminishing the identity of a library in an irreversible manner. Tools have made the observation that a substantial difference (e.g., three times or more) in the size of a candidate library in an APK and an actual library package indicates that they are likely different libraries [37,40]. As shown in §3.1, this causes problems for library identification as the overall similarity between a shrunk library bytecode in an app APK and its corresponding actual library package decreases significantly.

To address this, *Libra* utilizes a resilient two-component weighted similarity calculation. By incorporating weights, *Libra* effectively addresses the impact of missing methods, reducing its influence on the overall similarity score. Specifically, by assigning less weight to the in-app ratio, *Libra* maintains its effectiveness in scenarios involving Code Shrinking.

C4: Instruction Bias. The Dalvik bytecode Instruction Set encompasses a wide range of instructions, which may initially appear beneficial for improving discrimination power when used for learning the identity of a library. However, in reality, this complexity presents a challenge as app compilation and obfuscation can introduce alterations to the bytecode, resulting in discrepancies compared to the libraries' bytecode used to build the models. These alterations include instruction reordering, arithmetic operation splitting, condition flipping, call resolution changes, and more. If a detection approach learns too much about the instructions, it becomes overly sensitive to obfuscation techniques that modify instructions and opcodes. Conversely, learning too little about the instructions leads to a loss of precision, causing different methods to appear too similar.

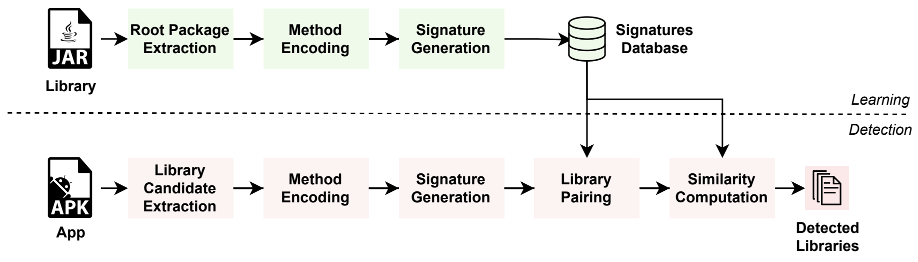

Fig. 2. Workflow of *Libra* with an offline learning phase and an online detection phase.

To overcome this, *Libra* encodes the instructions into a compact representation by mapping multiple opcodes to the same symbol. Moreover, it solely focuses on the mnemonic of the instructions, disregarding the operands as they are often subject to change by obfuscators. This approach enables *Libra* to strike a good balance between overlearning and underlearning the instructions, providing a more effective detection capability.

4 Detailed Design of *Libra*

Fig. 2 shows the workflow of *Libra*. We formulate the problem of Android third-party library identification as a pair-wise similarity problem. The problem takes in a set of library artifacts (JAR or AAR files) and an input app (APK file) where both consist of a set of classes which need to be compared with particular data and a similarity operator. *Libra* uses a two phase approach: An offline learning phase in which it builds a database of library signatures by processing the library artifacts and extracting pertinent information; and an online detection phase in which it identifies library candidates in an app, extracts their signatures and performs a pair-wise similarity computation with the offline database to identify the names and versions of the library candidates used in the app.

4.1 Learning Library Identities

In the learning phase, *Libra* first takes in an input library file, disassembles it, then extracts the root package name(s) of the library and groups the associated classes for signature extraction. *Libra* processes the classes under each identified package and computes a signature composed of a header and a body for each method defined in a class. Finally, it stores the library metadata (e.g., name, version) and signature into a database for later retrieval during the online detection phase. The following learning phases of *Libra* will be elucidated: (1) Root Package Extraction and (2) Signature Extraction.

Root Package Extraction. *Libra* traverses the package hierarchy of the library in breadth-first order looking for the first non-empty package, i.e., a root package

```
com.google.firebase
├── iid
│   └── FirebaseInstanceIdReceiver
└── messaging
    ├── FirebaseMessaging
    ├── ...
    └── reporting
        ├── ...
        └── MessagingClientEvent
```

(a) Package structure for the library `com.google.firebase:firebase-messaging:23.0.0`.

```
com.google.firebase.iid
└── FirebaseInstanceIdReceiver
```

(b) Root packages extracted and flattened for package `iid` .

```
com.google.firebase.messaging
├── ...
├── FirebaseMessaging
└── MessagingClientEvent
```

(c) Root packages extracted and flattened for package `messaging`.

Fig. 3. An example of a library having multiple root packages and the resulting root packages extracted along with the classes.

that contains at least one class definition. Note that there may be multiple packages associated with a single library if there are more than one package containing code units at the same level in the hierarchy. This allows *Libra* to handle the case of multiple root packages (C1). An example of this is shown in Fig. 3. Each package has the associated classes grouped with it for signature extraction. The extracted root package is flattened where each class under the package, including in subpackages, become associated with the root package.

Method Encoding. For each method in a class group, *Libra* encodes the Smali disassembly of the method body into a compact representation by mapping multiple instruction opcodes to the same symbol and discarding instruction operands. Figures 4a and 4b show a sample method and its encoded body. The full encoding map between is shown in Table A.1. This encoding step allows *Libra* to avoid instruction bias during learning (C4) by creating a lower-resolution method body (counters overlearning) without destroying the information contained in the instructions order (counters underlearning).

Signature Generation. For each method in a class group, *Libra* extracts the method's parameter types, return type, and encoded body.[5] With these data

[5] We exclude the instance initializer method (`<init>`), the class initializer method (`<clinit>`), and the resources class (`R`) since these tend to be highly similar amongst apps and libraries which may lead to spurious matches.

```
.method newRealCall(Lokhttp3/OkHttpClient;Lokhttp3/Request;Z)Lokhttp3/RealCall;
    .locals 2
    .param p0, "client"    # Lokhttp3/OkHttpClient;
    .param p1, "originalRequest"    # Lokhttp3/Request;
    .param p2, "forWebSocket"    # Z
    new-instance v0, Lokhttp3/RealCall;
    invoke-direct {v0, p0, p1, p2},
Lokhttp3/RealCall;-><init>(Lokhttp3/OkHttpClient;Lokhttp3/Request;Z)V
    ...
    return-object v0
.end method
```

(a) Bytecode method in Smali.

```
header: (Lokhttp3/OkHttpClient;Lokhttp3/Request;Z)Lokhttp3/RealCall
body: call move call ... return
```

(b) Encoded method.

```
header: (XXZ)X
body: 3:yaGRLBMTdLBMTdLBMTdGEMGgCdGEMGgdMIKTyV:yaI6TJ6TJ6TBHgCBHgdJKTyV
```

(c) Signature with header and body.

Fig. 4. Encoding and signature generation process. The signature header has the method parameter and return types where non-primitives changed to `X`. The body has the computed fuzzy hash of method mnemonics.

points *Libra* constructs a fuzzy signature which includes a header and a body as explained in the following.

To compute the signature header, *Libra* constructs a fuzzy method descriptor using the following transformation: The types are kept if they are primitives in Java, such as `int`, `boolean`, but are changed to `X` if they are non-primitive, such as `java.lang.Object`, `java.lang.String` (C4). Masking the types is essential in cases where identifier renaming obfuscation has been applied. If non-primitive types were utilized, and renaming has occurred, a mismatch would arise between the signature headers of the method signatures being compared (C4). For example, if the parameter types `okhttp3/OkHttpClient`, `okhttp3/Request`, and `okhttp /RealCall`, from Fig. 4, were used for the signature header instead of replacing each with `X`, and identifier renaming was applied renaming them to `A`, `B`, and `C`, respectively, the two signature headers, `(okhttp3/OkHttpClientokhttp3/RequestZ )okhttp3/RealCall` and `(ABZ)C`, would then never match.

To compute the signature body, *Libra* applies context-triggered piecewise hashing (CTPH) [22] on the encoded opcodes sequence, producing a hash in the format shown in Fig. 4c.[6] The first part of the hash is the size of the rolling

[6] CTPH offers advantages over other hashing methods in this setup as it employs a recursive rolling hash where each piece of the hash is computed based on parts of the data and is not influenced by previously processed data. Consequently, if there are changes to the sequences being hashed, only a small portion of the hash is affected. This is a desirable property for library identification in obfuscated apps since changes to the library bytecode packed in the app are expected.

window used to calculate each piece of the hash, the second part is a hash computed with the chunk size, and the third part is the hash with the chunk size doubled. This approach enhances the ability to handle both coarse- and fine-grained changes within a sequence due to obfuscation (C4).

Finally, and due to the nature of offline learning, it is necessary to store the results for lookup during the online detection phase. *Libra* stores each library identity (name and version), root package names, fuzzy signature (header and body), and metadata associated with the library in a database.

4.2 Detection and Similarity Computation

In the online detection phase, *Libra* identifies the library names and versions used by an incoming Android app (typically an APK file). The detection phase consists of the following stages: (1) Library Candidate Extraction, (2) Signature Extraction, (3) Library Pairing, (4) Similarity Computation. Each stage depends on the previous as pertinent information and data is extracted and propagated.

Library Candidate Extraction. Similar to the root package extraction step in the learning phase, *Libra* traverses the package hierarchy of the app and identifies all the app root packages. It then parses the `AndroidManifest.xml` file of the app and identifies the main components and their packages, and discards root packages that belong to the app main components since these belong the app's own code and therefore not libraries. It then considers each of the remaining root packages a library *candidate* and groups its classes in the same manner as in the learning phase for usage by the subsequent stages of the analysis.

Method Encoding & Signature Generation. For each library candidate, *Libra* encodes methods and constructs their signatures following the same approach in §4.1.

Library Pairing. To reduce detection time complexity, *Libra* attempts to avoid unnecessary similarity comparisons by first trying to pair each library candidate with libraries learned in the offline phase grouped by name, i.e., a group of different versions of the same library. *Libra* pairs a library candidate with a member of a library group if both of the following conditions are met: (1) The library candidate package name matches with one or more of the root package names of the library in the database. (2) The difference between the number of classes of the library candidate and the library in question is less than a predefined threshold τ (defaults to 0.4). A formulation of the reduction in search space is provided in Appendix B. The first condition states that *Libra* is aware of the root package linked with the library candidate and exclusively compares it with other libraries that have the same package association. The second condition originates from a heuristic, suggesting that a substantial discrepancy in the number of classes between the library candidate and another library indicates a lower probability of them being the same [37, 40].

Similarity Compuation. For each library candidate C and learned library L pair $\langle C, L\rangle$, *Libra* computes a pair-wise similarity score between the methods M_C of the library candidate and the methods M_L of the learned library by, first, computing the set M of pair-wise mutually-similar methods:

$$M = \{\langle m_C, m_L\rangle \mid m_C \in M_C, m_L \in M_L, \\ S(m_C) = S(m_L), \Delta(H(m_C), H(m_L)) \leq \delta\} \tag{1}$$

where S is the fuzzy method signature function, H is the fuzzy method hash function, Δ is the Levenshtein distance [23] which represents how similar the two method hashes are to each other, and δ is a predefined threshold (defaults to 0.85). The threshold δ is used to handle changes in the method instructions due to obfuscation (C3, C4) and was chosen from prior work [37].

Given M, *Libra* computes the final similarity score as the weighted sum of the ratios of matched methods in the library and the app, given by:

$$sim(C, L) = \alpha \cdot \frac{\min(|M_L|, |M|)}{|M_L|} + \beta \cdot \frac{\min(|M_C|, |M|)}{|M_C|} \tag{2}$$

where α, β are weighting parameters such that $\alpha + \beta = 1$. The similarity score ranges from 0 (lowest) to 1.0 (highest).

The purpose of these weighting parameters is to adapt to different degrees of code shrinkage by dampening the impact of code removal on the overall similarity score. In our experiments, we observed that α and β values of 0.8 and 0.2, respectively, yielded satisfactory overall results when code shrinking has been applied (C3), or there exists a shared root package between libraries (C2). The latter scenario arises when multiple libraries are associated with the same root package, potentially resulting in a low match ratio for the library candidate. However, through appropriate weighting, this challenge is overcome, enabling the determination that the library candidate and library are indeed similar.

Finally, *Libra* ranks the results based on the final similarity score, serving as a confidence indicator for the likelihood of the library's presence in the app, and reports the top k matches (defaults to 1).

5 Evaluation

We implemented *Libra* in Python in 2.4k SLOC. For fuzzy hashing, *Libra* relies on ssdeep [22]. Our experiments were conducted on an Ubuntu 20.04 server with Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20 GHz and 252 GiB of RAM.

We conducted several experiments to determine the effectiveness of *Libra* against state-of-the-art tools, including LibScout [15], LibPecker [43], Orlis [34], LibID-S [41], and Libloom [21].[7]

[7] Note that the Android SDK Support Library [7] was excluded from the counts for consistency with all evaluated tools.

We used the default settings for all the tools. For LibID [41], we used the LibID-S variant since it performs the best [41] and set a 0.8 threshold for LibScout to maintain consistency with LibID.

We used two public benchmarks in our experiments: the ATVHunter [37] and Orlis [4, 34] datasets. These two datasets were selected due to their widespread usage by prior work and their inclusion of diverse obfuscation techniques. The ATVHunter dataset consists of 88 non-obfuscated apps, and three sets of 88 (3 x 88) apps obfuscated with control flow randomization, package flattening and renaming, and code shrinking using Dasho [2]. In contrast, the Orlis dataset is organized based on the obfuscator used, encompassing the obfuscation techniques used in the ATVHunter dataset along with string encryption. The dataset consists of 162 non-obfuscated apps, and three sets of 162 (3 x 162) apps obfuscated with the obfuscators Allatori [1], Dasho [2], and Proguard [6].

For each experiment, we measured Precision, Recall, and F1 score for detection effectiveness, and the average detection time for the runtime overhead. The identify of a library in all experiments consists of both a name and a version number. A *true positive* (TP) is identified when a tool reports a library and the app contains that library. A *false positive* (FP) is identified when a tool reports some library that is not contained in the app. A *false negative* (FN) is counted when a tool does not report a library even though it is contained in the app.

5.1 Effectiveness Results

Table 1 shows detection results on the non-obfuscated ATVHunter dataset. The effectiveness of *Libra* is evident where it successfully identifies all non-obfuscated libraries contained in the apps. Here, *Libra* outperforms prior studies achieving an overall 88% F1 score, showcasing an improvement from prior work ranging from 7% up to 484%. Other tools show lower precision values resulting from higher numbers of falsely identified libraries.

With obfuscation enabled, *Libra*'s TP rate consistently outperforms all prior work with its recall ranging from 43% to 90% across all techniques. Table 2 shows *Libra* and Libloom both displaying effectiveness against control flow randomization with low FNs for the two, while others prove unsuccessful. Interestingly, Libloom scores a higher F1 score with this technique. This is due to Libloom discarding the order of instructions within a method, making it more resilient to techniques that also disrupt instructions order, at a cost of reduced precision as more methods appear similar (C4). Nevertheless, *Libra* still outperforms it across all the remaining obfuscation techniques.

With package flattening in Table 3, there is a general decrease in detection power for all tools as most utilize package hierarchy structures as features however, *Libra* maintains the best TP rate while achieving an F1 score performance increase from 67% up to 1386% across all tools. *Libra*'s robust weighted similarity calculation proves resilient against code shrinking in Table 4, correctly identifying 90% of libraries, and outperforming the remaining tools in F1 score by a range of 24% to 755%.

Table 1. Detection results on ATVHunter non-obfuscated apps.

Tool	TP	FP	FN	Precision	Recall	F1 score
LibScout	57	84	1	0.4043	0.9828	0.5729
LibPecker	50	43	8	0.5341	0.8621	0.6596
Orlis	5	3	53	0.5889	0.0862	0.1504
LibID-S	57	112	1	0.3373	0.9828	0.5022
Libloom	58	25	0	0.6988	1.0000	0.8227
Libra	58	16	0	0.7838	1.0000	0.8788

Table 2. Detection results on ATVHunter control-flow-randomization-obfuscated apps.

Tool	TP	FP	FN	Precision	Recall	F1 score
LibScout	5	0	53	1.0000	0.0862	0.1587
LibPecker	23	15	35	0.6012	0.3966	0.4779
Orlis	5	4	53	0.5484	0.0862	0.1490
LibID-S	0	12	58	0.0000	0.0000	-
Libloom	58	25	0	0.6988	1.0000	0.8227
Libra	48	40	10	0.5455	0.8276	0.6575

Table 3. Detection results on ATVHunter pkg-flattening-obfuscated apps.

Tool	TP	FP	FN	Precision	Recall	F1 score
LibScout	0	0	58	-	0.0000	-
LibPecker	2	1	56	0.5740	0.0345	0.0651
Orlis	5	3	53	0.5889	0.0862	0.1504
LibID-S	1	1	57	0.5000	0.0172	0.0333
Libloom	12	11	46	0.5217	0.2069	0.2963
Libra	25	18	33	0.5814	0.4310	0.4950

Table 4. Detection results on ATVHunter code-shrinking-obfuscated apps.

Tool	TP	FP	FN	Precision	Recall	F1 score
LibScout	0	1	58	0.0000	0.0000	-
LibPecker	3	2	55	0.5735	0.0517	0.0949
Orlis	4	2	54	0.5904	0.0690	0.1235
LibID-S	3	11	55	0.2143	0.0517	0.0833
Libloom	33	24	25	0.5789	0.5690	0.5739
Libra	52	36	6	0.5909	0.8966	0.7123

With the non-obfuscated Orlis dataset in Table 5, *Libra* outperforms all compared tools by 11% to 540%, maintaining the same F1 score observed in the previous section and retaining the highest precision. This trend persists across all obfuscated apps, where *Libra* consistently achieves the highest F1 score.

Tables 6 and 7 show detection results for apps obfuscated by Allatori and Dasho. Overall, even though the results show a decline in F1 score performances by all tools, *Libra* demonstrates the most resilience towards these techniques with an improvement ranging from 7% to 605% and 15% to 455% for the F1 score on Allatori and Dasho-obfucated apps respectively. *Libra* correctly iden-

Table 5. Detection results on Orlis non-obfuscated apps.

Tool	TP	FP	FN	Precision	Recall	F1 score
LibScout	102	144	1	0.4146	0.9903	0.5845
LibPecker	99	87	4	0.5309	0.9612	0.6840
Orlis	8	5	95	0.6116	0.0777	0.1378
LibID-S	101	216	2	0.3186	0.9806	0.4810
Libloom	103	53	0	0.6603	1.0000	0.7954
Libra	101	25	2	0.8016	0.9806	0.8821

Table 6. Detection results on Orlis Allatori-obfuscated apps.

Tool	TP	FP	FN	Precision	Recall	F1 score
LibScout	7	0	96	1.0000	0.0680	0.1273
LibPecker	71	40	32	0.6357	0.6893	0.6614
Orlis	6	3	97	0.6169	0.0583	0.1065
LibID-S	72	166	31	0.3025	0.6990	0.4223
Libloom	89	61	14	0.5933	0.8641	0.7036
Libra	92	50	11	0.6479	0.8932	0.7510

Table 7. Detection results on Orlis Dasho-obfuscated apps.

Tool	TP	FP	FN	Precision	Recall	F1 score
LibScout	37	52	66	0.4157	0.3592	0.3854
LibPecker	6	11	97	0.3481	0.0583	0.0998
Orlis	8	11	95	0.4039	0.0777	0.1303
LibID-S	37	97	66	0.2761	0.3592	0.3122
Libloom	96	201	7	0.3232	0.9320	0.4800
Libra	51	30	52	0.6296	0.4951	0.5543

Table 8. Detection results on Orlis Proguard-obfuscated apps.

Tool	TP	FP	FN	Precision	Recall	F1 score
LibScout	102	144	1	0.4146	0.9903	0.5845
LibPecker	99	87	4	0.5309	0.9612	0.6840
Orlis	8	5	95	0.6116	0.0777	0.1378
LibID-S	101	216	2	0.3186	0.9806	0.4810
Libloom	103	53	0	0.6603	1.0000	0.7954
Libra	101	27	2	0.7891	0.9806	0.8745

tifies the most libraries with the apps obfuscated by Allatori, and achieves the highest precision against both obfuscators. In contrast, the detection rates of LibScout, LibPecker, and LibID-S decline due to the effects of package flattening and control flow randomization on their package hierarchy structure features.

Finally, for Proguard in Table 8 where only identifier renaming and package flattening are enabled, the metrics across all tools remain unchanged from the non-obfuscated results. The results show that *Libra* outperforms other tools by 10% to 535%, displaying its resilience to the techniques applied by Proguard.

Table 9. Average runtime of all experiments.

Tools	Library Learning (seconds per library)	App Learning (seconds per app)	Library Detection (seconds per app)
LibScout	1.66	-	5.04
LibPecker	-	-	509.40
Orlis	5.18	-	850.30
LibID-S	0.23	2.07	0.60
Libloom	0.25	0.42	1.26
Libra	4.39	-	9.20

Overall, the results demonstrate *Libra*'s greater detection effectiveness than the state of the art, surpassing prior works by a margin ranging from 7% to 540% for non-obfuscated apps, and from 7% and to 1,386% for obfuscated ones.

5.2 Runtime Overhead

Table 9 shows the aggregate average learning and detection time per library and app for experimented tools. Time is divided into Library Learning, App Learning, and Library Detection, as tools perform different operations. *Libra* exhibits slightly longer learning and detection times (4.39 s learning per library, 9.20 s detection per app) than three of the five tools, although still within the same order of a few seconds. This is partially attributed to its relatively costly method-level granularity for computing fuzzy hashes. LibScout's fuzzy hashing similarly contributed to a higher detection time. LibPecker and Orlis were the least efficient, with longer runtimes due to class matching and code analysis. Both LibID-S and Libloom demonstrated fast learning and detection, although this comes at the expense of precision.

Overall, the performance of *Libra* meets the practical requirements and expectations for its intended use.

6 Discussion and Limitations

6.1 Threats to Validity

The obfuscation techniques used to evaluate the effectiveness of *Libra* and the state of the art were chosen from readily available, established benchmarks in the field. There may exist other obfuscation techniques in the literature that are not captured by these obfuscators. The used obfuscation tools are what developers commercially use for their apps which gives an accurate representation of how the different detection tools perform on apps in the wild.

The default values for the thresholds in §4 were chosen to offer good trade-offs for library detection in general cases out of the box. However, these thresholds may prove to be too high when dealing with specific obfuscation techniques that involve the insertion or removal of significant amount of code. To address this, the thresholds could be parameterized based on the detection of certain obfuscation techniques, allowing for better adaptability and accuracy in different scenarios. Parameter tuning against different obfuscation techniques could also be performed to further refine the thresholds and the detection power of *Libra*.

6.2 Performance Optimization

Certain aspects of the approach and its implementation can be optimized to achieve higher runtime performance. First, libraries can be processed in parallel during the learning process to cut down on the overall effective learning time. Second, early cutoffs can be employed during the detection phase if it is unlikely that the number of matched methods would exceed what is need to produce a high-enough final similarity score. This can be a check that is calculated on-the-fly while comparisons are being made. Finally, the comparisons performed during the library pairing step are all independent and can be parallelized to reduce the overall detection time per app.

6.3 Native Libraries

Libra currently only supports identifying bytecode libraries within Android apps. Apps could also utilize native libraries, written in C/C++, via the Java Native Interface (JNI) [3]. Identifying (obfuscated) native libraries in an app comes with its own challenges that extend beyond the scope of this work [14,17,30]. As such, we defer the identification of native libraries to future work.

7 Related Work

Prior studies on bytecode library identification have explored diverse approaches with varying degrees of effectiveness against obfuscation. Earlier techniques relied on package and class hierarchies to measure similarity between app packages and libraries. For example, LibScout [15] used package-flattened Merkle trees to obtain fuzzy method signature hashes. LibPecker [43] constructed class signatures from class dependencies and employed fuzzy class matching at the package level for similarity comparisons. LibRadar [26] built clusters of apps and libraries and generated fuzzy hashing features from the clusters based on the frequency of Android API calls. Techniques dependent on class and package hierarchies showed limited resilience to obfuscation techniques [28, 39], particularly package renaming and flattening, since obfuscators could easily manipulate class connectivity by merging or splitting classes and packages.

In more recent approaches, method instructions were used to enhance resilience to obfuscation. For instance, LibD [24] constructed library instances using homogeny graphs and utilized opcode hashes in each block of a method's Control-Flow Graph (CFG) for feature extraction. Orlis [34] constructed a textual call graph for methods and employed fuzzy method signatures to compute similarity. LibID [41] constructed CFGs from library binaries for feature extraction and utilized Locality-Sensitive Hashing for similarity. ATVHunter [37] used class dependency graphs to split library candidates and utilized both method CFGs and basic-block opcodes features for similarity measurement. Libloom [21] encoded signature sets from package and classes in a Bloom Filter and computed similarity with a membership threshold. While these tools offered some resilience to obfuscation techniques, their detection power degraded with package flattening and code shrinking as demonstrated in our experiments.

8 Conclusion

We introduced *Libra*, an Android library identification tool designed to tackle the challenges of detecting libraries within Android apps, particularly in the presence of obfuscation. *Libra* effectively addresses issues such as multiple and shared root packages, code shrinking, and instruction bias. Employing a two-phase learning and detection approach, *Libra* utilizes novel techniques to handle obfuscation and code shrinkage. These techniques involve leveraging data from method descriptors and instructions, encoding method instructions, employing fuzzy algorithms, and utilizing a two-component weighted similarity calculation. Our benchmarking results on multiple datasets showcase the effectiveness of *Libra*, demonstrating its ability to accurately identify library names and versions across various degrees of obfuscation.

Acknowledgment

We thank the anonymous reviewers for their insightful feedback. Opinions expressed in this article are those of the authors and do not necessarily reflect the official policy or position of their respective institutions.

A Method Encoding Codebook

Table A.1 shows the codebook used by *Libra* to encode method instructions. We conducted feature selection to determine the best mapping using Fisher's score [18] to gain insights into the most discriminatory instructions. Our analysis revealed that field getters, setters, and arithmetic operators exhibited low variance, making them less useful for discrimination. Consequently, we decided to combine these arithmetic instructions into a single move instruction.

B Search Space Reduction from Library Pairing

The pairing size complexity for pairs that satisfy condition one is $O(k)$, where n is the number of libraries in the database, and $k \ll n$ represents the group size.

Table A.1. Bytecode encoding codebook used by *Libra*.

Smali Instruction	Encoded Representation
`nop`	-
`move* v0, v1`	`move`
`move-result* v0`	`move`
`return*`	`return`
`const* v0, lit`	`move`
`monitor-* v0`	`monitor`
`check-cast v0, type`	`call`
`instance-of v0, v1, type`	`call; move`
`array-length v0, v1`	`call; move`
`new-* v0..vn, type`	`call; move`
`goto* ref`	`jump`
`cmp* v0, v1, v2`	`if; move`
`if-* v0, v1, ref`	`if`
`*get* v0, v1, v2`	`move`
`*put* v0, v1, v2`	`move`
`invoke-* v0..vn ref`	`call`
`neg-* v0, v1`	`move`
`not-* v0, v1`	`move`
`*-to-<type> v0, v1`	`call; move`
`arith./log.-* v0, v1, v2`	`move`

On the other hand, the pairing size complexity for condition two is $O(|P_{C2}|)$, where P_{C2} is defined as:

$$P_{C2} = \left\{ \langle C, L \rangle \mid C \in A, L \in D, \frac{\text{abs}(|A| - |D|)}{\max(|A|, |D|)} < \tau \right\}$$

where C is the library candidate, L is the library, A is the app, and D is the database. If no conditions are met, the library candidate is paired with the entire database, resulting in a pairing size complexity of $O(n)$. Note that this is unlikely as there are a wide range of library sizes from the order of 10^0 to 10^3 and condition two is likely to be met.

References

1. Allatori. https://allatori.com/
2. Dasho. https://www.preemptive.com/products/dasho/
3. Get started with the NDK. https://developer.android.com/ndk/guides
4. Libdetect dataset. https://sites.google.com/view/libdetect/home/dataset
5. Maven repository: Central. https://mvnrepository.com/repos/central
6. Proguard. https://www.guardsquare.com/proguard
7. Support Library | Android Developers. https://developer.android.com/topic/libraries/support-library
8. SolarWinds attack explained: And why it was so hard to detect. https://www.csoonline.com/article/3601508/solarwinds-supply-chain-attack-explained-why-organizations-were-not-prepared.html (2020)

9. Synopsys research reveals significant security concerns in popular mobile apps amid pandemic. https://news.synopsys.com/2021-03-25-Synopsys-Research-Reveals-Significant-Security-Concerns-in-Popular-Mobile-Apps-Amid-Pandemic (2021)
10. Number of apps available in leading app stores as of 3rd quarter 2022. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/ (2022)
11. Numbers from Google I/O: 3 billion active Android devices. https://9to5google.com/2022/05/11/google-io-2022-numbers/ (2022)
12. Shrink, obfuscate, and optimize your app. https://developer.android.com/studio/build/shrink-code.html (2023)
13. Ali, M.: Sensors Sandbox. https://github.com/mustafa01ali/SensorsSandbox
14. Almanee, S., Ünal, A., Payer, M., Garcia, J.: 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). IEEE (2021)
15. Backes, M., Bugiel, S., Derr, E.: Reliable third-party library detection in android and its security applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (2016)
16. Derr, E., Bugiel, S., Fahl, S., Acar, Y., Backes, M.: Keep me updated: An empirical study of third-party library updatability on android. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (2017)
17. Duan, R., Bijlani, A., Xu, M., Kim, T., Lee, W.: Identifying open-source license violation and 1-day security risk at large scale. In: Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security (2017)
18. Fisher, R.A.: The use of multiple measurements in taxonomic problems. Annals of eugenics **7**(2), 179–188 (1936)
19. Glanz, L., Amann, S., Eichberg, M., Reif, M., Hermann, B., Lerch, J., Mezini, M.: Codematch: obfuscation won't conceal your repackaged app. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (2017)
20. Han, H., Li, R., Tang, J.: Identify and inspect libraries in android applications. Wireless Personal Communications **103**(1), 491–503 (2018)
21. Huang, J., Xue, B., Jiang, J., You, W., Liang, B., Wu, J., Wu, Y.: Scalably detecting third-party android libraries with two-stage bloom filtering. IEEE Transactions on Software Engineering (2022)
22. Kornblum, J.: Identifying almost identical files using context triggered piecewise hashing. Digital investigation **3**, 91–97 (2006)
23. Levenshtein, V.I., et al.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady (1966)
24. Li, M., Wang, W., Wang, P., Wang, S., Wu, D., Liu, J., Xue, R., Huo, W.: Libd: Scalable and precise third-party library detection in android markets. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE) (2017)
25. Liu, B., Liu, B., Jin, H., Govindan, R.: Efficient privilege de-escalation for ad libraries in mobile apps. In: Proceedings of the 13th annual international conference on mobile systems, applications, and services. pp. 89–103 (2015)
26. Ma, Z., Wang, H., Guo, Y., Chen, X.: Libradar: fast and accurate detection of third-party libraries in android apps. In: Proceedings of the 38th international conference on software engineering companion (2016)
27. Narayanan, A., Chen, L., Chan, C.K.: Addetect: Automated detection of android ad libraries using semantic analysis. In: 2014 IEEE Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP) (2014)

28. Sihag, V., Vardhan, M., Singh, P.: A survey of android application and malware hardening. Computer Science Review **39**, 100365 (2021)
29. Soh, C., Tan, H.B.K., Arnatovich, Y.L., Narayanan, A., Wang, L.: Libsift: Automated detection of third-party libraries in android applications. In: 2016 23rd Asia-Pacific Software Engineering Conference (APSEC) (2016)
30. Tang, W., Luo, P., Fu, J., Zhang, D.: Libdx: A cross-platform and accurate system to detect third-party libraries in binary code. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) (2020)
31. Tang, Z., Xue, M., Meng, G., Ying, C., Liu, Y., He, J., Zhu, H., Liu, Y.: Securing android applications via edge assistant third-party library detection. Computers & Security **80** (2019)
32. Wang, H., Guo, Y., Ma, Z., Chen, X.: Wukong: A scalable and accurate two-phase approach to android app clone detection. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis (2015)
33. Wang, Y., Rountev, A.: Who changed you? obfuscator identification for android. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft). pp. 154–164. IEEE (2017)
34. Wang, Y., Wu, H., Zhang, H., Rountev, A.: Orlis: Obfuscation-resilient library detection for android. In: 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft) (2018)
35. Wang, Y., Chen, B., Huang, K., Shi, B., Xu, C., Peng, X., Wu, Y., Liu, Y.: An empirical study of usages, updates and risks of third-party libraries in java projects. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 35–45. IEEE (2020)
36. Xu, J., Yuan, Q.: Libroad: Rapid, online, and accurate detection of tpls on android. IEEE Transactions on Mobile Computing **21**(1) (2020)
37. Zhan, X., Fan, L., Chen, S., We, F., Liu, T., Luo, X., Liu, Y.: Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In: 43rd International Conference on Software Engineering (2021)
38. Zhan, X., Fan, L., Liu, T., Chen, S., Li, L., Wang, H., Xu, Y., Luo, X., Liu, Y.: Automated third-party library detection for android applications: Are we there yet? In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 919–930. IEEE (2020)
39. Zhan, X., Liu, T., Fan, L., Li, L., Chen, S., Luo, X., Liu, Y.: Research on third-party libraries in android apps: A taxonomy and systematic literature review. IEEE Transactions on Software Engineering **48**(10) (2022)
40. Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P.: Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In: Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks (2014)
41. Zhang, J., Beresford, A.R., Kollmann, S.A.: Libid: reliable identification of obfuscated third-party android libraries. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 55–65 (2019)
42. Zhang, Y., Wang, J., Huang, H., Zhang, Y., Liu, P.: Understanding and conquering the difficulties in identifying third-party librariesfrom millions of android apps. IEEE Transactions on Big Data (2021)
43. Zhang, Y., Dai, J., Zhang, X., Huang, S., Yang, Z., Yang, M., Chen, H.: Detecting third-party libraries in android applications with high precision and recall. In: IEEE 25th Conference on Software Analysis, Evolution and Reengineering (2018)
44. Zhang, Z., Diao, W., Hu, C., Guo, S., Zuo, C., Li, L.: An empirical study of potentially malicious third-party libraries in android apps. In: 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (2020)