LibKit: Detecting Third-Party Libraries in iOS Apps

Daniel Domínguez-Álvarez
IMDEA Software Institute
Madrid, Spain
University of Verona
Verona, Italy

Alejandro de la Cruz
IMDEA Software Institute
Madrid, Spain

Alessandra Gorla
IMDEA Software Institute
Madrid, Spain

Juan Caballero
IMDEA Software Institute
Madrid, Spain

ABSTRACT

We present LibKit, the first approach and tool for detecting the name and version of third-party libraries (TPLs) present in iOS apps. LibKit automatically builds fingerprints for 86K library versions available through the CocoaPods dependency manager and matches them on the decrypted app executables to identify the TPLs (name and version) an iOS app uses. LibKit supports apps written in Swift and Objective-C, detects statically and dynamically linked libraries, and addresses challenges such as partially included libraries and different compiler versions and configurations producing variants of the same library version. On a ground truth of 95 open-source apps, LibKit identifies libraries with a *precision* of 0.911 and a *recall* of 0.839. LibKit also significantly outperforms the state-of-the-art CRiOS tool for identifying TPL boundaries. When applied to 1,500 apps from the iTunes Store, LibKit detects 47,015 library versions, identifying popular apps that contain old library versions.

CCS CONCEPTS

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

KEYWORDS

mobile apps, iOS, third-party libraries

ACM Reference Format:
Daniel Domínguez-Álvarez, Alejandro de la Cruz, Alessandra Gorla, and Juan Caballero. 2023. LibKit: Detecting Third-Party Libraries in iOS Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616344

1 INTRODUCTION

There exist many scenarios where entities other than the developer may need to know the third-party libraries (TPLs) a mobile app uses. These include: (i) a company selecting among different proprietary apps wants to know whether those apps contain vulnerable TPLs;(ii) a malicious library has been identified and users want to know whether the apps they use contain it; (iii) an app is observed to perform some privacy-violating behavior and market auditors need to establish if the behavior comes from the app or a TPL it uses; and (iv) regulators need to measure the prevalence of advertisement libraries to determine if a merger of ad companies may create a dominant market position [23].

In both iOS and Android ecosystems, most mobile apps and many TPLs are closed-source, so TPL identification approaches need to operate on released app packages (IPAs for iOS, APKs for Android) and should be able to identify TPLs even if the library source code is not available. TPL identification comprises three problems. The goal of *boundary identification* is to split the code of an app (e.g., Mach-O executable for iOS, DEX bytecode for Android) into components where one component corresponds to the app's code and there is one additional component for each TPL. The goal of *library identification* is to output the names of all TPLs an app uses. Finally, the goal of *library version identification* is to output the name and version of all TPLs an app uses.

In this paper, we present the first TPL detection approach that can identify *the name* and *version* of libraries present in an iOS app, and a tool called LibKit that implements it. A few works have proposed iOS TPL identification approaches [10, 31, 35]. However those approaches either use clustering-based techniques that only address the boundary identification problem (i.e., CriOS [31], Tang et al. [35]) or are specific to TPLs that are available both in Android and iOS [10].

LibKit follows prior Android TPL identification approaches that operate in two phases: *library fingerprint generation* and *library detection* [4, 5, 36, 38, 40, 42]. The main advantage over iOS clustering-based techniques is that our approach can name a library and identify its version, beyond identifying the boundaries between the app's and TPL's code. At a high level our approach works as follows: (1) It takes as input the library repository of CocoaPods, one of the major dependency managers for Swift and Objective-C projects, used by over 3M iOS apps [11]. (2) It generates a library version fingerprint for each library version in the CocoaPods repository. A library version fingerprint is a distinctive set of features that capture the unique properties of a library version. In our approach, a library version fingerprint comprises a set of class fingerprints, each capturing syntactic features about a class that is part of the library version. The fingerprints are extracted statically from the library version's binary code. The generated fingerprints are then stored in a database. (3) Given an app, LibKit decrypts the app's binary code, analyzes its binaries to obtain the class features, and produces a set of class fingerprints, one for each class in the app. Then, it looks for matching app's class fingerprints in the library class fingerprints in the database. LibKit outputs the list of library names and versions identified in the app. Steps (1) and (2) only need to be performed once for each library version. LibKit can incorporate new library versions as soon as they appear in CocoaPods, repeating (1) and (2) only for new versions.

Existing fingerprint-based Android TPL identification approaches cannot be easily ported to iOS because they rely on Android's package structure, which does not exist in iOS. The reliance on package structure has been identified as a main limitation of Android TPL identification approaches by a recent independent evaluation [41]. The only Android fingerprint-based approach that does not leverage the package structure is ORLIS [38]. However, ORLIS does not support library version identification and it performs worst among publicly available Android TPL identification tools [41].

We design an automated build pipeline for CocoaPods libraries, which allows us to produce fingerprints for 86,597 TPL versions belonging to 14,043 TPLs. This is the largest library database in TPL identification works, seven times larger than the largest (12K versions) used in Android TPL identification [40]. We design our fingerprints to work on class information available in iOS native code, regardless if the native code comes from compiling Objective-C or Swift source code, and despite the class information not being as rich as that in Android's bytecode. Our fingerprints leverage a similarity hash for matching class fingerprints that are similar, but not identical. The matching algorithm aims for the highest possible coverage of app classes and allows for partial library coverage, i.e., identifies TPLs even when, due to dead code elimination, only part of the TPL code makes it into the app's native code.

We evaluate LibKit according to its ability to identify the correct name and version of libraries included in a set of iOS apps. For this, we have created two ground truth datasets: one with the name of 1,066 libraries included in at least one of 95 open-source apps available in Github (GT95) and another one with 511 library versions (name and version) that appear in at least one of 43 open-source apps (GT43). For library identification on GT95, LibKit achieves a *precision* of 0.911, *recall* of 0.839, and *F1 score* of 0.874. When considering all libraries in our ground truth, including those not present in our database, it achieves a *precision* of 0.911, *recall* of 0.524, and *F1 score* of 0.666. We also evaluate the accuracy of LibKit for identifying not only the correct library name but also the correct version. For library version identification on GT43, LibKit achieves a *precision* of 0.721, *recall* of 0.716, and *F1 score* of 0.725.

We also evaluate LibKit against the state of the art. Since LibKit is the first tool for library identification and library version identification in iOS, there is no perfect baseline to compare with. The closest iOS work is CriOS because it is generic (i.e., targets any TPL), but it only addresses the problem of boundary identification. We obtained the original CriOS source code from its authors and spent significant work updating it so that it could handle recent iOS apps. We compare LibKit against CRiOS for the boundary identification problem. LibKit significantly outperforms CRiOS for this task, achieving an F1 score of 0.722 compared to 0.307. As explained above, fingerprint-based TPL identification tools for Android rely on package structure or do not address the problem of library version identification. We tried porting LibScout to iOS, but the required changes were so significant that it no longer represented the original approach and could not be used as a representative baseline. Comparing LibKit's accuracy with that reported in the recent independent evaluation by Zhang et al. [41] is the best we can do in this situation. In their evaluation, LibScout was the best-performing tool for both library identification and library version identification and LibKit achieves higher F1 scores on iOS apps than those reported by Zhang et al. for LibScout on Android apps.

Finally, we apply LibKit on 1,500 apps from the iTunes Store for which we do not have a ground truth. This experiment simulates how LibKit could support the work of a security analysis that needs to identify which TPLs a set of applications include, to either identify apps that include known malicious libraries or known vulnerable library versions. LibKit detects 47,015 library versions, with a median of 25.5 libraries per app. We report the top 10 libraries identified and show that popular apps contain old library versions.

To allow future work on iOS TPL identification to use LibKit as a baseline, we have released the code, database, and ground truth required to replicate this research [26, 27].

This paper makes the following contributions:

- It presents the first library identification and library version identification approach for iOS apps.
- It automatically builds fingerprints for 86,597 versions of 14,043 TPLs available in the CocoaPods repository.
- It builds a ground truth of 95 apps with their libraries and 43 apps with their library versions.
- We release LibKit, our fingerprint database, and our ground truth datasets [26, 27].

2 BACKGROUND

This section first explains how iOS apps are developed in Section 2.1 and then details how developers can integrate TPLs using the CocoaPods package manager in Section 2.2.

2.1 iOS App Development

The original official language for developing iOS apps was Objective-C, a superset of the C language that adds an object-oriented layer and runtime. In 2014, Apple released Swift, a new programming language to replace Objective-C. Swift was designed to maintain compatibility with Objective-C. For this reason, even a pure Swift app contains parts of the Objective-C runtime and each Swift class is also an Objective-C class.

Building. iOS app developers typically use the official Xcode IDE as development environment. Xcode handles all steps for building an app including compilation, linking, and post-build steps like signing, and packaging. All executable code produced in the build process is in Mach-O executable files. These include the app's code, dynamic libraries, statically linked libraries, and vendored libraries, i.e., open-source libraries whose source code is directly copied into the app's source base. Dynamic libraries are distributed as Frameworks comprising of the library code in a Mach-O executable and any resources the library requires. Frameworks typically contain their name in the path to their executable, making their identification easier. However, the Framework files may not be named with the known library name and the version is not disclosed. The binary code of statically linked libraries and vendored libraries will instead be included into the main app executable. Nowadays, iOS apps typically contain multiple Mach-O executables: one executable for the main app (including the app's code, vendored libraries, and statically linked libraries) and one executable for each Framework that is not a system library pre-installed in iOS.

Packaging. iOS apps are distributed as IPA containers, which are ZIP files with a specific structure. At its top-level, an IPA file has a *Payload* folder that contains all files necessary to run the app, with the exception of system libraries. The Payload folder contains one or more app bundles (i.e., .app subfolders). Each app bundle contains an *Info.plist* configuration file, an executable, associated resources (e.g., translations, assets, images), and its Frameworks. Apps needs to be digitally signed. iOS will refuse to run apps not properly signed. An alternative is to use a jailbroken device that circumvents signature checks.

Distribution. To publish an app through the official iTunes Store, a developer must create a developer account. Apps in the iTunes Store are signed by Apple. The FairPlay DRM protection used by Apple will encrypt the executable code using a developer-specific key before publishing it in the store, leaving other files (e.g., resources) unencrypted.

2.2 CocoaPods

iOS developers may use a package manager (PM) to handle their app's dependencies, e.g., download third-party libraries and add them to the app's project. There exist three iOS PMs: *CocoaPods* [11], *Carthage* [8], and *Swift Package Manager* (Swift PM) [34]. All three support libraries distributed as source code or precompiled. A key difference between them is that CocoaPods uses a central repository to store specifications (called *Podspecs*) of the available library versions. This central repository allows library developers to make their TPLs visible to app developers, who can easily find TPLs to use. In contrast, Carthage and Swift PM do not have a central repository and thus app developers must find libraries on their own.

When a developer wants to publish in CocoaPods a new library, or a new version of an already available library, it submits a *Podspec* file to the public repository. CocoaPods assumes semantic versioning for library versions (MAJOR.MINOR.PATCH). Figure 1 shows an excerpt of the *Podspec* of Firebase, a popular Google library. The *Podspec* first contains general information about the library such as its name, version, and the source from where the library can be downloaded (e.g., URL to a repository or an archive). Then, it lists dependencies broken into modules, or subspecs in the CocoaPods jargon. In the example, the library has two subspecs (Core, AdMob), but only the default *Core* subspec will be installed by default. The Firebase/Core subspec depends on two other libraries (FirebaseAnalytics, FirebaseCore). The AdMob subspec depends on the Firebase/Core module and the library Google-Mobile-Ads-SDK.

To include libraries in an app, the developer generates a *Podfile* that states the libraries that the app depends on. CocoaPods uses the *Podfile* to automatically download the specified libraries (and its dependencies) and to incorporate them into the app's Xcode project

```
{ "name": "Firebase",
  "version": "4.7.0",
  "source": {"http": "https://dl.google.com/.../Firebase-4.7.0.tar.gz"},
  "default_subspecs": [ "Core" ],
  "subspecs": [
    {"name": "Core",
     "dependencies": {"FirebaseAnalytics": "4.0.5", "FirebaseCore": "4.0.12"},},
    {"name": "AdMob"
     "dependencies": {"Firebase/Core": [], "Google-Mobile-Ads-SDK": "7.26.0"},},]
}
```

Figure 1: Simplified *Podspec* for Firebase 4.7.0.

```
platform :ios, '9.0'
target 'test' do
  use_frameworks!
  pod 'Firebase', '4.7.0'
  pod 'Objection'
  pod 'SSZipArchive', '~> 2.2'
end
```

Figure 2: *Podfile* for a test app requiring three libraries.

so that they are compiled (if distributed as source code) and linked when building the app. Figure 2 shows an example *Podfile* for a *test* app, which requires three libraries (or *pods*). In this case, the app developer requests through the use_frameworks! statement that the libraries are built (if possible) as separate Frameworks. This causes the app to contain four Mach-O binaries, one with the test app code and one for each library. For Firebase, the developer requires version 4.7.0. Since the *Podfile* does not specify any Firebase subspec, only the default Firebase/Core subspec will be included. For Objection, since the developer did not specify any version, CocoaPods will install the latest version. For SSZipArchive, the developer used the optimistic operator ~> 2.2, which is equivalent to range [2.2.0, 2.3.0). Since developers may not specify a library version, or may provide a range of valid versions, CocoaPods decides which library versions are included among those satisfying the constraints. The CocoaPods installation produces a *Podfile.lock* file that resolves the dependencies into concrete library versions. In the example, the *Podfile.lock* file will specify that Firebase 4.7.0, Objection 1.6.1, and SSZipArchive 2.2.3 were included into the app's project. Unfortunately, the *Podfile.lock* file is not part of the built app. Thus, it is only available when building the app from source code and cannot be used for TPL detection in our scenario where the input is the app's binary code.

3 APPROACH

The key idea behind any fingerprint-based identification technique lies in representing an analyzed artifact (e.g., a malware sample, a TPL, a whole binary file, or a single compiled class) in a fingerprint capturing the unique features present in such artifact. Such fingerprint can then be used for reliable identification based on these distinctive features in other artifacts.

LibKit is a fingerprint-based technique to identify TPLs is iOS apps, and therefore comprises two phases, described in Figure 3: *library fingerprint generation* and *library detection.* Library fingerprint generation takes as input a large collection of known TPLs to generate a database of library fingerprints. LibKit can automatically collect and build fingerprints for libraries in the CocoaPods repository, although its modular design allows to analyze TPLs from other sources with some manual work. Specifically, library fingerprint generation retrieves a library by getting its CocoaPods

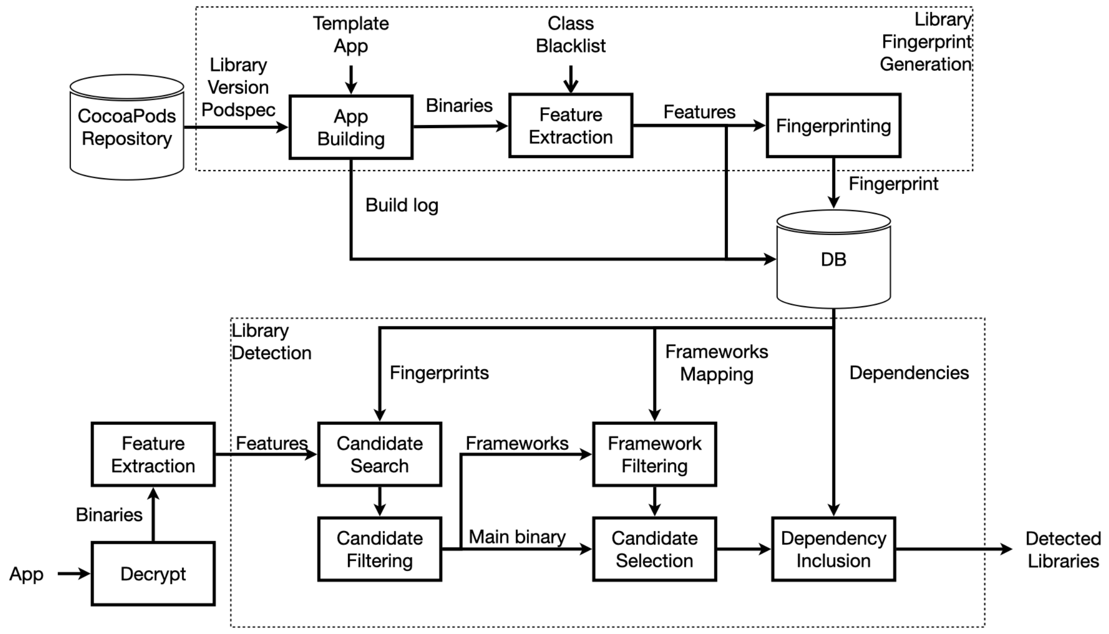

Figure 3: LibKit comprises two phases: The library fingerprint generation phase, at the top, builds the database of fingerprints of a set of known TPLs. In the second phase (library detection, at the bottom) the information in the database is used to analyze an app with unknown TPLs. LibKit builds fingerprints for each component of the app and aims to match them in the database of known libraries in an attempt to identify the name and the version of each library.

Podspec and builds it with all its dependencies. Taking the Firebase library in Figure 1 as an example, LibKit would retrieve the Firebase source code from the specified location, and then would retrieve all the dependencies (i.e. Firebase/Core, FirebaseAnalytics, FirebaseCore, Google-Mobile-Ads-SDK) recursively from CocoaPods. LibKit resorts to a template app that solely includes the Firebase library as dependency to produce the executable MachO binary files of the library. We use a template application because CocoaPods is meant to be integrated with an application project, and this process can produce the binary files to be analyzed regardless of whether the library –or any of its dependencies– is distributed as source code or pre-compiled as dynamic or static library.

Each MachO binary file is then analyzed by the Feature Extraction module, which employs an automated lightweight static analysis to extract syntactic code features. Such features are then used to produce a fingerprint of the library to be stored in the database. When extracting the features, LibKit uses a blacklist to discard features that may negatively effect the quality of fingerprint (Section 3.1 provides more details on this aspect). LibKit resorts to simhash [9] to compute the fingerprints of the extracted features. The advantage of simhash is that it uses a probabilistic method to generate similar fingerprints for similar objects. Therefore thanks to simhash the fingerprints of two consecutive minor versions (e.g. Firebase 1.4.7 and Firebase 1.4.8) of the same library are expected to have fingerprints closer in similarity than two versions of the same library separated by a major release (e.g. Firebase 0.4 and Firebase 1.7) or two different libraries (e.g. Firebase and Protobuf).

The library fingerprint generation phase only needs to be done once for each library version contained in the CocoaPods repository. LibKit already comes with a database of 86K library version fingerprints that we produced for our evaluation.

In the Library Detection phase (bottom part of Figure 3), LibKit analyzes a closed source iOS apps aiming to identify any of the known TPL fingerprints in the app binaries. To this end, it first needs to decrypt the executable code that comes encrypted due to Apple policies as explained in Section 2.1. It then analyzes each binary file with the same lightweight static analysis component used to extract features and build fingerprints for libraries in the library fingerprint generation phase. LibKit searches for TPL candidates having their fingerprint matching the features found in the app. This step is not as trivial as looking for exactly matching TPL fingerprints in the database. The executable code of a TPL in an app may have been produced following a different compilation process, which may include dead code elimination and other optimizations. This leads to have different fingerprints even for exactly the same library version. Thanks to simhash, though, fingerprints of the same library version are supposed to be at least similar, and therefore would match anyway given a similarity tolerance threshold. All the details of how LibKit searches and matches app features with candidate TPLs are further explained in Section 3.2.

3.1 Library Fingerprint Generation

The top part of Figure 3 describes the architecture of the library fingerprint generation. It takes as input the CocoaPods library repository and produces a database of library version fingerprints. Given the Podspec file of a specific version of a library in CocoaPods, LibKit follows three steps for generating its library version fingerprint: *building*, *feature extraction*, and *fingerprinting*.

Building. Given a target library version in CocoaPods, LibKit builds a *template app* that includes it. The template app contains the bare minimum code to include the target library version with all its dependencies and build a stand-alone app without any real functionality. Using a template app is needed because CocoaPods does not build stand-alone libraries, but rather includes them into an app's project. Using a template app also allows us to produce the binary files of the library to be analyzed regardless of whether the library –or any of its dependencies– is distributed as source code or pre-compiled as dynamic or static library.

To build the template app, LibKit creates a new Xcode project and produces a *Podfile* for the template app that requires the target library version. The following code shows the *Podfile* created for the Crashlytics 3.14.0 library.

```
platform :ios, '9.0'
target 'templateapp' do
  use_frameworks!
  pod 'Crashlytics', '3.14.0'
end
```

The *Podfile* sets iOS 9 as the target platform. The *use_frameworks!* line tells CocoaPods to try to build the app dependencies as Frameworks. However, some libraries such as Crashlytics are distributed as pre-compiled .a archives, forcing Xcode to link them statically into the app's main binary. LibKit then uses CocoaPods to install the target library version in the *Podfile* into the app's project. CocoaPods includes the target library version and all its dependencies into the build script of the template app project. The CocoaPods installation produces a *Podfile.lock* file that resolves the dependencies in the *Podspec* of the target library version into concrete library versions to be included into the app's project.

Next, LibKit leverages Xcode to build the template app. Since the template app only includes the library, but does not really use it, LibKit uses Xcode's debug mode to build it, which disables dead code elimination. This guarantees that the produced template app contains the complete target library version and its dependencies. If the building process is successful, the template app will comprise of a main Mach-O binary (*templateapp*) and another binary for each library built as a Framework. As a last step, the built binaries and the *Podfile.lock* file are processed to generate a *build log*, which captures a mapping from each executable to the library versions it corresponds to. It also includes a dependency tree with the dependencies of the target library version declared in the *Podspecs* of all installed libraries. The dependency tree will miss vendored libraries since those are not declared in the *Podspecs*.

Feature extraction. The feature extraction process takes as input the built binaries and the build log. For each Mach-O binary, it uses the dsdump parser [15] for extracting the list of classes it contains. During this process it filters out the skeleton classes that are known to belong to the template app using a static class blacklist. For each

```
@interface NSDuck : NSObject {
BOOL flying;
}
- (void)quackWithVolume:(int)volume;
@end
```

(a) Objective-C source class.

```
class_name : NSDuck
class_language : objective-c
methods: {
    method_name : quackWithVolume
    method_type : class_method
    method_interface : void,int }
variables: {
    objc_ivar_name : flying
    objc_ivar_type : bool }
```

(b) Extracted features.

Figure 4: Objective-C feature extraction example.

remaining class, 4 properties are extracted: *class name*, *class language* (Swift or Objective-C), the *list of instance variables* (only for Objective-C classes), and the *list of class methods*. For each class method, 3 properties are extracted: *method name*, *method type* (class or instance method), and its *interface* (for Objective-C methods, for Swift methods the interface is encoded in the name). The interface of a method is a mangled string that defines the *return type* and the *type of its parameters*. For each Objective-C instance variable, 2 properties are extracted: *name* and *type*. Figure 4a shows a sample Objective-C class *NSDuck* with one method named *quackWithVolume*. Figure 4b shows the features extracted for that class.

Class fingerprints. A class fingerprint is a 64-bit Simhash value [9]. Simhash is a similarity hash used in many fingerprint-based detection techniques because of its ability to generate similar fingerprints for similar objects. Specifically, it takes as input a set of hashes and produces a fixed size hash value with the property that similar inputs produce similar hash values (i.e., with low Hamming distance). Simhash can approximate the Jaccard index between two sets of hash values: a small Hamming distance between two Simhash values indicates a high Jaccard index, and a large Hamming distance indicates a low Jaccard index.

In our case, the input to the Simhash function is variable number of 64-bit FNV1a hashes. One hash covers the concatenation of the class name, class language, number of instance variables , and number of methods. An additional hash is produced for each Objective-C method, for each Swift method, and for each Objective-C instance variable. The hash of each method and instance variable includes the class name to avoid spurious matches with unrelated classes that may have similarly named methods and variables.

In Figure 4, the class fingerprint for the *NSDuck* class would be the 64-bit Simhash of three hashes: one for the class name (NSDuck), language (Objective-C), number of methods (1), and number of instance variables (1); another one for the quackWithVolume method; and a final one for the flying instance variable.

Class fingerprints capture class and method metadata. They do not consider the code of the methods. The advantage of this design is that it can identify the same class regardless of code changes due to different compiler configurations. This is important because LibKit generates the class fingerprints using compiler settings that may not match those used by the developers of the apps that use the library. One disadvantage is that two versions of the same class with identical metadata but containing code differences e.g., a patch that only adds a NULL pointer check in one method, will have the same class fingerprint and thus cannot be differentiated. Another disadvantage is that our fingerprints are not resilient to symbol renaming. However, Wang et al. [37] identified only 0.06%

of one million iOS apps collected from the official iTunes Store as being obfuscated using symbol renaming. While obfuscation is an important problem, it is not an urgent issue in today's iOS landscape, and thus we plan to address it in future work.

The use of Simhash allows us to identify the same class despite small changes in its metadata. The kind of changes that we want to allow are those we would normally expect between minor versions of a library. For example, the addition, removal, or renaming of methods or attributes of a class would not greatly impact the class fingerprint, while large changes like heavy refactoring, class renaming, or the addition of a large number of methods would produce significantly different class fingerprints. This is important because the generated database of libraries may not be complete. For example, the database may include some versions, but not all versions, of a library. In this scenario, LibKit should still identify that the app contains the library. While it cannot identify the correct library version since it is not in the database, it aims to identify the closest version of the library that is in the database. We evaluate library version identification in Section 5.2.

Library version fingerprint. A library version fingerprint comprises of a set of *class fingerprints*, one for each of the library classes. More specifically, the library version fingerprint comprises of the set of class fingerprints for all classes in all binaries produced when building the template app. For example, if the template app for a target library version comprises of two binaries (the main templateapp binary and a Framework), then the library version fingerprint of that library version would comprise a set of Simhashes whose size is the sum of the number of classes in both binaries (minus those classes in the blacklist).

It is possible for multiple library versions to have the same fingerprint, which makes them indistinguishable from each other. This happens for two main reasons: (i) close versions of the same library that have identical class metadata and only differ in their code; (ii) libraries that are forks or exact clones of each other. Once the database has been populated, LibKit identifies the library versions with the same fingerprint and puts them in an equivalence class. For each equivalence class, it identifies a *leader*. The role of the leader is to be the library version output in the results. The leader selection counts how many times a library version depends on another in the equivalence class. The library with the highest number of dependants is selected as the leader. An alternative would be to output all library versions in the equivalence class.

3.2 Library Detection

The bottom part of Figure 3 describes the library detection architecture. It takes as input an app's IPA and outputs the list of library versions the app uses. The first step is decrypting the app, which is explained in Section 4.2. After decrypting the app, the next step is, for each decrypted binary, to parse it, extract the class and method features, and generate the class fingerprints, as described in Section 3.1. The class fingerprints are input to the *candidate search*, whose task is to find for each class fingerprint in the app, a set of similar class fingerprints in the database, which we call candidates.

Candidates that do not pass certain selection criteria are removed. Depending if the binary being processed is the main app binary or a Framework, the *candidate filtering* or *framework filtering* take care of the removal. The *candidate selection* then identifies the best matching candidate libraries. Finally, the *dependency inclusion* adds the dependencies between the detected library versions, selects a leader for each equivalence class, and produces the final list of library versions identified.

Candidate search. Each binary in the app $\mathcal{A}$ is composed of a set of class fingerprints. For each class fingerprint c in the app's binary, the candidate search function $\mathcal{S}$ returns as candidate v^l any library in the database containing at least one class fingerprint f more similar than a *similarity threshold* T_s to the class fingerprint c in the app.

$$\mathcal{S}(c) = \{v^l : (v^l, f) \in \mathbb{D}, simil(c, f) \geq T_s\}$$

The *simil* function is the normalized similarity between two class fingerprints, one coming from the app (c) and the other from a library in the database (f). Since c and f are 64-bit Simhash values, LibKit computes their similarity based on the Hamming distance of their bitstrings, thus $simil(c, f) = \frac{64-hamming(c,f)}{64}$.

The similarity threshold T_s captures the minimum similarity for two class fingerprints to be considered a match. In Section 5, we empirically determine that $T_s = 0.8$ gives optimal results.

Candidate filtering. The candidate search selects any library version that has at least one class in common with the app. Thus, it may return hundreds or even thousands of candidates. It makes little sense, though, to consider a library if the number of classes found in the app is a very small ratio of the whole library (e.g., the app contains only 1 class of the 100 in the candidate library). This case is more likely a fingerprint collision, which might happen for very small classes with very few features.

The second step therefore filters out from this list of candidates the libraries that do not match a significant part of the classes. For each v^l identified during the previous step, LibKit keeps only the ones that satisfy $\frac{\mathcal{L}(v^l)-\mathcal{M}(v^l)}{\mathcal{L}(v^l)} \geq T_m$, being $\mathcal{L}(v^l)$ the number of class fingerprints the library version has and $\mathcal{M}(v^l)$ the number of classes in v^l that could not be found in $\mathcal{A}$ by name. This filtering removes library versions that match very few classes of the total classes the app lib contains, since it is unlikely that dead code elimination would remove the majority of the classes. In Section 5, we empirically evaluate different T_m values and select $T_m = 0.35$.

Thus, the candidate filtering function as follows:

$$\mathcal{F} = \{v^l : v^l \in \mathcal{S}(c), c \in \mathcal{A}, \frac{\mathcal{L}(v^l) - \mathcal{M}(v^l)}{\mathcal{L}(v^l)} \geq T_m\}$$

Framework filtering. Libraries compiled as Frameworks have their name in the path of the application bundle. For all library versions distributed as source, the database keeps a mapping between the library version and the Framework binaries produced during its processing in the library fingerprint generation. When the binary under analysis is a Framework, LibKit thus directly extracts its name from the path and matches it against the mapping. LibKit keeps in $\mathcal{F}$ only those library versions that match. If no library candidate matches, instead, LibKit does not remove any library versions from $\mathcal{F}$, and keeps it as is. Thus, if a Framework was renamed, no library candidate will match it, and the detection will proceed without this optimization.

Candidate selection. $\mathcal{F}$ contains library version candidates that may cover class fingerprints in common with other candidates. However, at the end of the library detection phase each class fingerprint found in the app should be assigned to at most one library version candidate. Keeping in mind that Frameworks contain only one library per binary, while the main binary may contain many statically linked libraries, beside the app code, LibKit resorts to two different algorithms to select the best match.

In both cases, LibKit first ranks the list of filtered library version candidate $\mathcal{F}$ according to a score $p(v^l)$. This score is computed as a multiplication of three values. The first value is the *coverage score*, defined as the number of classes the candidate matches times the ratio of classes the candidate matched in app $\mathcal{A}$ and the total number of classes in the library version fingerprint. The second value is the *average normalized similarity* of the classes matched by the candidate. The last value is the number of different versions of the library the candidate contained in $\mathcal{F}$, which we call the *popularity* of the library. The intuition behind this *popularity* score is that since close versions of a library do not differ much, they will have similar library version fingerprints. Thus, it is common that many versions of the libraries that most likely are the best matching ones appear in $\mathcal{F}$.

$$p(v^l) = \frac{|C(v^l)|^2}{\mathcal{L}(v^l)} \times \overline{simil}(v^l) \times pop(v^l)$$

In the case of the main binary, LibKit solves the set cover problem in a greedy fashion, as shown in Algorithm 1. *cov* at line 2 is initialized with all the classes in the app that need to be covered by a library candidate in $\mathcal{F}$. Starting from the candidate with the highest $p(v^l)$, LibKit removes from *cov* the classes that v^l covers, it adds v^l to the selected libraries, and finally removes from $\mathcal{F}$ the libraries that do not cover any remaining class in *cov*. This process continues until the list in $\mathcal{F}$ is empty. When the binary is a Framework, instead, LibKit first picks the candidates with the best *coverage score*. From this subset it picks the candidate with the best *average normalized similarity*, and adds it to the list of selected libraries.

Algorithm 1 Algorithm for selecting candidates in the main binary

```
1: procedure SolveMainBinary($\mathcal{F}$)                ▷ $\mathcal{F}$ is sorted by $p(v^l)$
2:     $cov \leftarrow \{c : c \in \mathcal{A}, \mathcal{S}(c) \neq \emptyset\}$
3:     $removed \leftarrow \emptyset$
4:     $selection \leftarrow \emptyset$
5:     for $v^l \in \mathcal{F}$ do
6:         if $v^l \in removed$ then
7:             continue                                   ▷ Ignore $v^l$
8:         end if
9:         if $cov = \emptyset$ then
10:            break                                      ▷ We are out of classes to cover
11:        end if
12:        $selection \leftarrow selection \cup \{v^l\}$
13:        $cov \leftarrow cov \setminus C(v^l)$
14:        for $c \in C(v^l)$ do
15:            $removed \leftarrow removed \cup \mathcal{S}(c)$
16:        end for
17:    end for
18:    return $selection$
19: end procedure
```

Table 1: Library database summary.

Item	Count
Libraries	14,043
Library versions	86,597
Classes	8,714,001
Objective-C classes	63.7%
Swift classes	36.3%
Methods	106,993,407

Dependency inclusion. Finally, for each selected library, LibKit reports all its dependencies in the final list of libraries found in the app, since these are implicitly included in the bundle.

4 DATASETS

To evaluate our approach we need libraries and apps, as well as ground truth to evaluate accuracy. In Section 4.1 we describe how we build fingerprints for a large number of libraries available through CocoaPods. In Section 4.2 we describe how we collect popular apps from the iTunes Store. Finally, in Section 4.3 we describe how we build a ground truth for a small number of apps to evaluate the library detection.

4.1 Library Database

The seeds for the library database construction were the 188,129 library versions belonging to 17,302 libraries targeting iOS 9 or later available in the CocoaPods repository on April 28th, 2021. iOS 9 is the lowest SDK our Xcode version supports. We managed to compile 99,877 (53.1%) of those library versions. The most common compilation failures were that a dependency targeted iOS 8 and that the library used an old Swift version (e.g., Swift 3) that our Xcode version no longer supports. We were able to extract fingerprints from 86.7% of the compiled versions (46% of the original seeds). The most common failure to build a fingerprint was pure C libraries because they do not contain classes. Table 1 summarizes the produced library database, which contains 86,597 library versions belonging to 14,043 libraries. The total number of classes is 8,714,001, of which 63.7% are Objective-C classes and 36.3% Swift classes. Those 8,714,001 classes contain 106,993,407 methods, an average of 10.2 methods per class.

4.2 App Collection

Our app collection pipeline replicates the one proposed by CriOS [31]. It downloads apps by instrumenting the iTunes Windows client, installs them on an iPhone device, and dumps the memory using Frida [17] after decryption has completed. We use the app collection pipeline to download and decrypt 1,500 randomly chosen apps from the list of popular apps in the Italian market of the iTunes Store. The download took 1.2 days using a single Windows VM and the decryption rate was 100 apps/hour using two devices.

4.3 Ground Truth

We have created two ground truth datasets: one with 43 apps for which we have identified the library versions they use (GT43) and another with 95 apps for which we have only identified the names of the libraries they use, but not their version (GT95).

GT43. To build our ground truth datasets, we started by searching GitHub for open-source iOS apps, which yielded 140 apps. To obtain the desired ground truth for an app, we need to obtain its Podfile.lock file, which states the library versions that CocoaPods included when building the app. For example, if the app's podfile defines a range of compatible library versions, the Podfile.lock file will state the specific version CocoaPods chose in that range (i.e., the one LibKit should detect). Unfortunately, Podfile.lock is only available when building the app from source, or if the developers commited it to the app's source repository after building the app. It is not available for proprietary apps in the iTunes Store, even if they were built using CocoaPods.

Compiling an iOS open-source app is most often a painful process due to limited documentation and the need to set up required dependencies. Thus, we only managed to compile 10 of the 140 apps. In addition, another 33 apps (disjoint from the 10 we managed to compile) had their Podfile.lock file available in their repository. For these 43 apps we know the library versions CocoaPods included in the app. However, the app could also include some vendored libraries, i.e., the library source was copied into the app's source and thus is compiled as part of the app. Podfile.lock does not contain vendored libraries since CocoaPods is not aware of those. To identify vendored libraries, we manually examined the app's source code in the repository. Overall, the 43 apps contain 511 library versions belonging to 347 libraries, with a total of 26,160 classes. We use GT43 to measure how accurately LibKit detects library versions.

GT95. Among the 140 apps, there were 95 apps (the 33 with a Podfile.lock in their repository and another 62 not in GT43) that were available in the iTunes Store. This allowed us to download the most recent app version at the time from the iTunes Store. For these 95 apps, we cannot get the library versions used as we do not have access to their Podfile.lock. However, we can parse the podfile in the source repo to identify the names of the TPLs the app uses. Since the podfile is needed to build the open-source app, the developers generally add it to the repository, in contrast to the optional Podfile.lock. Note that the podfile is only available for open-source apps using CocoaPods, but not for proprietary apps in the iTunes Store. Thus, it cannot be used for library detection. For these 95 apps we only know the names of the libraries the apps use, but not the library version. We use GT95 to measure how accurately LibKit detects libraries. The 95 apps contain 1,066 libraries with 37,283 classes.

This process shows that building a ground truth of library versions an app uses is challenging, even for open-source apps. Unfortunately, we could not select the apps we wanted for the ground truth, but rather the GT datasets include all apps for which we could obtain data. However, we believe the resulting datasets are representative as they contain apps of different sizes ranging from small (e.g., KeePassium) to very large (e.g., Firefox).

5 EVALUATION

Our evaluation addresses the following research questions: **RQ1:** What is the accuracy of LibKit for detecting libraries? **RQ2:** How does LibKit compare to the state of the art? **RQ3:** What is the accuracy of LibKit for detecting library versions? **RQ4:** What libraries does LibKit detect on apps from the iTunes Store?

5.1 RQ1: Library Identification

To evaluate the accuracy of LibKit for detecting libraries, we run LibKit on GT95 multiple times using different parameters. For each run, we compare the identified libraries (ignoring the identified version) with the list of libraries in the GT. In particular, we compute the number of detected libraries in the GT (True Positives), the number of detected libraries not in the GT (False Positives), and the number of undetected libraries in the GT (False Negatives). We split the false negatives into those due to libraries present and not present in our library database, so that we can separate database coverage from the accuracy of the detection. From these values we derive three standard metrics: *Precision*, *Recall*, and *F1 score*.

Library detection has two parameters: the similarity threshold (T_s) and the coverage threshold (T_m). To identify optimal values for these parameters we run the library detection varying their values in the ranges $0.8 \leq T_s \leq 0.95$ and $0.2 \leq T_m \leq 0.5$, in increments of 5%. The best results are achieved using a similarity threshold $T_s = 0.8$. Interestingly, the T_m coverage threshold does not affect the results within these value ranges. This is due to the greedy algorithm selecting candidate libraries that cover the most, so the coverage threshold only applies to whatever remains after high coverage classes have been selected. Based on this evaluation, we select $T_s = 0.8$ and $T_m = 0.35$ as parameter values for the rest of the evaluation.

Using those parameter values, we first measure the accuracy excluding the impact of the library database, i.e., excluding FNs of libraries not in our database. The library detection results are: *Precision* of 0.911, *Recall* of 0.839, and *F1 score* of 0.874. We next estimate the impact of the library database coverage. In GT95, 248 (22%) of the libraries are not in CocoaPods. These libraries are distributed through other means. To cover such libraries we could incorporate other library sources such as GitHub. Another 164 (15%) libraries are in CocoaPods, but LibKit could not generate a fingerprint for them. Covering these libraries would require improvements to our automated build pipeline, e.g., supporting older iOS frameworks. If we include FNs from missing libraries the library detection results are: *Precision* of 0.911, *Recall* of 0.524, and *F1 score* of 0.666. The impact of database coverage is significant, despite our library database being the largest of its kind with 86K library versions. In contrast, the largest database in Android TPL detection works contains 12K library versions [40].

Even when including these FNs, the results from LibKit are comparable to the best Android tools. In particular, Zhang et al. [41] evaluated the accuracy of 5 Android TPL detection tools, observing that most tools achieve high precision (i.e., above 0.84) but all have low recall (i.e., below 0.50). In their evaluation, the best library identification tool was LibScout with a *precision* of 0.974, *recall* of 0.490, and *F1 score* of 0.652, slightly worse than LibKit.

False negatives. Next, we analyze the false negatives where the library is present in our database. More than half of these FNs are limitations on our GT that does not capture dependencies due to vendored libraries. For example, one app includes the UserExperior library, which embeds SSZipArchive as a vendored library. Our GT generation correctly identified the vendored library and both libraries are in the GT, but the GT does not capture the dependency between them. LibKit correctly identifies UserExperior, but does not identify SSZipArchive because it is (correctly) considered part of UserExperior. Thus, SSZipArchive is counted as a FN, despite the detection being arguably correct. Adding dependencies to the GT is in our future work plan, but it requires significant manual work. The similarity threshold is behind 33% of the FNs. In these cases, the highest similarity between a class from the library in the app and the corresponding library class in the database is below 0.8. One explanation for these cases is that our database contains far away versions of the library compared to the version the app uses, so the similarity is low. Reducing the similarity threshold could help remove some of these FNs, but it would introduce FPs and hamper scalability by largely increasing the number of candidates. The coverage threshold is behind 8% of the FNs. These are cases where the library in the app does not cover at least 35% of the full library in the database. These may be due to aggressive dead code elimination while building the app, or again to low coverage of the library versions in our database.

False positives. A common reason for false positives are actually false negatives, what we call FN-FP pairs. It often happens that when LibKit misses a library, it introduces a FP by selecting a different library that contains the missing library. For example, one app uses the Reachability library. LibKit misses Reachability and instead flags geekSDK, which vendors Reachability. In this case, LibKit is correctly capturing Reachability, but the fact is hidden in the results, introducing both a FN and a FP. Another case, responsible for 48% of FPs, are small libraries in the database that may contain very few classes (often only one). Apps may contain similar small classes creating collisions that make the small library to be included in the results. We also found a handful of cases caused by the usage of other languages in app development. For example, there are two React Native apps that contain the React Native runtime, which is not in our database. Instead, LibKit identifies a particular library that contains the React Native runtime. This case is analogous to a FN-FP pair, but with a FN due to a library not in the database.

Our analysis confirms results by Zhang et al. [41] that identified library dependencies as one of the most challenging aspects of TPL detection.

5.2 RQ2: Comparison with State of the Art

We compare LibKit against CRiOS, which we consider the state of the art in iOS TPL detection. The approach by Tang et al. [35] is specific to network libraries, and also requires dynamic analysis. And, the approach by Chen et al. [10] only applies to libraries released for both Android and iOS. We do not try porting any Android approach to iOS because most use the Android-specific package structure. ORLIS [38] is the only Android approach that operates solely on classes, but it has been shown to perform worst amongst publicly available Android tools [41], so it does not seem worth the porting effort.

CRiOS is a clustering-based approach that takes as input a set of apps and identifies groups of classes that appear together in more than one app, share a name prefix, and have class cohesion. For each app, it outputs the unlabeled clusters of apps, where each cluster supposedly represents a TPL. Since CRiOS does not output library names and versions, the comparison focuses on the identification of the class boundaries between the TPLs and the app classes.

Table 2: Comparison between LibKit and CRiOS for identifying the class boundaries of TPLs on the GT95 apps.

	All Apps			Pure Objective-C Apps		
Tool	**Prec.**	**Recall**	**F1**	**Prec.**	**Recall**	**F1**
CRiOS	92.6%	18.4%	30.7%	87.9%	35.9%	51.0%
LibKit	94.0%	58.6%	72.2%	92.9%	70.5%	80.1%

GT95 contains the TPLs in each app and the list of app classes belonging to each TPL, which can be used as a reference clustering. We can compare this reference clustering to the clustering produced by CRiOS using external clustering validity metrics [20], which compare the input clustering to the reference one based only on the cluster agreement, **without** considering the cluster labels. In particular, we use a version of Precision, Recall, and F1 often used in malware clustering approaches [6, 33]. Other external validity metrics like Rand Statistic, Jaccard Coefficient, and Folkes and Mallows Index could equally be used. We can similarly compare LibKit to the reference clustering since its output includes the list of app classes belonging to each identified TPL.

Since the release of CRiOS, many changes have happened in iOS, which cause CRiOS to fail on 72 of the GT95 apps (45 apps that contain Swift and 27 apps using newer Objective-C versions). To address this issue, we modified CRiOS to replace its *class-dump* Mach-O parser with the *dsdump* parser LibKit uses. With this fix we could run CRiOS on all apps in GT95.

Table 2 reports the comparison results over all apps in GT95, as well as only on the 50 pure Objective-C apps since CRiOS was not designed to support Swift. The results show that LibKit beats CRiOS in both datasets in all metrics. The recall for CRiOS is very low as it misses Swift libraries, libraries that only appear in one app, and classes that appear in one library version but not in another version of the same library. When removing apps that use Swift, CRiOS results improve, but the F1 score of LibKit remains 29 points higher. Thus, LibKit clearly improves on CRiOS for identifying TPL boundaries in an app. Furthermore, LibKit is able to automatically label the libraries (and versions) in the app.

We also run CRiOS on the 1,500 apps collected from the iTunes Store. CRiOS outputs 5 clusters in two applications. In one app, it outputs 3 clusters that correspond to the Crashlytics, Firebase, and Answers libraries. In the other app, it outputs 2 clusters that correspond to the Fabric and AdMob libraries. For the other 1,498 it outputs no clusters, i.e., it does not identify any TPLs. In contrast, Section 5.4 shows that on the same 1,500 apps, LibKit identifies 47,015 library versions. The results indicate that as the number of apps increases, it becomes increasingly hard for CRiOS to identify TPLs in the input apps.

5.3 RQ3: Library Version Identification

To evaluate the detection of library versions we run LibKit on the GT43 dataset using the parameter values determined in RQ1. LibKit detects 421 library versions: 370 that exactly match the one

Table 3: Top 10 most prevalent libraries detected among the 47,015 different library versions identified

Library	Publisher	Apps	Ver.
Google-Mobile-Ads-SDK	Google	494	78
GoogleUtilities/AppDelegateSwizzler	Google	493	44
FBSDKCoreKit	Facebook	445	46
PromisesObjC	Google	413	23
Protobuf	Google	271	27
Crashlytics	Google	236	25
FirebaseInstanceID	Google	232	25
GoogleDataTransport	Google	216	30
Alamofire	Alamofire	198	20
GoogleAnalytics	Google	183	6

in the GT (TPs) and 51 that differ from the GT (FPs). It misses 84 library versions (FNs). Thus, the accuracy metrics for library version detection are: *Precision* of 0.879, *Recall* of 0.815, and *F1 score* of 0.846.

Zhang et al. evaluated three Android tools for library version identification, observing that the best one was LibScout with an F1 score of 64.47% in a dataset of similar versions and 22.25% on another dataset where versions had higher differences. Thus, LibKit achieves a higher F1 score on version identification than the best-performing Android tool.

5.4 RQ4: Library Detection on Store Apps

We run the library detection on the 1,500 apps collected from the iTunes Store. LibKit detects 47,015 library versions across all 1,500 apps. Of those, 31,041 (66%) are statically linked (or vendored), while the remaining 34% are Frameworks. This highlights the utility of LibKit for the analysis of iOS apps. Simply examining the name of the Framework files in the App bundles would miss two thirds of the libraries. In addition, for the Frameworks, LibKit is able to provide the version, which is not part of the Framework name. Among the 47,015 detections, there are 11,704 unique library versions (i.e., 13.5% of those in our DB) and 1,592 unique library names (11.3% of those in our DB). Each app has a median of 25.5 libraries, mean of 26.4, with an standard deviation of 19.9. In 131 apps, LibKit did not detect any libraries and the largest number of libraries detected in one app is 106.

Table 3 shows the top 10 most prevalent libraries identified on the 1,500 apps. Of those, 8 are published by Google, one by Facebook, and another one by the Alamofire Software Foundation. The table highlights Google's efforts to have a wide presence in the iOS advertisement ecosystem since iOS has a 27% mobile OS market share [29]. The most popular library is Google's AdMob advertising library, present in 494 apps, Second is the *AppDelegateSwizzler* subspec of the *GoogleUtilities* library present in 493 apps. Third comes *FBSDKCoreKit*, which allows to integrate Facebook into an iOS app and is found in 445 apps.

The rightmost column in Table 3 shows that the apps contain a large number of versions for the same library. Except for GoogleAnalytics, the number of unique versions for each library ranges from 20 in Alamofire up to 78 for AdMob. Given that the apps were collected within 1.2 days, this indicates that many apps were running old versions of the libraries. One example is the *Documents (Office Docs)* app by Savy Soda, with 4.6K ratings in the Italian market. The version we crawled was 12.2, released on August 28th, 2019. However, among the 8 library versions LibKit detects, five were released between 2014 and 2018. For example, LibKit detects *FirebaseInstanceID* 1.0.10, which was released on 2017, two years earlier than the app release. LibKit also detects *OneDriveSDK*, a library that Microsoft has recently deprecated and has not received code updates since February 2019. None of the recent app releases (latest from February 2022) mention the deprecation of the library in their release notes. Furthermore, the app only gets 1-4 updates each year over the last five years and stayed 1.5 years without an update around our collection date. These results highlight that old library versions are common among popular apps in the iTunes Store and how LibKit can be used to identify apps with such outdated dependencies.

Runtime. On average, LibKit takes 8 minutes to detect the libraries in an app. Zhan et al. [41] measured the runtime of five publicly available Android TPL detection tools. Compared with those, LibKit would be slower than LibRadar (6 seconds on average per app) and LibScout (1.4 minutes), but faster than ORLIS (24 minutes), LibPecker (5.11 hours), and LibID (23.12 hours). Similar to our number, those numbers do not include the clustering time for LibRadar or the library generation time for the other tools. Note that Zhan et al. used a database of 59 libraries and 2,115 library versions for all tools except LibRadar. In contrast, LibKit's database is 40 times larger, comprising of 14,043 libraries and 86,597 library versions. This negatively affects LibKit in the comparison, as the detection time for all tools is at least linear on the size of the database [41].

6 RELATED WORK

TPL detection approaches for mobile apps can be split between clustering-based approaches that infer libraries by identifying code components shared by multiple input apps (e.g., [24, 28, 31, 43]) and those that build a database of library fingerprints directly from the library code (e.g., [4, 5, 36, 38, 40, 42]). Clustering-based approaches have the advantage that they can identify previously unknown TPLs, but they cannot name the specific library or library version a cluster represents, unless a separate mapping is built (typically manually). Also, they may fail to identify niche or emerging TPLs. Furthermore, they have been shown to have problems separating versions of the same library and identifying partially included libraries (e.g., due to dead code elimination) [41]. To avoid those limitations, our approach builds fingerprints from known libraries. To address the problem of library database coverage we leverage the repository of the CocoaPods PM, unique to the iOS ecosystem, which allows us to build the largest library version database so far with 86K libraries versions from 14K libraries.

The majority of TPL identification approaches focus on Android apps and cannot be applied to iOS because they leverage the Android-specific package structure. The only Android TPL detection work that operates at the class-level is ORLIS [38], but previous work has shown ORLIS to perform worst amongst publicly available Android tools [41]. A few works address TPL identification on iOS apps. CriOS [31] first proposed a clustering-based approach that groups classes with the same name prefix and class cohesion into libraries. As part of their approach to identify vulnerabilities in iOS network services, Tang et al. [35] proposed a clustering-based technique that dynamically obtains the callstack of an app when it invokes the *bind* function and identifies the TPL wrapping bind by grouping similar callstacks. However, their approach only applies to network libraries. A different approach was used by Chen et al. [10] who applied TPL detection to Android apps and then built constant string invariants of the Android TPLs to identify them in iOS apps. Unfortunately, their approach only applies to libraries available both in Android and iOS. The limitations of clustering-based approaches and the fact that the only Android approach that could be ported to iOS had low accuracy motivated us to design a novel iOS TPL detection approach.

Several works have proposed iOS analysis techniques for other applications. PiOS [16] statically identifies the methods used by an Objective-C binary to detect privacy violations. Joorabchi and Mesbah [22] apply image recognition for detecting when the user interface of an iOS app has changed during execution. Other works identify vulnerabilities in iOS apps such as those introduced by the misuse of cryptographic APIs [25] and credentials [39]. Other works study iOS dependencies in CocoaPods [13, 32]. Another research line compares the Android and iOS platforms. Some studies focus on the development model [12, 18, 19]. Others compare the security and privacy features and issues of both platforms [1, 2, 7, 10, 14, 21, 23, 30].

7 DISCUSSION

This section discusses limitations and avenues for improvement.

Fingerprint uniqueness. Library versions with the same fingerprint (i.e., same class metadata) cannot be differentiated and are added to an equivalence class. Those cases are largely due to consecutive library versions, with roughly two-thirds due to consecutive patch-level versions, and another 25% due to consecutive minor versions. We also observe our fingerprints producing higher collisions in Swift libraries compared to Objective-C libraries. This is due to compiled Swift code containing less information than compiled Objective-C code, e.g., it does not contain instance variables, which are only included in our fingerprints for Objective-C.

To make the fingerprints more unique we could add more features extracted from the binary code, e.g., from functions and variables that do not belong to classes and from the code of the methods and functions. There are two main reasons why we left such features for our future work. First, there is limited tooling for analyzing iOS native code. Popular binary analysis platforms such as Angr [3] have little support for iOS and Mach-O binaries, e.g., to enable disassembly, function identification, CFG reconstruction, and type inference. We may need to implement much of that analysis ourselves, a very significant endeavor on top of the work we already performed. Second, some problems where code analysis would be very useful such as handling obfuscation are still not prevalent in iOS, i.e., Wang et al. [37] showed only 0.06% of apps in the iTunes Store are obfuscated. It is also worth noting that additional code features would not solve all collisions since we observe consecutive library versions with identical code where changes happen only in data files or in auxiliary code not included in the app.

Database coverage. While code analysis would help to distinguish some close library versions, our results show that low recall is mostly due to limited coverage in the library version database. This happens even if our library version database comprises 86K library versions, seven times larger than those used in Android TPL detection. The limited coverage is due to multiple reasons including libraries not distributed through CocoaPods, library versions that we could not compile, and limitations of the analysis (e.g., lack of support for old Swift versions). In future work, we plan to increase the coverage of our library database by incorporating GitHub projects that use other package managers like Carthage and Swift PM and examine if they use any libraries not present in CocoaPods. We also plan to improve our automated compilation pipeline to increase the successful compilation rate.

Ground truth size. One threat to the validity of the library version identification results is the limited size of GT43 with 511 TPLs present in 43 open-source apps. It is possible that a larger ground truth would reveal harder cases that would lower the overall library version identification accuracy. As explained in Section 4, building library version ground truth is painful since it requires successfully compiling the apps and manually examining their source code to identify vendored libraries. We are releasing our ground truth to foster future work in the area and plan to explore approaches to automate the ground truth construction in future work.

8 CONCLUSION

We presented LibKit, a fully automated TPL identification tool that is the first to identify the name and version of TPLs present in iOS apps. LibKit supports apps developed in Swift, Objective-C, or a combination of both; detects statically and dynamically linked libraries; and addresses the challenges of partially included libraries, and that different compiler versions and configurations can produce different outputs for the same library. LibKit automatically builds fingerprints for 86K library versions available through CocoaPods and matches them on the decrypted app executables. We evaluate LibKit for the problems of library identification and library version identification, showing that its accuracy positively compares with the best performing Android tools. LibKit also significantly outperforms CRiOS, the previous state-of-the-art tool for iOS apps, for the problem of detecting TPL class boundaries in iOS apps.

As future work we would like to address some of the limitations of LibKit such as adding support for code written in other languages (e.g., C/C++); strengthening the fingerprints against obfuscation by incorporating instruction-level or CFG-level features; increasing the coverage of our library database; and increasing the success rate of the compilation pipeline (e.g., by supporting older iOS versions). We would also like to perform a user study to quantify the benefits for human analysts when using LibKit.

ACKNOWLEDGMENTS

This work was funded by the Madrid regional government program S2018/TCS-4339 (BLOQUES-CM) and by the Spanish Governement MCIN/AEI/10.13039/501100011033/ERDF through grants TED2021-132464B-I00 (PRODIGY) and PID2022-142290OB-I00 (ESPADA). Those projects are co-funded by European Union ESF, EIE, and NextGeneration funds.

DATA AVAILABILITY

We have made available the code, database, and ground truth required to replicate this research at [26] and [27].

REFERENCES

[1] Mohd Shahdi Ahmad, Nur Emyra Musa, Rathidevi Nadarajah, Rosilah Hassan, and Nor Effendy Othman. 2013. Comparison between android and iOS Operating System in terms of security. In *CITA 2013: 8th International Conference on Information Technology in Asia*. IEEE, 1–4.

[2] Fattoh Al-Qershi, Muhammad Al-Qurishi, Sk Md Mizanur Rahman, and Atif Al-Amri. 2014. Android vs. iOS: The security battle. In *WCCAIS 2014: World Congress on Computer Applications and Information Systems*. 1–8.

[3] angr 2022. angr. https://github.com/angr/angr.

[4] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 356–367.

[5] Salman A Baset, Shih-Wei Li, Philippe Suter, and Omer Tripp. 2017. Identifying Android library dependencies in the presence of code obfuscation and minimization. In *ICSE 2017: Proceedings of the 39th International Conference on Software Engineering*. 250–252.

[6] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, Behavior-Based Malware Clustering. In *Network and Distributed System Security*.

[7] Zinaida Benenson, Freya Gassmann, and Lena Reinfelder. 2013. Android and iOS users' differences concerning security and privacy. In *CHI 2013: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 817–822.

[8] carthage 2022. Carthage Dependency Manager. https://github.com/Carthage/Carthage.

[9] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. 380–388.

[10] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *IEEE S&P: 2016 IEEE Symposium on Security and Privacy*. 357–376.

[11] cocoapods 2022. CocoaPods Dependency Manager. https://cocoapods.org/.

[12] Daniel Domínguez-Álvarez and Alessandra Gorla. 2019. Release Practices for iOS and Android Apps. In *WAMA 2019: Proceedings of the 3rd International Workshop on App Market Analytics*. 15–18.

[13] Daniel Domínguez-Álvarez, Alessandra Gorla, and Juan Caballero. 2022. On the Usage of Programming Languages in the iOS Ecosystem. In *SCAM 2022: Proceedings of the 22nd IEEE International Working Conference on Source Code Analysis and Manipulation*.

[14] Daniel Domínguez-Álvarez, Alessandra Gorla, Juan Caballero, and Roberto Giacobazzi. 2019. Are you Sure They are the Same? Identifying Differences Between iOS and Android Implementations. In *Actas de las V Jornadas Nacionales de Ciberseguridad*. 332–333.

[15] dsdump 2022. dsdump. https://github.com/DerekSelander/dsdump.

[16] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications.. In *NDSS 2011: 18th Annual Symposium on Network and Distributed System Security*. 177–183.

[17] frida 2022. Frida Instrumentation Toolkit. https://frida.re/.

[18] Mark H Goadrich and Michael P Rogers. 2011. Smart smartphone development: iOS versus Android. In *SIGCSE 2011: Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. 607–612.

[19] Tor-Morten Grønli, Jarle Hansen, Gheorghita Ghinea, and Muhammad Younas. 2014. Mobile application platform heterogeneity: Android vs Windows Phone vs iOS vs Firefox OS. In *AINA 2014: 28th IEEE International Conference on Advanced Information Networking and Applications*. 635–641.

[20] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. 2001. On clustering validation techniques. *Journal of intelligent information systems* 17, 2 (2001), 107–145.

[21] John Hubbard, Ken Weimer, and Yu Chen. 2014. A study of SSL proxy attacks on Android and iOS mobile applications. In *CCNC 2014: 11th IEEE Consumer Communications and Networking Conference*. 86–91.

[22] Mona Erfani Joorabchi and Ali Mesbah. 2012. Reverse engineering iOS mobile applications. In *WCRE 2012: 19th Working Conference on Reverse Engineering*. 177–186.

[23] Konrad Kollnig, Anastasia Shuba, Reuben Binns, Max Van Kleek, and Nigel Shadbolt. 2022. Are iPhones Really Better for Privacy? A Comparative Study of iOS and Android Apps. *PETS* 2022, 2 (2022), 6–24.

[24] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: Scalable and Precise Third-party Library Detection in Android Markets. In *ICSE 2017: Proceedings of the 39th International Conference on Software Engineering*. 335–346.

[25] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2015. icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications. In *NSS 2015: Proceedings of the 2015 International Conference on Network and System Security*. 349–362.

[26] libkit 2023. LibKit release URL. https://doi.org/10.5281/zenodo.7042015.

[27] libkitWP 2023. LibKit Website. https://sites.google.com/view/libkit.

[28] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In *ICSE 2016: Proceedings of the 38th International Conference on Software Engineering*. 653–656.

[29] mobileStats 2022. Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide.

[30] Ibtisam Mohamed and Dhiren Patel. 2015. Android vs iOS security: A comparative study. In *ITNG 2015: 12th International Conference on Information Technology-New Generations*. 725–730.

[31] Damilola Orikogbo, Matthias Büchler, and Manuel Egele. 2016. CRiOS: Toward large-scale iOS application analysis. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. 33–42.

[32] Kristiina Rahkema and Dietmar Pfahl. 2022. Dataset: Dependency Networks of Open Source Libraries Available Through CocoaPods, Carthage and Swift PM. In *MSR 2022: 19th International Conference on Mining Software Repositories*. 393–397.

[33] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. 2011. Automatic Analysis of Malware Behavior using Machine Learning. *Journal of Computer Security* 19, 4 (2011).

[34] swiftpm 2022. Swift Package Manager. https://www.swift.org/package-manager/.

[35] Zhushou Tang, Ke Tang, Minhui Xue, Yuan Tian, Sen Chen, Muhammad Ikram, Tielei Wang, and Haojin Zhu. 2020. iOS, Your OS, Everybody's OS: Vetting and Analyzing Network Services of iOS Applications. In *USENIX Security: 29th USENIX Security Symposium*. 2415–2432.

[36] Dennis Titze, Michael Lux, and Julian Schuette. 2017. Ordol: Obfuscation-Resilient Detection of Libraries in Android Applications. In *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 618–625.

[37] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. 2018. Software Protection on the Go: A Large-Scale Empirical Study on Mobile App Obfuscation. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 26–36.

[38] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. 2018. ORLIS: obfuscation-resilient library detection for Android. In *MobileSoft 2018: Proceedings of the 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems*. 13–23.

[39] Haohuang Wen, Juanru Li, Yuanyuan Zhang, and Dawu Gu. 2018. An Empirical Study of SDK Credential Misuse in iOS Apps. In *APSEC 2018: Proceedings of the 25th Asia-Pacific Software Engineering Conference*. 258–267.

[40] Jian Xu and Qianting Yuan. 2020. LibRoad: Rapid, Online, and Accurate Detection of TPLs on Android. *IEEE Transactions on Mobile Computing* 21, 1 (2020), 167–180.

[41] Xian Zhan, Tianming Liu, Yepang Liu, Yang Liu, Li Li, Haoyu Wang, and Xiapu Luo. 2021. A Systematic Assessment on Android Third-party Library Detection Tools. *IEEE Transactions on Software Engineering* (2021).

[42] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. 2018. Detecting third-party libraries in android applications with high precision and recall. In *SANER 2018: 25th IEEE International Conference on Software Analysis, Evolution, and Reengineering*. 141–152.

[43] Zicheng Zhang, Wenrui Diao, Chengyu Hu, Shanqing Guo, Chaoshun Zuo, and Li Li. 2020. An Empirical Study of Potentially Malicious Third-Party Libraries in Android Apps. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*.

Received 2023-02-02; accepted 2023-07-27