ABSTRACT
Third-party libraries are vital components of Android apps, yet they can also introduce serious security threats and impede the accuracy and reliability of app analysis tasks, such as app clone detection. Several library detection approaches have been proposed to address these problems. However, we show these techniques are not robust against popular code obfuscators, such as ProGuard, which is now used in nearly half of all apps. We then present LiiID, a library detection tool that is more resilient to code shrinking and package modification than state-of-the-art tools. We show that the library identification problem can be formulated using binary integer programming models. LiiID is able to identify specific versions of third-party libraries in candidate apps through static analysis of app binaries coupled with a database of third-party libraries. We propose a novel approach to generate synthetic apps to tune the detection thresholds. Then, we use F-Droid apps as the ground truth to evaluate LiiID under different obfuscation settings, which shows that LiiID is more robust to code obfuscators than state-of-the-art tools. Finally, we demonstrate the utility of LiiID by detecting the use of a vulnerable version of the OkHttp library in nearly 10% of 3,958 most popular apps on the Google Play Store.

CCS CONCEPTS
• Security and privacy → Software reverse engineering.

KEYWORDS
Third-party library, Android, Obfuscation, ProGuard

1 INTRODUCTION
In recent years, the smartphone ecosystem has developed rapidly, reaching around 1.4 billion by 2018, with over 80% of the market share controlled by Android [32]. The success of the Android ecosystem is fueled by the rich variety of third-party apps. Third-party libraries play a vital role in Android app development. They enable developers to: promote their apps with social media support; monetize their apps through advertisements; or simply facilitate app development and extend app functionality. In fact, a previous study has shown that, on average, over 60% of sub-packages in Android apps are from common libraries [33]. Nevertheless, the prevalence of third-party libraries can also bring new challenges that may affect the security and privacy of the Android ecosystem.

One challenge is that the presence of libraries may constitute a barrier for mobile app analysis. First, app clone detection schemes should not take library code into account during analysis. Since a large proportion of program code is contributed by libraries, which may well be used in many other apps without modification, the accuracy of detection schemes will be significantly skewed if library code is not properly excluded. Second, library code may result in significant overhead during static analysis. Static analysis is the basis for many Android research topics, but it is often challenged by computing power and resources [4]. Since libraries are often irrelevant to the main functionality of an app, removing them, or analyzing them separately, would improve efficiency [37].

Another major challenge is that third-party libraries may introduce serious security and privacy threats. One vulnerability in a popular library could compromise a great number of apps. For example, Facebook Android SDK version 3.15 [23] and certain versions of the OkHttp library [38], both of which were popular in Android apps, contained authentication vulnerabilities. In addition, several popular libraries have spied on SMS messages [24] or even established backdoors [25]. Attackers may also inject malicious code into popular libraries and redistribute them through unofficial platforms to remotely control smartphones or steal information. According to recent research, more than half of all Android apps with vulnerabilities were at risk due to libraries [36].

These challenges motivate the design of reliable and scalable techniques to identify libraries in mobile apps. However, there are several challenges. First, many apps adopt code obfuscators, such as ProGuard [16], to obfuscate the name of classes, fields, and methods; approaches based on simple identifier matching [28, 39] are not applicable for such apps. Second, many detection schemes tried to extract library candidates at scale by using clustering approaches [19, 21, 33]. Although these schemes require no prior knowledge about the libraries before clustering, they require significant effort to label each cluster later. In addition, due to the lack of detailed information on libraries, clustering-based schemes cannot determine which version of a library is inside an app. Thus, they cannot be used to study whether apps use a vulnerable version...
of libraries. Last, library code can be dynamically changed during the build process by code optimizers. For instance, ProGuard, an open source tool that has been integrated into the Android build system, will, by default, optimize all code and remove unused dead-code during the build process. This process is called code shrinking or dead-code elimination. In addition, ProGuard can apply package modification operations, which includes package flattening and class repackaging, to obfuscate packages and change class hierarchies. A recent study has shown that 49% of apps (88% of obfuscated apps) used ProGuard and 21% of them applied package modification operations [34]. Most of the existing studies (e.g., LibRadar [21] and LibD [19]) failed to consider code shrinking and package modification operations, while others that did consider them (e.g., LibScout [5] and Orlis [35]) showed poor accuracy in our experiments.

Aim. We design a novel third-party Android library detection tool, LibID, that can reliably identify the library version used in Android apps given the library and app binaries. LibID is resilient to common code obfuscation techniques, including identifier renaming, code shrinking, control-flow randomization, and package modification.

Design. We start by building library and app profiles directly from their binaries by selecting obfuscation-resilient features. Then, we match library classes with app classes based on their profile. In particular, we formulate the constraints between matched classes and construct novel Binary Integer Programming (BIP) models to find the optimal match pairs that satisfy these constraints. LibID also leverages a special Locality-Sensitive Hashing (LSH) technique to improve its efficiency and scalability by avoiding full pair-wise feature comparisons. In terms of computation complexity, we design two schemes for LibID: LibID-S and LibID-A, with a focus on scalability and accuracy, respectively. In addition, we develop a systematic approach to generate synthetic apps with different versions of libraries. We use generated apps as the ground truth to tune the detection thresholds of LibID and tentatively compare the results with state-of-the-art library detectors: LibScout, Orlis, and LibPecker [40]. Furthermore, we apply all library detectors to open source apps on F-Droid and benchmark their performance. Finally, we conduct a large-scale study on popular Google Play apps and identify a high proportion of popular apps using vulnerable libraries.

Contributions. We make the following contributions:
(1) We use recent software engineering methods, such as LSH ensemble, to design and implement a state-of-the-art third-party library detection system for Android.
(2) We are the first to formulate library matching constraints and convert library detection into a BIP problem.
(3) We propose a method of generating synthetic apps, where the ground truth is known, in order to empirically determine appropriate detection thresholds.
(4) We show LibID achieves higher F1 score than other state-of-the-art methods both when no obfuscator is used and when ProGuard/Allatori/DashO is used.
(5) We show that LibID successfully detects vulnerable versions of the OKHttp library in nearly 10% of popular Google Play apps; LibScout only finds 7.5%.
(6) We make all source code available for other researchers1.

2 SYSTEM DESIGN

The aim of LibID is to reliably identify third-party libraries from Android binaries against popular obfuscation techniques, which requires fine-grained analysis that is usually time-consuming. To improve the usability of LibID, we introduce two library identification schemes: LibID-S (§3) and LibID-A (§4), which focus on scalability and accuracy, respectively.

The workflow of LibID is illustrated in Fig. 1. It consists of two major steps: profiling and matching. In particular, LibID first generates a profile for every library and app binary. Then, these profiles are compared using a matching process. The output of the matching process allows LibID to report the likelihood of any given library appearing in any given app. If there is a match, LibID additionally reports the in-app package names of all matching libraries in an app, which enables researchers to quickly find the library code in the app.

3 LIBID-S

3.1 Profiling

As shown in Fig. 1, the profiling step can be divided into three stages. We do not differentiate the profiling of apps and libraries here because they use the same techniques. In particular, LibID-S first

---

1https://github.com/ucam-cl-dtg/LibID
Table 1: Grammar for the basic block signature

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Block</td>
<td>B</td>
</tr>
<tr>
<td>Field</td>
<td>F0</td>
</tr>
<tr>
<td>Read</td>
<td>F0</td>
</tr>
<tr>
<td>Write</td>
<td>F1</td>
</tr>
<tr>
<td>Method Call</td>
<td>M0</td>
</tr>
<tr>
<td>New Instance</td>
<td>M0</td>
</tr>
<tr>
<td>Other</td>
<td>M1</td>
</tr>
<tr>
<td>String</td>
<td>S</td>
</tr>
<tr>
<td>If</td>
<td>I</td>
</tr>
<tr>
<td>Return</td>
<td>R</td>
</tr>
<tr>
<td>Goto</td>
<td>G</td>
</tr>
</tbody>
</table>

This augmented basic block signature represents a feature of the class and the value is the set of class signatures of that class.

Furthermore, we associate the class features with each basic block signature by prefixing the basic block signature with four new components as shown in Fig. 2, which include:

- **Class Access Flag**: This field keeps the native Java access flag of the class since obfuscators typically do not change these class access flags.
- **Superclass Name**: If the class is extended from a class in the Android SDK, then this field will record the name of the superclass. Otherwise, this field is $[X]$.
- **Class Interfaces**: This field only records class interfaces that are from the Android SDK. If there are multiple qualified interfaces, a separator ‘|’ will be inserted in this field to separate them. This field is ‘[]’ if the class does not implement any class from the Android SDK.
- **Method Descriptor**: We use the same format as LibScout [5] and OSSPolice [12] to profile class methods. Each method parameter is represented by its type in Dalvik and non-framework types will be replaced with a placeholder $X$.

This augmented basic block signature represents a feature of the class, and thus we call it a *class signature*, a class typically has a set of class signatures. An example of a class signature is given in Fig. 2. Then, each Android app or library can be characterized by a *class signature dictionary*, where the key is the name of its member class and the value is the set of class signatures of that class.

3.2 Matching

After generating both app and library profiles, LiID-S compares their similarity and calculates the confidence of the library being used in the app. We refer to this procedure as the matching process. As shown in Fig. 1, the matching process can be broken down into the following three steps: class matching, dependency matching and library matching.

**Class Matching.** This stage aims to find the candidate matches between library and app classes. An intuitive way to achieve our goal is to compare whether the signatures of these classes are the same. However, obfuscators can remove unused basic blocks from the original class or even delete the whole class, and thus a library class may have various signatures in different apps. In this regard, instead of looking for the same class signature sets, we define the *class signature containment* index as follows to quantify the similarity between two classes.

**Definition 3.1 (Class Signature Containment).** For two classes $c_1$ and $c_2$, whose signature set is $S_1$ and $S_2$, respectively, the class signature containment of $c_1$ in $c_2$ is defined as:

$$S_C(c_1, c_2) = \frac{|S_1 \cap S_2|}{|S_1|}$$  (1)

In LiID-S, library profiles are extracted directly from their binaries, and thus they contain a complete set of library class signatures. However, in-app versions of a library class may contain only a small subset of signatures due to the dead-code elimination. Therefore, LiID-S only considers an app class $c_a$ as a candidate match to a library class $c_l$ if $S_C(c_a, c_l)$ is equal to 1, since code shrinking does not change this index.

This containment index has been used to detect piggybacking apps [11, 13]. However, it requires pair-wise comparison which is time-consuming. Locality-Sensitive Hashing (LSH) [2] provides an efficient solution to this problem. It pre-processes the dataset by creating signature hashes such that similar items have a higher chance of sharing the same hash. Here, LiID-S applies a special LSH index technique named LSH Ensemble [42], which supports the containment query, to speed up the matching process.

**Dependency Matching.** The dependency matching stage aims to find the true match pairs from the candidates. Based on Definition 3.1, it is likely that a single class in an app is matched to multiple candidate library classes, or vice versa. However, this is problematic since each library class can produce at most one matching in-app class, and vice versa. For example, in Fig. 3, it is clear that $L1$.class and $L2$.class cannot both be matched to $A1$.class. In fact, there can be at most two true matching pairs in Fig. 3.
In general, suppose there are \( m \) library classes \( C_L \) that have candidate matches in the app and \( n \) app classes \( C_A \) that have candidate matches in the library. Then, we can generate a candidate match matrix \( \mathbf{R} \in \{0,1\}^{m \times n} \), where \( R_{ci,ca} = 1 \) if and only if \( c_i \in C_L \) and \( c_a \in C_A \) are a candidate match. A true class match matrix \( C \in \{0,1\}^{m \times n} \) ensures a unique match between library and app classes should have the following uniqueness constraints:

\[
C_{c_i,c_a} \leq R_{c_i,c_a}, \quad \forall c_i \in C_L, c_a \in C_A
\]

\[
\sum_{c_a \in C_A} C_{c_i,c_a} \leq 1, \quad \forall c_i \in C_L
\]

\[
\sum_{c_i \in C_L} C_{c_i,c_a} \leq 1, \quad \forall c_a \in C_A
\]

Or in words, there should be at most one non-zero value in each row and column in \( C \) and \( C \) is a subset of \( R \).

In Android apps and libraries, each package may include a set of classes and several sub-packages. In most instances, obfuscators do not change the internal package structures. Therefore, LibID-S by default will only match an app class with a library class if they are at the same level in the matching package. For example, in Fig. 4, suppose each library class is a candidate match to an app class. Here, we use the superscript “*” to denote candidate match pairs (e.g., \( a.class \) is a candidate match to \( a*.class \)). If we want to check whether the library package Okio and app package a is a match, then the match pair \((Okio/o.class, a/o*.class)\) is valid while \((Okio/a.class, a/b/a*.class)\) is invalid because classes in the latter pair are at a different level to the matching package. However, if we are comparing package Okio with sub-package b in Fig. 4 (a), then that pair is valid because both a.class and a*.class are directly under the matching package.

To make sure the matched class pairs are \( valid \), we need some additional constraints. Suppose there are \( m_P \) library packages \( P_L \) and \( n_P \) app packages \( P_A \) that contain matched classes or their sub-packages contain matched classes. For example, if \( C_L \) consists of two classes \((a/b/x.class\) and \( a/c/y.class\)), then \( P_L \) would have three elements \((a/b, a/c, a)\). Let \( P \in \{0,1\}^{m_P \times n_P} \) be the true package match matrix, then the hierarchy relation between classes can be ensured by the following hierarchy constraints:

\[
P_{pl,p_a} \leq P_{parent(pl),parent(p_a)}, \quad \forall pl \in P_L, p_a \in P_A \tag{3a}
\]

\[
\sum_{p_a \in P_A} P_{pl,p_a} \leq 1, \quad \forall pl \in P_L \tag{3b}
\]

\[
C_{c_i,c_a} \leq P_{package(c_i).package(c_a)}, \quad \forall c_i \in C_L, c_a \in C_A \tag{3d}
\]

Constraint 3a ensures the app and library packages can only be a match if their parent package also matches. In particular, if a matching package is already a top-level package (no parent package), this constraint will be ignored. Constraint 3b and 3c make sure each library package can only be matched to at most one app package. Constraint 3d ensures that classes can only be matched if their packages match. Collectively, these hierarchy constraints guarantee that matched class pairs have the same hierarchy.

The objective of LibID-S is to find the maximum number of matched class pairs, which can be formulated to the following Binary Integer Programming (BIP) problem:

\[
\text{maximize} \quad \sum_{c_i \in C_L, c_a \in C_A} C_{c_i, c_a}
\]

\text{subject to} \quad \text{Constraint 2 and 3}

The BIP problem has been extensively studied for many decades and there are many sophisticated software packages available to help solve this problem [30]. Previous work has shown that 21% of apps using ProGuard have applied package modification operations, which includes package flattening and class repackaging [34]. If package flattening is enabled, rule-specified packages will be put into a single parent package, but the structure of the package will be preserved. Notably, we do not require the hierarchy of packages to be the same. For instance, it is possible to have a match between package Okio and a/b in Fig. 4 (a). Therefore, LibID-S is robust to package flattening and package renaming tools that could change the hierarchy of root packages but not internal package structures.
By contrast, if class repackaging is applied, rule-specified classes will be moved to a single parent package, and thus the classes hierarchies in the original package will no longer be preserved. An example is given in Fig. 4 (b), where all classes in the library have been repackaged to package a in the app. If we still use Constraint 3, then many true class matches would be missed. To address this problem, we create a Class Repackaging Detection (CRD) mode. When LibID-S runs in the CRD mode, every app package that does not have sub-packages could be the parent package of repackaged classes. Let \( P_A \) be the collection of these app packages, then LibID-S has the following hierarchy constraints when the CRD mode is on:

\[
\sum_{p_a \in P_A} p_{p_l, p_a} \leq 1, \quad \forall p_l \in P_L
\]

\[
P_{p_l^1, p_a} = p_{p_l^1, p_a}, \quad \forall p_l^1, p_l^2 \in P_L, p_a \in P_A
\]

\[
C_{c_l, c_a} \leq p_{\text{package}(c_l), \text{package}(c_a)}, \quad \forall c_l \in C_L, c_a \in C_A
\]

Or in words, all library packages could only be matched to a single app package that does not contain a sub-package. For example, in Fig. 4 (b), app package a could be a candidate match to the library package okhttptp3. In this way LibID-S is also able to deal with class repackaging techniques if shrinking is not applied at the same time.

**Library Matching.** LibID aims to pinpoint the third-party library version in Android apps. An intuitive solution is to check the proportion of library classes that are present in the app. However, some classes may have no viable signature, which we call *unproductive classes*. An example of an unproductive class is a class that contains only abstract methods (i.e., no basic block). Unproductive classes contribute little to the uniqueness of the library and will never be matched to other classes. Therefore, we exclude them from the proportion calculation. Nevertheless, this evaluation index is not robust against code shrinking. As a fix, we also consider the proportion of matched classes in the matched app package and define the *library match index* as follows:

**Definition 3.2 (Library Match Index).** Suppose the root package \( P_L \) of library \( L \) is matched to a package \( P_A \) in app \( A \), and there are \( N_C \) true class match pairs between \( L \) and \( A \), then the library match index of \( L \) in \( A \) is:

\[
M(L, A) = \frac{N_C}{\min(N_{P_l}, N_{P_A})}
\]

Where \( N_{P_l} \) and \( N_{P_A} \) are the number of productive classes in \( P_L \) and \( P_A \), respectively.

The library match index is suitable for quantifying the confidence of the presence of the library in the app. On the one hand, apps may use several libraries that share the same root package. In this case, \( N_C / N_{P_A} \) can be pretty low even if the library is used by the app. On the other hand, if code shrinking is applied, then \( N_C / N_{P_l} \) could be very low but \( N_C / N_{P_A} \) would remain high. In both cases, \( SC(L, A) \) can still retain a high value. Therefore, we can decide whether app \( A \) uses library \( L \) using two thresholds:

\[
M(L, A) > \Gamma_1 \quad \text{and} \quad \frac{N_C}{N_{P_l}} > \Gamma_2
\]

The threshold \( \Gamma_2 \) ensures there is enough information about the library for us to make a meaningful decision. In most cases, \( M(L, A) \) should be close to 100% if app \( A \) uses library \( L \). Although LSH will inevitably introduce false negatives and false positives, the relation constraints can help to reduce false alarms during the matching process. However, if both class repackaging and shrinking are applied, then \( M(L, A) \) can still be a small value. In this case, we can adjust the value of \( \Gamma_1 \) to balance between the detection of libraries and introducing false positives.

In general, each library update may modify only a small fraction of the code, and thus there may be multiple library versions that satisfy Constraint 5. For each matched library, LibID-S only keeps the versions that have the highest value of \( M(L, A) \).

**Summary.** LibID-S only relies on features that are consistent under code shrinking, control flow randomization, and package modification. The use of the *LSH Ensemble* greatly reduces the matching time by avoiding pair-wise comparison between classes and all calculations in LibID-S are relatively light-weight. Therefore, LibID-S is a suitable tool when computing resources are limited. Nevertheless, although we have considered the *uniqueness* and *hierarchy* constraints during the matching process, this coarse-grained analysis can still result in false matches between classes. In addition, if multiple libraries share the same root package in the app and each library only has a few classes remained in the app, LibID-S may not be able to identify these libraries.

### 4 LIBID-A

#### 4.1 Profiling

On the basis of LibID-S, we design a more accurate library detection system named LibID-A. LibID-A employs finer-grained features (invocation, interface, and inheritance dependencies) during the library detection process, which enables it to not only find library matches for each app more precisely but also pinpoint class match pairs with higher accuracy. Overall, LibID-S does not make use of the information about dependencies between different classes, which may result in incorrect matching. To address this problem, LibID-A further records the following additional information:

- **Method Calls:** If a callee is obfuscatable, LibID-A will record the name of the called class and method under the key of the caller class in the *invocation dictionary*.
- **Class Inheritance:** If a class’s superclass is obfuscatable, LibID-A will record the name of the superclass under the key of the current class in the *inheritance dictionary*.
- **Class Interfaces:** If the class’s interface is obfuscatable, LibID-A will record the name of the interface under the key of the current class in the *interface dictionary*.

Notably, we only keep records of the above information if the relating entities are obfuscatable (i.e., not from the Android SDK). Otherwise, they should have already been integrated into class signatures. Then, LibID-A will build dependency graphs based on these dictionaries. Here, we define a dependency graph as an undirected graph, where each node corresponds to a class and each edge represents an invocation, inheritance or interface dependency between these two classes. Notably, these dictionaries are only used to construct the dependency graph. No static identifier (e.g., method and class name) will be utilized for string matching as they can easily be changed by identifier renaming.
### 4.2 Matching

The matching process in LiInD-A also includes three major phases: **Class Matching.** This process is the same as the class matching in LiInD-S. We use the class containment similarity index in Definition 3.1 to find all candidate class matches while applying LSH Ensemble to speed up the matching process.

**Dependency Matching.** In contrast to LiInD-S, apart from uniqueness and hierarchy constraints, the dependency matching in LiInD-A also includes the following constraints:

- **Invocation Constraints:** For any two app classes \((c_a, c_a^2 \in C_A)\), if \(l_a^1\) invokes a method in \(c_a^2\) and the formatted method descriptor is \(d\), and their true class match in the library is \(c_l^1\) and \(c_l^2\), respectively, then there must be a method call from \(c_l^1\) to \(c_l^2\) whose descriptor is also \(d\). It is not necessarily true the other way around.

- **Inheritance Constraints:** For any two app classes \((c_a, c_a^2 \in C_A)\), if \(c_a^1\) is the superclass of \(c_a^2\), and their true class match in the library is \(c_l^1\) and \(c_l^2\), respectively, then \(c_l^1\) must also be the superclass of \(c_l^2\), and vice versa.

- **Interface Constraints:** For any two app classes \((c_a, c_a^2 \in C_A)\), if \(c_a^2\) is an interface of \(c_a^1\), and their true class match in the library is \(c_l^1\) and \(c_l^2\), respectively, then there must be a method call from \(c_l^1\) to \(c_l^2\) whose descriptor is also \(d\). It is not necessarily true the other way around.

The formulation of these constraints are very similar to LiInD-S, and thus we do not reiterate them here. Notably, the invocation and interface constraints do not work the other way around. The reason is that both method calls and interfaces can be deleted by code shrinking, but the superclass will always be kept if the child class exists. LiInD-A also has a CRD mode, which replaces the default hierarchy constraints with Constraint 4 while keeping other constraints the same.

The objective of LiInD-A can be formulated as follows:

\[
\text{maximize } \sum_{c_l^1 \in C_L, c_a \in C_A} C_{c_l^1, c_a} + w \sum_{c_a^1, c_a^2 \in C_A} \left( V + H + T \right)_{c_a^1, c_a^2}
\]

subject to five types of constraints described in this section

where \(V, H, T \in \{0, 1\}^{n \times n}\) are the invocation, inheritance, and interface matrix, respectively. Their element is equal to 1 if and only if there is corresponding dependency between \(c_a^1\) and \(c_a^2\). \(w\) is a weighted parameter that should be small enough (e.g., 0.0001). In other words, LiInD-A aims to find the optimal solution that has the largest number of class match pairs. If there are multiple solutions, it selects the one with the largest number of dependency matches.

**Library Matching.** To reliably detect libraries when both class repackaging and shrinking are applied, LiInD-A utilizes the dependency graph to perform better library matching.

**Relation Pruning:** Although class repackaging can move classes from different packages into a single package, the dependency graphs stay the same. Therefore, we should only consider classes in the app package that are reachable from any matched classes. In other words, an app class needs to be in the same connected component with any matched class in the dependency graphs in order to be considered. We name this process relation pruning.

**Ghost Hunting:** Library classes may invoke methods from classes that exist in another library under the same package name. This is most common when libraries are developed by the same company. For instance, Leakcanary calls methods from Leakcanary-Watcher and they share the same root-package com/squareup/leakcanary. In this case, we cannot eliminate the interference of other libraries through relation pruning. Instead, for each library class, if it depends on another class that is not in the library and not from Android SDK, we will record this connection and refer to the connected class as a ghost class. For each ghost class, we find its matched app class by dependency matching and remove it from the dependency graphs. This process is named ghost hunting.

For each library and app match, we first implement ghost hunting to eliminate the distraction of ghost classes and then apply relation pruning to avoid the interference of classes that are not from the library but in the same app package. Let \(N_AP\) be the number of qualified app classes, then the definition of library match index for LiInD-A can be updated as:

\[
M(L, A) = \frac{N_C}{\min(N_P, N_{P_A})}
\]

Where notations have the same meaning as in Definition 3.2. If the class matching is accurate, \(M(L, A)\) should be the same after applying class repackaging or when other classes are inside the same package. Similar to LiInD-S, we can use the Constraint 5 to decide if library \(L\) is used in app \(A\).

**Summary.** Overall, LiInD-A has enhanced the abilities of LiInD-S. By introducing finer-grained constraints, LiInD-A provides higher accuracy in terms of class-level matching, which further enables LiInD-A to pinpoint the version of libraries with greater accuracy. In addition, LiInD-A utilizes the dependency graphs to separate different library classes, which enables it to better deal with class repackaging and the interference of shared-package libraries. The adoption of finer-grained features will inevitably slow down the execution of LiInD-A. To achieve a balance between time efficiency and accuracy, LiInD-A also uses LSH Ensemble to speed up the matching process and apply various constraints to minimize false matches. The execution of LiInD-A for different apps is completely parallelizable.

### 5 EVALUATION

LiInD uses Androguard [3] version 2.0 with the DAD decompiler to construct the CFGs of methods. We extend the original Androguard project to support multi-dex files and our signature algorithms. In addition, we make use of the datasketch [41] package to implement the LSH Ensemble algorithm. All LSH parameters are set to default values that are tuned by benchmarks. The BIP models are implemented with the Gurobi [17] optimizer. We deploy LiInD on a computing cluster node with 32 Intel Xeon Gold 6142 CPUs; each CPU has 12040MB RAM.

To evaluate the effectiveness of LiInD, we collect 69 popular libraries with 1,444 versions from central repositories (e.g., Maven Central and JCenter), open-source platforms (e.g., Github), and official websites. Since Androguard cannot directly parse the downloaded library jars, we convert these .jar files to .dex files using the dex2jar [27] tool. The decompilation, however, does not work
in all cases. During our experiment, around 94.17% of libraries are successfully converted, while LibScout successfully loads 94.47% libraries. In addition to LibScout, we compare LnilD with state-of-the-art work, Orlis [35] and LibPecker [40], under different settings.

5.1 Library Detection on Synthetic Apps

To determine the detection threshold $\gamma_1$ and $\gamma_2$ in Constraint 5, we propose a novel way to generate synthetic apps with different versions of libraries and different obfuscator configurations. We use ProGuard as the obfuscator due to its popularity. A recent study has shown that about 88% of obfuscated apps chose ProGuard as their only obfuscator, and 70% of them used the default ProGuard configuration [34].

Our method works by creating an Android project. Then, for each library in our dataset, we manually collect code snippets that provide main library functions from either the official documentation or example projects. We store these snippets in a database, along with the import statements, program, and dependencies for each library. In our dataset, the main APIs of most libraries are consistent for a wide range of versions. Therefore, a single generated app can be used with many library versions. We compile each app in the database under four different ProGuard settings, as shown in Table 2. Identifier renaming is enabled in all groups except when ProGuard is disabled.

We apply both tools to detect in-app libraries and calculate the number of true positives (TP), false positives (FP) and negatives (FN). Here, a false negative means the tool does not report any version of the library, while a false positive is counted if the tool reported some versions of the library excluding the correct one. Since we know in advance which library is used in each app, these evaluation indexes can be measured against ground truth. Then, we calculate the $F_1$ score to quantify the performance of both tools. To determine $\gamma_1$ and $\gamma_2$, we first set them to 1 and 0, respectively, to observe the result and then gradually adjust them to appropriate values ($\gamma_1 = 0.8$, $\gamma_2 = 0.1$) to achieve the highest $F_1$ score.

**Evaluation.** We tentatively compare the performance of LnilD with LibScout, Orlis, and LibPecker because they are state-of-the-art library detection tools which are designed to handle code shrinking. In particular, Orlis only reports matches between app and library classes. Therefore, we regard all library versions reported by Orlis as possible matches. For LibPecker, we set all thresholds based on the original paper. LibPecker reports the similarity between the app and each library candidate, and thus we choose the library version(s) with the highest similarity, if above the threshold, as the matched in-app library version(s).

Table 2 presents the experiment results under different ProGuard settings. It shows that LnilD achieves higher $F_1$ scores than other tools in all cases. When ProGuard is disabled, all tools except Orlis achieve high accuracy. Overall, Orlis is ineffective in identifying in-app library versions. Table 2 (b) shows that LnilD is much more effective than LibScout and LibPecker when class repackaging is enabled. LnilD accurately identifies libraries for more than 98.4% apps and reports fewer false negatives. In comparison, LibScout and LibPecker are correct for 0.7% and 9.6% apps, respectively, because they rely on the package hierarchy information.

<table>
<thead>
<tr>
<th>Table 2: Library Detection Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) ProGuard Disabled</td>
</tr>
<tr>
<td>Tool # apps # TP # FP # FN F1</td>
</tr>
<tr>
<td>LnilD 359 250 436 0.5114</td>
</tr>
<tr>
<td>LibScout 969 33 43 0.9623</td>
</tr>
<tr>
<td>LibPecker 1,045 984 38 23 0.9699</td>
</tr>
<tr>
<td>LnilID-S 1044 1 0 0.9995</td>
</tr>
<tr>
<td>LnilID-A 1045 0 0 1</td>
</tr>
</tbody>
</table>

| (b) Class Repackaging Enabled     |
| Tool # apps # TP # FP # FN F1    |
| LnilD 499 276 270 0.6464         |
| LibScout 7 2 1036 0.0133        |
| LibPecker 1,045 100 66 2 0.1747 |
| LnilID-S 1029 5 11 0.9923       |
| LnilID-A 1028 14 3 0.9918       |

| (c) Shrinking Enabled (Default ProGuard Setting) |
| Tool # apps # TP # FP # FN F1 |
| LnilD 226 169 537 0.3903 |
| LibScout 26 4 902 0.0543 |
| LibPecker 932 126 90 0.2382 |
| LnilID-S 924 8 0 0.9957 |
| LnilID-A 932 0 0 1 |

| (d) Shrinking and Class Repackaging Enabled |
| Tool # apps # TP # FP # FN F1 |
| LnilD 225 165 542 0.3889 |
| LibScout 0 0 932 N/A |
| LibPecker 932 22 47 863 0.0461 |
| LnilID-S 305 145 482 0.4931 |
| LnilID-A 831 40 61 0.9427 |

Table 2 (c) demonstrates that other tools are also ineffective against code shrinking. In particular, if code shrinking is enabled, library code may be partially removed in the app. We define an index $I_C$ to quantify the library information remaining in the app:

$$I_C = \frac{\text{# Signatures of Library Classes in App}}{\text{# Signatures of Library Classes in SDK}} \quad (7)$$

Based on the definition, $I_C$ will be 1 if code shrinking is disabled. We calculate $I_C$ for every synthetic app that applied code shrinking. Because synthetic apps are relatively simple, most of them contain only a small portion of library classes after code shrinking. Fig. 5 presents the relation between the recall and $I_C$ when shrinking is enabled. It reveals that LnilD behaves pretty well against code shrinking, while LibScout, LibPecker and Orlis only detect libraries when $I_C > 70\%$, and still fail to detect many libraries in that case.

The weakness of LnilID-S is clear when both shrinking and class repackaging are enabled. LnilID-S uses the library match index $M(L, A)$ in Definition 3.2 to quantify the confidence of a library $L$
being used in an app A. However, this value can be very low if the majority of library classes have been deleted (low $N_C/N_P$) while there are many other non-library classes in the same package (low $N_C/N_P$). By contrast, LIDI-A only considers app classes that are related to the library by eliminating other classes through ghost hunting and relation pruning, and thus its library match index in Equation 6 is still high for the right library. Nevertheless, LIDI-A does produce several false alarms. There are two reasons for the inaccuracy. First, class-level matching is not always accurate due to lack of hierarchy information, which would further influence the ghost hunting and relation pruning processes. Second, in a few cases, ProGuard keeps some library classes in the original package to ensure normal functioning while moving others to a single parent package, which could influence the accuracy of LIDI-A.

### 5.2 Library Detection on F-Droid Apps

Despite the promising results on synthetic apps, performance on more sophisticated apps may vary. In general, commercial apps may invoke more library functions than the collected code snippets, and they usually have a much larger code base and more complicated interactions between classes. In addition, using more advanced obfuscators may also affect the result. Therefore, we need a more realistic ground truth to evaluate the performance.

Wang et al. has compiled F-Droid projects under different obfuscators, including ProGuard, Allatori, and DashO, and published the dataset online [26]. The dataset contains 200-300 app binaries for each obfuscation configuration, with the ground truth of in-app library versions. However, apps in the dataset that should be obfuscated by ProGuard were, in fact, unobfuscated. Therefore, we manually compiled 215 F-Droid projects with ProGuard applied (both code shrinking and class repackaging are enabled). We use both self-compiled apps and the dataset provided by Wang et al. as the ground truth to evaluate the performance of library detectors on commercial apps with different obfuscators. LIDI-S and LIDI-A are set with the thresholds determined in §5.1. Since all tools may report multiple library versions, we define two types of precision: pinpoint precision and in-range precision. In particular, the pinpoint precision of a tool is the proportion of correctly identified libraries over all the libraries reported. When calculating the in-range precision, we regard all reported versions of a library as a single entity. If the reported versions of an entity include the correct version, then we refer to this entity as a correctly identified entity. The in-range precision of a tool is the proportion of correctly identified entities over all the entities reported. For example, if LIDI-A reports “ACRA-4.6.1, ACRA-4.6.2, Gson-2.5” while the ground truth is ACRA-4.6.1, then the respective pinpoint and in-range precision of LIDI-A are 1/3 and 1/2. The pinpoint and in-range $F_1$ scores are calculated using the pinpoint and in-range precision, respectively.

Table 3 presents the experiment results. Overall, LIDI-A behaves better than state-of-the-art tools in all four cases. The $F_1$ score (both pinpoint and in-range) of LIDI-A is greater than 0.5 when ProGuard is applied, while that of other tools is lower than 0.25. In all cases, LIDI has the highest in-range precisions among these tools. Nevertheless, results also reveal that the performance of LIDI for these F-Droid apps is not as good as in §5.1. We investigate several cases where LIDI-A does not identify the library correctly and find the following causes: (i) $l_c$ is less than 5% for some libraries.
We further analyze the app that LibID-A, which could defeat the design of LibID. To study the library usage in production apps and further evaluate (iii) the obfuscation in some apps changes the class access flag and class interfaces and splits a single library class into several classes, which could defeat the design of LibID-A.

5.3 Libraries in Popular Google Play Apps

To study the library usage in production apps and further evaluate our approach, we downloaded 3,958 apps from the top-100 apps across 59 categories on the Google Play store in April 2017. We employ both LibID (using LibID-A) and LibScout to analyze these popular apps. On average, LibID detects one more library per app than LibScout. Table 4 gives the summary of top five most popular libraries identified by LibID. In particular, the columns respectively present the number of apps using each library identified by LibID, LibScout, and both of them. The table shows that a large proportion of libraries have been missed by LibScout. We confirm the reason is that LibScout struggles to detect libraries after code shrinking.

Vulnerable Library Detection. A primary use case of LibID is to identify vulnerable in-app libraries. Here, we use OkHttp, a popular Android library used in 29% of apps in our dataset for managing HTTP-based network requests, as an example. There is a severe vulnerability in OkHttp for versions 2.x before 2.7.4, and 3.x before 3.1.2, that allows an attacker to bypass the certificate pinning (CVE–2016–2402) [8].

<table>
<thead>
<tr>
<th>Library</th>
<th># LibID</th>
<th># LibScout</th>
<th># Common</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gson</td>
<td>1419</td>
<td>1114</td>
<td>1093</td>
</tr>
<tr>
<td>Facebook</td>
<td>1243</td>
<td>1082</td>
<td>1058</td>
</tr>
<tr>
<td>OkHttp</td>
<td>1157</td>
<td>690</td>
<td>689</td>
</tr>
<tr>
<td>Okio</td>
<td>1139</td>
<td>563</td>
<td>561</td>
</tr>
<tr>
<td>Bolts</td>
<td>918</td>
<td>317</td>
<td>316</td>
</tr>
</tbody>
</table>

We apply LibID, LibScout, Orlis, and LibPecker to check if a vulnerable version of the OkHttp library is used in popular apps. The results are presented in Table 5. All tools find that many popular apps use at least one version of vulnerable OkHttp libraries. In particular, LibID identifies 393 affected apps, while the figure for LibScout and LibPecker is 296 and 339, respectively. All tools but Orlis agree on 294 apps, but the remaining list of affected apps differs. We notice that OkHttp contains a field marking the actual version of the library, and this field has not been obfuscated in the majority of our test apps. Therefore, we only need a little manual effort to verify the results. Note that LibID does not use string constants as a feature to detect libraries. Thus, the version information of OkHttp in the app binaries only helps us to confirm the result instead of assisting LibID. For those apps that LibID and LibScout disagree, all 99 additional apps reported by LibID are true positives. Meanwhile, LibScout uniquely reports 2 vulnerable apps, and one of them is a false positive; the other one is missed by LibID. We further analyze the app that LibID fails to detect. It turns out the app uses both a shrunk version of OkHttp 2.4.0 (vulnerable) and an unobfuscated version of OkHttp 3.4.1 (patched). During the matching process, LibID mismatches the OkHttp 2.4.0 library to the OkHttp 3.4.1 package but the matching confidence does not reach the threshold. As a result, this vulnerable app is missed by LibID.

Overall, the results demonstrate the accuracy and robustness of LibID, but it also raises security concerns about the Android platform. Although the vulnerability of OkHttp was reported in February 2016, around 10% of popular apps in the Google Play dataset still use an insecure version of the OkHttp library. We find more than 30% of apps in two categories (NEWS_AND_MAGAZINES and TRAVEL_AND_LOCAL) and at least 10% of apps under 25 categories use vulnerable versions of OkHttp. Meanwhile, a large proportion of OkHttp libraries used in popular apps are unpatched, including more than 44% of OkHttp libraries in the top 100 free apps and the other six categories. We also discover that three of Google’s apps use vulnerable versions of OkHttp libraries, all of which are installed by millions of users. Although we confirm that these three apps do not invoke the vulnerable certificate pinning method, it is still best practice to upgrade the library version to ensure vulnerable code is not introduced in a future release. Finally, we sent emails to the developer of the reported 393 vulnerable apps on December 23, 2017. As a result, we received 22 non-automatic replies. Among these replies, 17 said they will investigate, 3 have updated the OkHttp library, and 2 promised to update it in the future release.

5.4 Limitation

LibID selects several features to profile and detect libraries. These features, however, are not robust against more advanced obfuscation techniques. For instance, the class access flag and class interfaces can be modified if “allowaccessmodification” and “mergeinterface-saggressively” are enabled in ProGuard. API hiding obfuscation can also hide calls to the Android SDK through Java reflection. Nevertheless, these cases are relatively rare compared with code shrinking and package modification operations (§5.2). In addition, ProGuard suggests users exploit these advanced features cautiously because they can reduce the performance and may cause JVM problems [16]. The aim of LibID is to be the first to reliably detect libraries under the most popular obfuscation functions, including code shrinking, identifier renaming, control flow randomization, and package modifications. We achieve this goal.

One major concern of LibID is the scalability. Using library detection on unobfuscated F-Droid apps as an example (§5.2), we calculate the average time spent on profiling a library, profiling an app, and matching an app with libraries for each tool. The results are presented in Table 6. Although LibID is more time efficient than Orlis and LibPecker, it spent more time than LibScout. This is because LibID performs a finer-grained analysis compared with LibScout. Both the CFG construction and candidate class matching are
time-consuming. The BIP solver could also take a long time to find a solution if there are too many constraints. Nevertheless, the use of LSH Ensemble has greatly improved the matching time by avoiding pair-wise comparison and we design two working schemes to prioritize either scalability or accuracy. As seen in Table 6, LibID-S spent much less time in the matching process because it does not make use of the dependency graph. In addition, for large-scale analysis, we can run LibID in parallel on a computing cluster since each task is independent. In general, LibID-A is preferred for small-scale analysis, or large-scale analysis when there are enough computing resources; LibID-S is favored in other cases.

6 RELATED WORK

The high prevalence of third-party libraries in mobile apps has been an obstacle to app clone detection research for many years. Most of the existing studies employed naive whitelisting techniques to identify and exclude common libraries before analysis, either by comparing package names [1, 9] or the hash value of known libraries [10]. However, these approaches can miss many less-popular libraries, and the lack of granularity makes them incapable of detecting obfuscated libraries. More advanced work filters libraries by clustering. WuKong [33] chose the frequency of Android API calls as the feature to identify libraries by clustering. Andarwin [11] grouped similar semantic blocks from the Program Dependence Graph (PDG) to detect library and applied LSH techniques to accelerate the clustering process. LibDetect [14] identified and removed in-app library classes from an app using fuzzy hashing to increase the accuracy of app clone detection.

Grace et al. and Book et al. studied the presence and behavior of advertising libraries using a whitelist [7, 15]. Li et al. harvested 1,113 common libraries from 1.5 million Google Play apps by comparing the name of in-app methods and packages [18]. LibRadar [21] implemented and improved WuKong by providing an online detection platform and designing a better cluster algorithm. LibD [19] detected and classified library candidates based on dependencies between methods and packages. LibSift [29] performed library detection by comparing the primary components of the PDG. In addition, machine learning techniques have been applied to detect third-party libraries [20, 22]. Nevertheless, these proposals rely on the package hierarchies and can neither pinpoint library version nor handle popular obfuscation techniques such as code shrinking.

MobScanner [6] used weighted features to identify in-app library versions. However, it relies on string constants and is unreliable when intensive code shrinking is applied. Similar to LibID, LibScout [5] built library profiles from pre-collected library binaries. LibScout used a fuzzy descriptor to generate a method signature, and further obtain a class signature using the hash of in-class methods. Then, LibScout calculates the similarity of class signatures between a candidate package and a known library package to determine whether they are a match. Although LibScout is resilient against identifier renaming operations, it performs poorly if either dead-code elimination or class repackaging is enabled. OSSPolice [12] was designed to identify license violations. It extracted more features (e.g., string constants) from library binaries to better pinpoint library versions. Nevertheless, OSSPolice is only resilient to simple obfuscation techniques such as identifier renaming. Orlis [35] is another tool that can detect the library version in Android binaries. It assigned a weight to every method and class based on the number of instructions and tried to find the maximum weight bipartite matchings. Although it does not rely on any hierarchy information, Orlis requires pair-wise comparison and its threat model does not include code shrinking and package modifications.

Recently, LibPecker [40] utilized the class dependencies to perform obfuscation-resilient library matchings. Instead of matching the dependency graph, they encoded the graph into a set of fuzzy class signatures and calculate the Jaccard similarity between the library and app signatures to decide whether two classes are a match. Nevertheless, LibPecker relies on the package hierarchy and cannot handle drastic code shrinking. LibRadar [21] was also designed to be resilient to code shrinking and class repackaging. The authors have shown that Orlis outperformed LibDetect when identifying obfuscated in-app library classes. However, as shown in §5, Orlis is not designed to detect specific library versions.

7 CONCLUSION

This paper has presented LibID, a reliable tool that identifies third-party Android libraries and their version in app binaries. LibID includes two detection schemes: LibID-S and LibID-A, which focus on scalability and accuracy, respectively. We convert the abstract library identification problem into a BIP problem that has been well studied in the research community. Overall, LibID overcomes several limitations found in previous work and is able to determine the version of third-party libraries used in an app binary using identifier renaming, shrinking, control flow randomization, and package modification techniques. Evaluation is supported by our novel method that semi-automatically generates apps containing third-party libraries, as well as an analysis of hundreds of F-Droid apps. This provides valuable ground truth data to support accurate evaluation and comparison of our approach to previous work. Our experiments show that LibID can detect many more libraries than prior art, especially when code shrinking is enabled and the package hierarchy is modified. LibID is open source software and we encourage researchers to use this tool to assist their own research.

ACKNOWLEDGMENTS

Jiexin Zhang is supported by the China Scholarship Council. Alastair R. Beresford is partly supported by The Boeing Company and EPSRC under Grant No.: EP/M020320/1. Stephan A. Kollmann is supported by Microsoft Research and The Boeing Company. We thank Diana A. Vasile, Ricardo Mendes, Martin Kleppmann, and Peidong Zhu for helpful discussion and insight. We also thank anonymous reviewers for their feedback on the paper.
REFERENCES


