Reliable Third-Party Library Detection in Android and its Security Applications

Michael Backes
CISPA, Saarland University &
MPI-SWS
Saarland Informatics Campus

Sven Bugiel
CISPA, Saarland University
Saarland Informatics Campus

Erik Derr
CISPA, Saarland University
Saarland Informatics Campus

Abstract

Third-party libraries on Android have been shown to be security and privacy hazards by adding security vulnerabilities to their host apps or by misusing inherited access rights. Correctly attributing improper app behavior either to app or library developer code or isolating library code from their host apps would be highly desirable to mitigate these problems, but is impeded by the absence of a third-party library detection that is effective and reliable in spite of obfuscated code. This paper proposes a library detection technique that is resilient against common code obfuscations and that is capable of pinpointing the exact library version used in apps. Libraries are detected with profiles from a comprehensive library database that we generated from the original library SDKs. We apply our technique to the top apps on Google Play and their complete histories to conduct a longitudinal study of library usage and evolution in apps. Our results particularly show that app developers only slowly adapt new library versions, exposing their end-users to large windows of vulnerability. For instance, we discovered that two long-known security vulnerabilities in popular libs are still present in the current top apps. Moreover, we find that misuse of cryptographic APIs in advertising libs, which increases the host apps' attack surface, affects 296 top apps with a cumulative install base of 3.7bn devices according to Play. To the best of our knowledge, our work is first to quantify the security impact of third-party libs on the Android ecosystem.

1. INTRODUCTION

Third-party libraries have become a fixed part of mobile apps. Developers use them to, e.g., monetize their apps through advertisements, integrate their apps with online social media, include single-sign-on services, or simply leverage utility and convenience libraries for their apps' functionality.

However, third-party libraries are a double edged sword: While they can provide convenience for the app developer and can greatly enhance their host apps' features, they also have been shown to be a hazard to the end-users' privacy and security. A number of prior studies [10, 14, 4, 30, 34] has demonstrated that such libraries exhibit questionable privacy practices. For instance, they leak user-private information, exploit their host app's privileges, or track users. Two recent incidents of such questionable practices were revealed in the popular SDKs of Taomike—China's biggest mobile ad provider—and Baidu, that were found to be secretly spying on users and uploading their SMS to remote servers [37] and opening backdoors to the users' devices [35], respectively. In addition to such privacy violations, third-party libs increase the attack surface of their host apps when they do not adhere to security best practices and, hence, become a liability for the users' security. In the recent past, even popular libraries by reputable software companies, such as Facebook and Dropbox, were affected by highly severe vulnerabilities. The found vulnerabilities could lead to the leakage of sensitive data to publicly readable data-sinks [26], code injection attacks [28, 39], account hijacking [36], or linking a victim's device to an attacker-controlled Dropbox account [8].

Given the high prevalence of third-party libraries in apps and consequently their high impact on the health of the entire smartphone ecosystem, it is of no surprise that dedicated research has investigated new mechanisms to sandbox or remove libs, with a strong focus on advertisement libs [31, 27, 44, 30]. Yet, these proposals make one crucial assumption that currently limits them in their effectiveness: they assume that libraries can be clearly identified, either through developer input [27] or inspection of the app's code [31, 44, 30]. *Reliably* identifying libraries, however, forms a formidable and yet unsolved technical challenge. First, third-party libraries are tightly integrated into their host app by statically linking them during the app's build process into the app's bytecode, thus blurring the boundaries between app and library code. Second, app developers commonly make use of bytecode obfuscation tools, such as ProGuard [16]. One side-effect-free bytecode obfuscation technique is identifier renaming. It turns identifiers into short, non-meaningful strings, i.e. a package name *com.google* is transformed into *a.c.* Naïve library detection approaches based on identifier matching, like in [34, 14, 31, 44] or applied by third-party ad detector apps, fail even to this simple obfuscation technique.

Another problem, caused by the inability to reliably detect third-party libraries within applications, is the lack of accountability for privacy and security violations. For instance, a wide range of security-related analyses studied apps for privacy and security issues and raised awareness for various problem areas, including privacy leaks [12, 2, 42, 13, 3], permission usage [43], dynamic code loading [28], SSL/TLS

CCS '16, October 24–28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978333

(in-)security [25, 11], or (mis-)use of cryptographic APIs [9]. However, without being able to distinguish app developer code from third-party library code, the reported results are on a *per-app* basis and do not distinguish whether bad or improper behavior originates from app or library developers.

To increase efficiency of library sandboxing mechanisms and to be able to hold the correct principal (app or lib developer) accountable for security and privacy violations, a reliable and precise third-party library detection is required that is resilient against common obfuscation techniques.

Our contributions.

In this paper, we make two tangible contributions: First, we present an efficient and reliable approach for detecting third-party libraries within Android apps (see Section 4). By analyzing the original library SDKs, we extract profiles that are resilient to common obfuscation techniques, such as identifier renaming and API hiding. To achieve these properties our approach is based on class hierarchy information only and is independent of the libraries' code. Still, our profiles are fine-grained enough to not only detect distinct libraries, but also the exact version used in an app. For the actual library detection, we devised a profile matching algorithm that reports whether an exact copy of a given library version was matched. In the negative case, either because the correct version is missing in our database or dead-code elimination was applied to app code, a similarity score indicates the best matching profile for the library code in the app.

Second, we use our library detection technique in a longitudinal study of third-party libraries included in the top apps on Google Play (see Section 6). In this study, we are interested in finding answers to security- and privacy-related questions about libraries, such as *"How prevalent are third-party libraries in the top apps and how up-to-date are the library versions?"*, *"Do app developers update the libs included in their apps and how quickly do they update?"*, or *"How prevalent are vulnerabilities identified in prior research [28, 9] in libraries and how many apps are affected?"* To answer these questions, we first built a comprehensive repository of third-party libraries and applications (see Section 5). Our library set contains 164 libraries of different categories (Advertising, Cloud,..) and a total of 2,065 versions. We then collected and tracked the version histories for the top 50 apps of each category on Play between Sep 2015 and July 2016, accumulating to 96,995 packages from 3,590 apps. We complemented this database with meta-information, such as app and library release dates, which we collected from public sources or developer websites. Based on this data set, we show that app developers commonly neglect updates of third-party libraries. By analyzing the time-to-fix of two recent security/privacy incidents of the Facebook and Dropbox SDKs [36, 8], we show that this developer negligence in updating libraries exposes end-users to large windows of opportunities for attacks (e.g., on average 190 days for 51 apps with a vulnerable Facebook SDK in our data set). Lastly, we scan our library set for presence of API misuse vulnerabilities [28, 9] that would expose the libs' host apps to cryptanalytic and code injection attacks and discover 18 vulnerable libs (61 versions), which together affect 296 apps with a cumulative install base of 3.7bn. Overall, our work constitutes the first longitudinal security study of third-party libraries in the Android ecosystem. We provide the first valuable insights into the security-impact of libraries and as such motivate future work on improving library updatability on Android. In summary, we make the following contributions:

1) We are the first to devise a light-weight and effective approach (LibScout[1]) to detect third-party libraries in Android apps that is resilient to common obfuscation techniques and capable of pinpointing exact library versions.

2) We created a large third-party library database including 164 distinct libraries with 2,065 versions, which we profile. We collected the version history of 3,590 top apps on Play for a total of 96,995 distinct packages. We complement those databases with meta-data such as app/library release dates.

3) We conduct a longitudinal study of third-party libs in our app set to investigate their prevalence, the update frequency of apps and of libs, as well as the impact of app popularity and library API stability on the lib update frequency.

4) We study time-to-fix and vulnerability windows of apps that include vulnerable library versions (at the example of recently reported incidents of the popular Facebook and Dropbox SDK). Our results show large windows of opportunity for attackers against apps including those libraries.

5) Lastly, we re-apply existing app analysis techniques to library code to investigate improper usage of dynamic code loading and crypto APIs. We identify 61 library versions that affect 296 top apps on Play with a cumulative install-base of 3.7bn users and expose these apps to cryptanalytic attacks.

2. RELATED WORK

Code and app clone detection techniques for Android apps have been studied in many different respects. Prior work computed the similarity between apps using code-based similarity techniques [5, 17, 48] or by extracting semantic features from program dependency graphs [6, 7].

Other approaches are tailored to third-party libraries on Android, e.g., by employing the concept of whitelisting package names to detect libraries within app code [14, 4, 5]. As such approaches fail to cope with even simple obfuscation techniques such as identifier renaming, more robust approaches based on machine learning or code clustering have been investigated. *AdDetect* [24] and *PEDAL* [20] use machine-learning to detect advertising libraries. AdDetect uses hierarchical package clustering to detect (non-)primary modules of apps whereas PEDAL extracts code-features from library SDKs and uses package relationship information to train a classifier to detect libraries even when identifiers are obfuscated. *AnDarwin* [7] and *WuKong* [40] detect app clones with high accuracy by filtering library code that is detected by means of code clustering techniques. Such approaches rely on the assumptions that libraries are pervasively used by many apps, and that app developers do not modify the library. However, this second assumption is unrealistic, since automatic/manual dead-code elimination during app building will necessarily modify the library code[2]. Moreover, these approaches only provide binary classifications since they cannot name the concrete library versions used within the apps. The recent *LibRadar* [22] extends WuKong's clustering approach and generates unique profiles for each detected cluster. Profiles are generated from the frequency of API calls within distinct packages in a cluster and can subsequently be used for fast library detection. With this approach, *LibRadar*

[1] https://projects.cispa.uni-saarland.de/derr/libscout

[2] https://developer.android.com/studio/build/shrink-code.html

	Allatori [32]	dashO [29]	DexGuard [15]	DexProtector [18]	DIVILAR [47]	ProGuard [16]	Stringer [19]
API hiding (*)	–	–	✓	✓	–	–	–
Class encryption	–	–	✓	✓	–	–	–
Control-flow randomization (*)	✓	✓	–	–	–	–	–
Identifier renaming (*)	✓	✓	✓	✓	–	✓	✓
String encryption (*)	✓	✓	✓	✓	–	–	✓
Virtualization-based protection	–	–	–	–	✓	–	–

Table 1: Feature comparison of Android app obfuscators. Our approach is robust against features marked with (*).

was able to find 29K potential libraries on a large corpus of Google Play apps. This number presumably constitutes an over-approximation, since the original code clustering and the subsequent feature extraction are not performed on the original libraries. This lack of ground truth produces false positives when multiple libraries have the same root package (e.g. `com.google` for the various Google Play Service libraries and Google libs like Gson/Guice). To avoid such heuristics, we extract our profiles from the original library binaries. This comes at the cost of completeness but has several advantages such as less false positives and the possibility of inferring exact library versions. In addition, this allows computation of more reliable similarity values for partial library inclusions (as a result of code optimizations).

Besides library detection, research has also proposed techniques for privilege separation between host apps and advertising libraries. To this end, *AdSplit* [31] puts library code into separate processes, while *PEDAL* [20] uses an inline reference monitoring approach to allow users to selectively enable/disable functionality in libs that requires a permission. *AdDroid* [27] uses a system-centric approach and proposes a new ad-API in Android's application framework for app developers to allow privilege separation by construction. As a more rigorous approach, *APKLancet* [44] removes malware and ad library code from an app package. The code to be removed is identifed through semantic fingerprints that have previously been extracted from malware/library samples.

A separate line of work studied questions related to security and privacy in advertising libraries. Stevens et al. [34] investigate the permission (mis-)use of ad libs. Book et al. [4] conducted a longitudinal study of ad lib permissions and discovered that the absolute number of required permissions increases over time. However, in contrast to our longitudinal studies, they estimated the library release dates as the terminus ad quem of the release date of an app that includes the library. Library profiles are generated by hashing the library code detected via package name matching. In this case, the detection only works if the original library code is included without any modifications. Other approaches analyse private data exfiltration [14, 33] and security vulnerabilities in authentication/authorization SDKs [41]. In Section 7, we re-apply existing app security analyses to library code and use LibScout to measure the real-world impact of the discovered vulnerabilities in our app set.

3. REQUIREMENTS ANALYSIS

Version detection & similarity computation. Related work on library detection largely strived for completeness by reducing this problem to code similarity between apps. While this works well for detecting components within an app, such approaches always suffer from uncertainty as they are not based on the ground truth in form of the original library code. Moreover, this lack of ground truth precludes a more fine-grained detection including inferrence of the concrete library version. This information, however, is imperative to conduct longitudinal studies on the Android ecosystem to analyze library updatability, compatibility, and identifying vulnerable versions.

Therefore, we base our approach on the original library code provided as SDK by the library provider. Although relying on the original libraries comes with the drawback of incompleteness, particularly for less prevalent libraries, it is a necessary trade-off to infer concrete library versions, which would not be possible without ground truth. In addition and in contrast to common belief, libraries are often *not* included as is but in some modified form. In most of the cases this refers to bytecode optimization through dead-code elimination of unused library functionality. To reliably detect such partially incomplete code fragments in apps, ground truth is required to compute a similarity value between library code in the app and original lib versions. To this end, we build a comprehensive library database (see Section 5) including the library binary and complementary information, such as library name, version, and release date (if available).

Robustness against common obfuscation techniques. A widely-used third-party library detection technique is naïve package name matching. While package name matching provides a rough estimation of the different components within an app, this approach comes with several drawbacks. In the presence of code obfuscation, such as the commonly used identifier renaming, detection fails since the original package names no longer exist. Some libraries, such as the advertising lib *airpush*, even use a similar technique to defeat such naïve detection methods. To get an overview about available obfuscation techniques for Android other than identifier renaming, we studied popular obfuscation tools and their capabilities. A high-level feature comparison is presented in Table 1 (features like application water-marking or resource file encryption are not relevant for our analysis and therefore omitted). The summary shows that code-based detection approaches additionally would have to cope with different kinds of code-obfuscation techniques, such as API hiding or control-flow randomization. Ideally, a detection approach should be resilient to all presented obfuscation techniques. However, depending on the analysis technique (static or dynamic) inherent limitations apply.

Our approach is based on static analysis, hence dynamic code loading or class encryption to dynamically create code at runtime are general limitations. Similarly, virtualization-based protection, i.e., dex bytecode is replaced by a virtual instruction set that is interpreted by a custom virtual machine [47], is out of scope. To still handle the most common obfuscation techniques (marked with (*) in Table 1), our approach solely relies on class hierarchy information and does

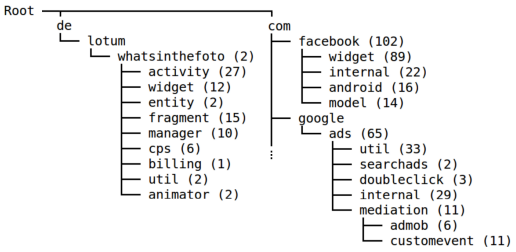

Figure 1: Generated partial (unobfuscated) package tree of an app. Numbers denote the classes per package.

not depend on the actual library code. Since integrity of the library code (e.g., piggybacking malware) is not a concern of this paper we do not face drawbacks from the design decision to rely on class hierarchy information.

The next section gives detailed information on our lightweight detection approach that fulfills these requirements, while Section 6 evaluate our prototype in terms of accuracy, memory requirements, and performance.

4. APPROACH

The workflow of our approach consists of two separate steps. We first extract profiles from any original library version in our repository. Given this set of profiles we statically detect libraries in apps by extracting app profiles and subsequently apply a matching algorithm to check whether and how libraries match. Profiles are generated from class hierarchy information only and do not rely on concrete library code. This is necessary to be robust against code-based obfuscation techniques such as control-flow randomization or API hiding. Our profile matching algorithm reports a *similarity score* between 0 and 1, indicating whether a library version was matched exactly, partially or not at all for a given application. The remainder of this section provides detailed information on the underlying data structures for the profile generation and the actual profile matching algorithm.

4.1 Package Tree

Java packages are a technique for organizing classes into namespaces. Packages are defined using a hierarchical naming pattern with levels in the hierarchy separated by dots. Packages that are lower in the hierarchy are usually referred to as subpackages. By convention, package names are written in lower case and companies should use their reversed Internet domain name as leading package, i.e. `google.com` uses a package name `com.google.*`. App and library developers usually stick to this convention or at least provide a namespace that is unlikely to appear in a different software component. This simplifies library integration in which lib code is statically linked during the app's build process.

The structure of the package hierarchy (often depicted as a tree) therefore gives a rough estimation on the included libraries in an application. Meta-information such as package relationships (parent, sibling, child/subpackage) and number of classes per package are not part of our profiles but will be used to improve accuracy of our matching algorithm. Figure 1 shows a partial, un-obfuscated package tree of the app `de.lotum.whatsinthefoto`. It includes two third-party libraries in the `com` namespace—the Facebook SDK and Google Admob—and the app code in the `de` namespace.

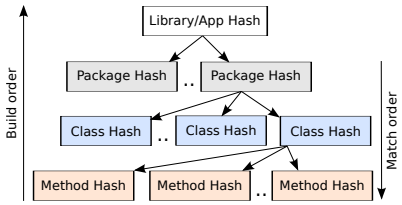

Figure 2: Merkle tree with a fixed depth of three

We generate the package tree by performing a standard class hierarchy analysis (CHA). For each application class we parse its package, i.e. for a class `com.google.ads.AdView` we receive an array of package fragments `[com, google, ads]`. Starting from the root node, tree nodes are traversed in the order of the package fragment array. Non-existing child nodes are generated on-demand. Once the array is processed the class counter of the current tree node is incremented. This tree representation is used for debugging purposes and to perform structural checks during profile matching.

4.2 Profile Extraction

For the actual profiles we use a variant of Merkle trees [23]. In these hash trees every non-leaf node is labeled with the hash of its child nodes. Our tree has a fixed depth of three where layers represent packages, classes and methods (see Figure 2). In contrast to the package tree, our Merkle tree is flattened at the package layer, i.e. each distinct package is a child of the root node. This allows an efficient and precise package-based comparison between original libraries and applications to be tested.

In general, Merkle trees are used to verify contents of large data structures. To this end, the initial hashes in the leaf nodes are generated by hashing a piece of data, e.g. the content of a file. However, relying on actual code makes our approach susceptible to code-based obfuscation such as API hiding or control-flow randomization. Therefore, the method hashes have to be computed from non-obfuscatable information. For that we use pruned method signatures. A signature is a string that uniquely identifies a method within an app. Figure 3 shows an example signature of method `openActiveSession` in class `Session` of package `com.facebook`. The method name is followed by the argument list in brackets and the return type. In a first step we remove anything but the argument list and return type which is called *descriptor*, since any information before the argument list may be subject to identifier renaming. In the final step, we replace any non-framework type (framework types can be looked up in the Android SDK) by the same placeholder identifier `X` that can not serve as a type. This *fuzzy descriptor* keeps any non-obfuscatable information, but different application types are no longer distinguishable. The advantage is that we do not have to record a global type mapping but this also implies that a single fuzzy descriptor may match other methods as well. However, the introduced fuzzyness for a single method is compensated by including all methods of a particular class. The higher the number of children (e.g. methods, classes) for a specific node, the smaller is the probability that two different nodes with the same number of children match.

To provide a deterministic hash generation for non-leaf nodes, child hashes are sorted and concatenated before being

```
      signature: com.fb.Session.openActiveSession(android.app.Activity,bool,com.fb.Session$StatusCallback)com.fb.Session
     descriptor:                                 (android.app.Activity,bool,com.fb.Session$StatusCallback)com.fb.Session
fuzzy descriptor:                                (android.app.Activity,bool,X)X
```

Figure 3: Transforming a method signature to an fuzzy identifier that is robust against identifier renaming

re-hashed. For the hashing, we found that the 128-bit MD5 hash algorithm provides a good trade-off between efficiency and precision for our use-case. As default, we store the hash tree excluding the method layer to retrieve space-efficient profiles. If maximum precision is required, method hashes can be stored with their original signatures. In addition, we introduce a `publicOnly` mode in which only public methods of public classes are considered during tree generation. While the resulting profile is less unique, it is, at the same time, robust against changes of the internal API. We use such profiles to check library API compatibility in Section 6.

Bytecode normalization.

Before building the hash tree of a component (library or app), we normalize its bytecode, in particular we remove anything compiler-generated. This refers to bridge and synthetic methods such as `accessor` methods in nested classes with private attributes that are accessed by the enclosing class. This way we focus on developer-written code only and abstract from concrete compilers (usually `javac` for packaged libraries and `dx` for apps).

Some libraries have other library dependencies (e.g. *OkHttp* requires *Okio* for some I/O classes). In apps those dependencies are resolved through static linking. If we analyze *OkHttp* in isolation, any type specified in *Okio* will not appear in the class hierarchy. For generating a fuzzy descriptor this is not a problem as we can treat such (non-existing) dependencies like normal library code. Hence, the application and library profile will not differ in this regard.

4.3 Profile Matching

Our matching algorithm tests whether and how a given library profile matches an app profile. To this end, for each library profile a *similarity score* between zero and one is computed, one indicating an exact library match. In contrast to profile extraction, matching is performed top-down in the Merkle tree. For testing whether a library *exactly* matches parts of the application profile it suffices to check whether all library package hashes are included in the hash tree of the app. It becomes more complicated if an application only partially matches the library code, e.g. if the app includes a library version that is not in the library database or only parts of the original library are included (as result of a dead-code elimination). This implies that only a subset of package hashes matches and the similarity score drops below one. The higher the score is, the better a given library version is matched. If exact matching fails, the similarity score is computed at a deeper tree level, either on class or method level depending on the desired precision. We define the similarity score on class level between a library package `lp` and an app package `ap` as follows:

$$score_c(lp, ap) = \frac{\#\text{ classes in lp that match in ap}}{\#\text{ classes in lp}} \in [0, 1]$$

This definition tolerates the addition of classes, i.e. the score does not change if the application package has more classes than the library packages. However, if the app package contains less classes the similarity score will be smaller than one. Given that package hashes may no longer match, the question is which app packages should be compared to which library packages. A naïve approach would exhaustively compute the similarity score for any library/app package combination and take the global maximum, i.e. the set of best matching app packages. This might introduce false positives when matched app packages do not have the same root package and/or the package hierarchy is not preserved. For two library packages {`com.google`, `com.google.ads`} valid candidate packages include {`a.b`, `a.b.d`} but neither {`a.b`, `a.c`} nor {`a.b`, `c.d`}. While `a.b` might be a valid candidate for `com.google`, the combination with `a.c` is invalid as the package hierarchy is no longer preserved. In the second invalid example, candidates do not have the same root package (`a`/`c`). To overcome this problem we apply the following four-step approach:

1) Candidate list. We first compute for each library package (lp) a list of candidate app packages (ap). An app package is a candidate if at least 50% of its class hashes match (configurable). An example could look like this:

$$lp_1\text{: } ap_1(0.95),\ ap_2(0.84),\ ap_3(0.75)$$
$$lp_3\text{: } ap_6(0.91),\ ap_4(0.60)$$
$$lp_2\text{: } ap_7(0.85),\ ap_9(0.82)$$

Every candidate list is sorted by score. In addition, library packages are sorted by similarity score of their highest candidate match.

2) Package linking. If a library package lp_1 with package name `com.foo` has app package candidates starting with the same package name, we can remove candidates with different root packages. In case identifier renaming has been applied to the app code, this direct linking no longer works. For filtering invalid combinations we therefore have to identify potential root packages. If lp_1 matches ap_1 with package name `a.b.c` we deduce that `a.b` is one potential library root package within the app. By applying this to all pairs $<lp_i, ap_j>$ we receive a list of potential root packages.

3) Partitioning. Instead of exhaustively testing all combinations like in the naïve approach, it suffices to compute the maximum for each partition/root package and then take the global maximum. For each root package we pre-filter the candidate list and remove any candidate that does not start with the current root package. For the remaining list the maximum is computed exhaustively via backtracking. To eliminate combinations that do not structurally match the library package hierarchy, we define an abort criterion by testing structural equivalence between app packages and library packages. More formally, backtracking is aborted if the following requirement is violated:

$$\forall ap_i, ap_j, lp_x, lp_y,$$
$$ap_i\ candidateOf\ lp_x,\ ap_j\ candidateOf\ lp_y,$$
$$relationship(ap_i, ap_j) = relationship(lp_x, lp_y)$$

$relationship(p_1, p_2)$ tests whether p_1 is parent, sibling, or child of p_2 and in case of a parent/child relationship it further determines the package distance (an immediate sub-

package has distance 1). If, for example, the backtracking algorithm traverses ap_1, ap_6 the calculation is aborted if $relationship(ap_1, ap_6) \neq relationship(lp_1, lp_3)$.

4) Global maximum. Finally, we select the maximum score over the partitions and sum up the matched classes (denoted as sum-classes). The *similarity score* on class level is consequently computed as

$$simScore_c = \frac{\text{sum-classes}}{\text{\# of classes in library}} \in [0, 1]$$

We classify a library as partially matched if the score exceeds a minimum threshold such as 0.6. This value was determined experimentally to find a good trade-off between false-positives and false-negatives since a low similarity score might either result from dead-code removal (partial library inclusion) or from a library version detected that is not in the database. To increase precision, the similarity score can also be computed on method level if method hashes are included in the profiles.

5. LIBRARY AND APP REPOSITORY

In this section, we explain how we established a database of third-party libraries and of applications, which we use to evaluate our LibScout tool (Section 6) and on which we conducted a longitudinal study of third-party libraries used in the top apps of Google Play (Section 7).

5.1 Library Database

The foundation of our approach is detecting known libraries in Android apps. This requires setting up a library database that contains the ground truth in the form of original code packages for each available library version.

Identification and retrieval of popular libs.

The first task to build a library database is to identify popular libraries, such as libs that are specifically developed for Android (e.g., ad and analytics libs) or Java support/utility libraries. In particular advertising libraries are one of the most prevalent library types for Android. Google's developer documentation on AdMob mediation networks[3] gives a good but incomplete view on available ad libraries that are compatible to its own AdMob library. Another major source of library statistics is provided by the Android third-party market *AppBrain*[4]. It provides an *Ad Detector* app to gather statistics about the market share of popular libs. Libraries are identified by checking for well-known identifiers, such as package names, and are categorized into the three groups (ad networks, social SDKs, and development tools). In addition to such readily-available information, we manually analyzed the package trees (cf. Figure 1) of 50 popular apps to identify additional libraries based on package names.

Following this first bootstrapping step, we retrieve the identified library binaries and, if possible, their complete history, since our approach relies on the ground truth in the form of the original library code. We found that there are different ways how library developers distribute their SDK. More and more libraries can be found on the maven central repository or are hosted on public GitHub repositories. In these cases it is trivial to retrieve the entire history. Other libraries, such as the Facebook SDK, are hosted directly at the library provider's website and might have migrated to Maven (as is the case for the Facebook SDK). Early versions of some libraries were distributed as open-source only (without pre-compiled binaries), hence it took some effort to compile each version with varying build environments. It gets more complicated if developer accounts are required to download a specific library (this is common for advertising libraries like Tapjoy or Flurry). Moreover, only the most current versions of some libraries were available. To still retrieve older versions, tricks like URL modification or searching for lib versions in known Android projects were required.

For each library version we store the binary code, name and auxiliary information like version number and release date, which are usually available via change log or directly from the host server. Moreover, we categorize each library in one of the following groups: Advertising, Analytics, Android, Cloud, Social-Media, and Utilities.

[3]developers.google.com/admob/android/mediation-networks
[4]http://www.appbrain.com/stats/libraries/

Category	# Libraries	# Library Versions
Android	51	560
Utilities	41	746
Advertising	40	337
Cloud	16	149
SocialMedia	9	149
Analytics	7	124
Total	164	2,065

Table 2: Number of distinct libs and versions per category

Library Statistics.

Our current database contains 164 distinct libraries with 2,065 versions. Table 2 shows the distribution of library/-versions across the categories. The database includes the most popular libraries for each category. For about 26% of all libraries we got less than four versions, however, at the same time the database contains more than seven versions for 55% of the libraries. The mean number of versions per library is 12.59±1.09. For the advertising library Heyzap we were able to collect 96 distinct versions. For 2,026 (98.11%) of the library versions in our database we were able to collect their release dates. From those release dates, we derive that the developers of the third party libraries in our data set release a new version on average every 117 ± 60 days and just in the first half of 2016 on average every 77 ± 20 days.

On a commodity laptop the average time for profile extraction is 2.8 seconds per lib version. The mean number of packages, classes, and methods in our library set is <13, 304, 1701>. This even exceeds the code base of many smaller apps. Outliers include the Google Play Service library with <85, 3,416, 18,794>. Those results show that many libraries are very-complex and/or offer a lot of functionality.

5.2 Play Store Crawler and Repository

Next, we provide an overview of how we selected our sample applications from the Google Play Store and built a longitudinal version history of those selected apps.

Upper Bound for Version Code.

We bootstrapped building our set of sample applications and their history by crawling the top 50 apps for each of the 20 categories on the Google Play Store in 2-hour intervals between September 2015 and July 2016, resulting in 4,666 distinct apps, which we continue to track even after they left

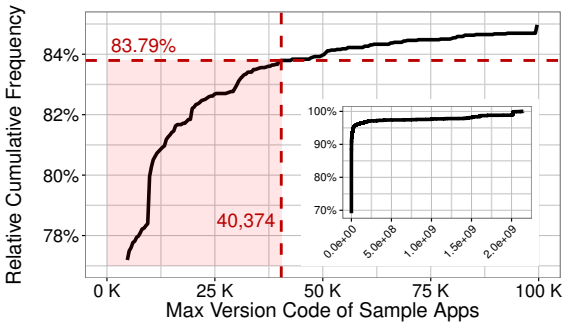

Figure 4: Distribution of maximum version codes in our initial app set and selected threshold for our crawler.

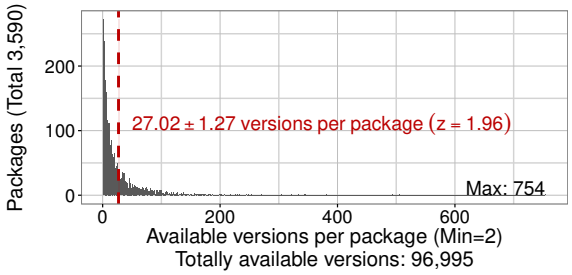

Figure 5: Sampled version codes and version codes per app.

the top 50 lists. We opted for this approach, because prior studies [38, 46] have established that the Google Play Store is a "superstar" market in which a small percentage of the free applications (i.e., the top apps) account for almost all of the downloads. Thus, our sample set represents the apps with the largest user bases on Play—together accounting for almost 46 bn downloads by July 2016 according to Play.

To build the version history for every discovered application, the Google Play API can be iteratively queried for lower version codes. For instance, when our initial app set contained the application package *org.wikipedia* with version code *10*, we can iteratively request to download versions *9, 8, 7, etc.* of this app package from the server. Unfortunately, the versioning rules[5] for Android apps do not require app developers to follow any specific scheme except that the version code must be monotonically increasing between updates. This implies that version codes can be distributed all over the integer value range. As a consequence, we have to determine a reasonable upper bound for the version code to achieve a good trade-off between coverage of package version histories and the required time to build the history. Figure 4 illustrates the relative cumulative frequency distribution of maximum version codes in our app set. While most developers choose their version codes from the lower end of the possible value range, some developers choose version codes from within the range of millions to billions, accounting for the long tail of version codes (see the subplot in Figure 4).

For our study, we decided to create histories for apps at the lower end with a maximum version code of 40,374, providing a coverage of 3,910 apps (83.79%) of all apps in our set. Moreover, it is noticeable, that the long tail of version codes has a jump in the CFD around version code $2 * 10^9$. This stems from the fact that the developers of 64 apps in our set chose version codes based on the release date (e.g., following the pattern *YYYYMMDDVV*, where *VV* is the revision-per-day). We additionally included those 64 apps from the long tail of version codes, since their code immediately reveals the app version's release date and we built their history by iterating version codes in date-format from the discovered version back to Jan 1, 2012. Those apps increase the coverage by 1.37%, for a total coverage of 85.16%.

Sampled Version Codes.

Figure 5 provides an overview of our application sample set after downloading all available versions for the top apps in the initial set. Overall, we have 96,995 packages for 3,590 distinct apps (excluding apps that have only one version available in our set). This results in an average of about 27 versions per app with a maximum of 754 available versions for the app *com.imo.android.imoimbeta*.

Release Dates and Update Frequency.

As a last step in building the sample set for our longitudinal study of apps, we complemented our sample app database with release dates for each app version. Since Google Play only provides the release date for the most recent version of an app, we collected the release dates of older app versions from market analysis services such as appannie.com, apk4fun.com, and appbrain.com. In total, our database contains the release dates of 75,339 distinct packages (77.67% of 96,995 packages) for the 3,590 apps for which we retrieved older versions, where the release dates range from 12/19/2009 to 07/29/2016.

Based on those release dates, we estimate that the developers of the apps in our sample set release an app update on average every 62±2.94 days, where the average update frequency per app has increased since 2010 (e.g., 38±1.53 days in 2015 and only 29±0.99 days in first half of 2016).

6. EVALUATION OF LIB DETECTION

We implement our approach on top of the WALA framework[1]. Our tool LibScout requires an additional 3.5 kLOC.

6.1 Library Profile Uniqueness

We start by answering the question on how effective our profiles are for distinguishing different library versions and evaluate the memory requirements for storing our profiles.

Since our profiles are generated with class hierarchy information only, it is possible that different library versions have the same profile when only code has changed but no method interfaces of the public or private APIs of the lib. In these cases we still detect the library but report the set of possible versions. For 53/164 (32.3%) of libraries, all versions have unique profiles. For the 2,065 library versions in our repository, we found that 1,225/2,065 (59.3%) of profiles are unique, i.e., we can unambiguously pinpoint the exact library version. About 40% of all profiles are ambiguous, i.e., there exist at least two versions with the same profile. All ambiguous versions occurred in clusters of consecutive versions. Such clusters are expected for minor version updates in which only bugfixes and minor code changes are implemented. The average size of such clusters is 2.77 versions and only two exceptional cases (*Amazon Analytics* and *braintree payments*) exist with a cluster size of 10 in each case. Although we cannot pinpoint the exact library version for

[5] developer.android.com/tools/publishing/versioning.html

#APKs	Ratio	Library	Category
41,518	41.22%	Facebook	Social media
30,310	30.10%	Gson	Utilities
18,026	17.90%	Flurry Analytics	Analytics
16,336	16.22%	Bolts	Utilities
14,229	14.13%	Crashlytics	Android
14,146	14.05%	OkHttp	Utilities
13,758	13.66%	Nine Old Androids	Android
12,201	12.11%	Facebook Audience	Advertising
11,066	10.99%	Picasso	Android
9,694	9.63%	Retrofit	Utilities

Table 3: Top 10 detected libraries in our app repository, excluding Google support and play service libs.

ambiguous profiles, we significantly reduce the search space for post-analyses (e.g., code inspection) to 2–3 candidate versions on average.

The size requirement for storing a single profile is linear to the number of packages, classes, and methods of a library/app and additionally depends on the chosen hash function (128 bit MD5). During our experiments, the precision of our approach did not change with larger hashes. The largest profile was generated for the *Adrally* Ad SDK (v2.2.0) with a size of 220 KB (normal profile without methods). Including methods and full debug info, such as the original method signatures, increases the size to a total of 3.1 MB. However, the average normal profile size is only ≈22 KB.

6.2 Library Prevalence

We apply LibScout with partial matching on our app dataset to study the prevalence of libraries. In addition, we automatically extract the root package for each library and apply a naïve package name detection for cases in which our profile matching does not report a result. Moreover, we use the package matching to validate the detection rate of the profile matching. Table 3 shows the top 10 detected libraries of our repository in absolute and relative numbers. We excluded Google support and Play service libraries as they represent eight of the top 10 libs, with the Android support v4 library leading with about 80%. Moreover, six Play service libraries would have been listed in the top 10, since many developers typically include the complete set of these libraries although not making use of it. The new list is led by the popular Facebook SDK that is included by about 40% of all apps. Further, the list includes advertisement (Facebook Audience) and tracking libraries as well as commonly used utility libraries such as Gson, Bolts, and OkHttp.

On average, we detect 13.1 distinct libraries/app. The app `com.science.wishboneapp` (various versions) contained the most libraries (55). By reporting the numbers for profile-only detection we receive an average of 9.7 libs/app while the naïve approach finds an additional 3.4 libs/app. There are two main reasons for our approach to not detect a library via profile matching. The first reason is an incomplete set of library versions in our database. This particularly applies to advertising libraries that are not publicly retrievable, i.e. it is generally difficult to build a complete library history. The second reason is that code optimization has been applied during an app's build process to remove unused library code. If the app contains less than 60% of the original lib code our partial matching algorithm no longer reports a match. Finally, we checked whether the number of libs/app evolves over time. To this end, we determined the average number of

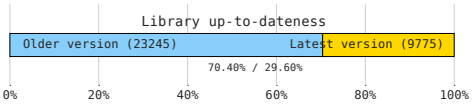

Figure 6: Up-to-dateness of included lib versions across the most recent versions of all apps in our repository.

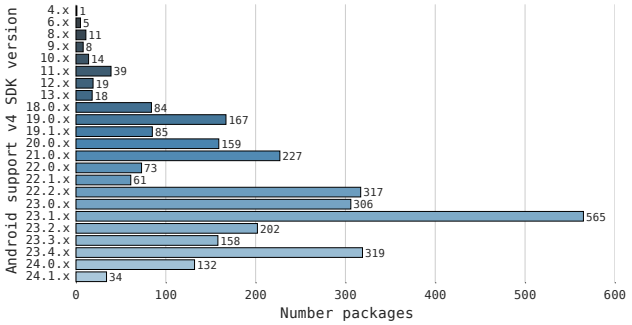

Figure 7: Distribution of Android support v4 SDK versions for the current top apps on Play.

libraries on the set of earliest app releases and most current app releases. Between these two sets, LibScout reports a slight increase of 3.4 on the average library count, i.e. there is a trend to include more libraries.

7. STUDY OF THIRD-PARTY LIBRARIES

Lastly, we conduct a longitudinal study of third-party libraries in our app set to investigate important security-relevant questions such as *"How up-to-date are libraries used in top apps?"*, *"How quickly do app developers react to discovered vulnerabilities in their included libraries?"*, and *"How prevalent is misuse of security APIs in third-party libs?"*.

7.1 Up-to-dateness of Libraries in Top Apps

For the most recent versions of all apps in our data set, we checked whether included third-party libs are up-to-date or outdated with respect to the app release date. Figure 6 summarizes our results. In almost three-quarter of the cases (23,245 lib inclusions or 70.40%) the app developer included an outdated lib version, where the delta to the most recent library version is one in only 7.99% of all cases and in the most extreme case 81 versions. In terms of time difference between library version release and adaption by apps, the apps in our data set required 324 ± 1 days on average to include a new library version. This is a rather poor adaption of newly released lib versions, particularly when considering that app developers release new versions on average almost twice as frequent as lib developers (see Section 5). Thus, we were interested in what potential factors could influence the adaption of a library. In particular, we investigated the library distribution channel, app popularity, and the libraries' public API compatibility.

Distribution channel and app popularity.

A first potential factor is the distribution channel of the library (cf. Section 5). Two popular libraries, Android support v4 and Facebook (see Table 3), are employing opposing strategies: Android support v4 is shipped with Android IDEs

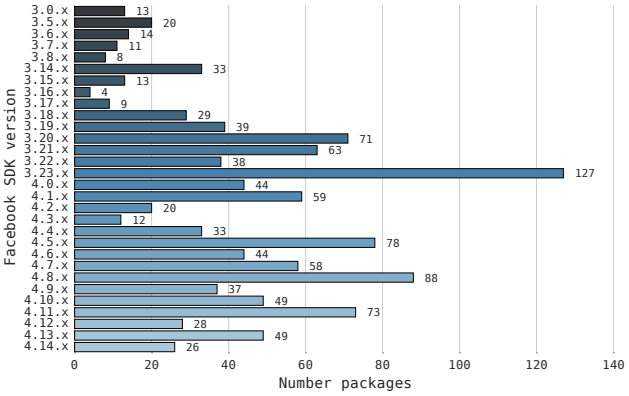

Figure 8: Distribution of Facebook SDK versions for the current top apps on Play.

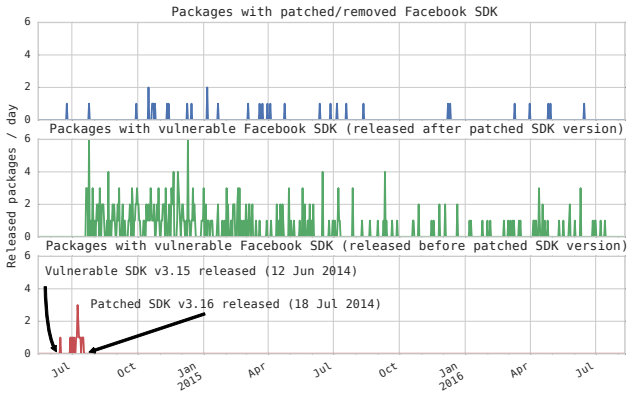

Figure 9: Daily releases of packages with a vulnerable or patched Facebook library between 04/2014 and 07/2016

(e.g., Android Studio and ADT) and is automatically added to apps based on their supported Android API levels, while Facebook prior to version 3.23.0 had to be manually downloaded from the developer pages and can since version 3.23.0 be retrieved via the Maven central repository. For Android support v4 we found that there is a clear bias towards newer library versions among the top apps (see Figure 7), while the majority of the top apps contain a (highly) outdated version of the Facebook SDK (see Figure 8). This indicates that app developers attend more carefully to the up-to-dateness of their IDE and its shipped packages than to manually retrieving external libraries (even from a central repository).

As a potential second factor we considered the app popularity measured in the app's number of downloads. However, we could not discover any effect (Kendall's $\tau = 0.01$ with $p = 0.8 \cdot 10^{-7}$) of an app's download rank (e.g., "1K", "50K", "10M") and the time required to adapt a new library version.

Public API compatibility.

Further, we were interested in whether the compatibility of the libraries' public APIs, through which app developers integrate libs, influences lib adaption. For quick adaption of a new library version, a stable public library interface for consecutive versions is helpful. Addition of new methods to the API is less problematic in this regard, while deletion of methods or changes in the signature (parameter or return types) might force app developers to adapt their code.

We analyzed all libraries in our database with at least ten consecutive versions for which we also have the release dates. This comprises in total 70 distinct libraries. We define the public API to be stable between consecutive versions, if there are at most additions of public methods but no deletions or modifications of existing signatures. The public API compatibility ranges from 7% (*Crittercism* lib) to 87% (*Amazon Analytics* lib). For this data set, we could not detect any statistically significant correlation between the adaption rate of new lib versions and a lib's API compatibility (Kendall's $\tau = -0.29$ with $p = 0.2401$) or the changes in the lib's public API (Pearson's $r = 0.01$ with $p = 0.7637$), respectively. This warrants further investigation into the motivation for app developers to adopt lib updates and potentially other factors, such as library documentation (e.g., both *Facebook* and *Android's support v4* only have slightly above 50% compatibility on average between updates and they release detailed changelogs and even upgrade guides to support developers, but have quite different rates of adaption of their updates among the top apps).

7.2 Detecting Vulnerable Library Versions

We use LibScout to investigate the presence of vulnerable third-party libraries within our app repository. In particular, we use two highly severe vulnerabilities from the recent past as case studies: an account hijacking vulnerability in the Facebook SDK v3.15 [36] and CVE-2014-8889 of the Dropbox SDK versions 1.5.4–1.6.1 [8] that allowed attackers to capture the user's Dropbox files via vulnerable apps. For each incident, we investigated the number of affected packages/apps, their user-base, as well as vulnerability window and time-to-fix for the affected apps.

Facebook account hijacking.

Facebook released their SDK v3.15 for Android, which contained an account hijacking vulnerability, on 06/11/2014. In the histories of our sample set apps, we discovered, in total, 394 affected packages from 51 distinct apps, when only considering packages with exact matches of the vulnerable lib version. For 18 of those apps, we knew the number of downloads from Play and are able to estimate a lower bound of 69M downloads of vulnerable packages. For 356 affected packages, our data set contained the release dates and enabled us to investigate the vulnerability windows and times-to-fix of those packages. Figure 9 illustrates the releases of vulnerable and fixed packages in our data set, where we also consider removal of the Facebook lib from the app as a fix. Most noticeably, the majority of the vulnerable packages (338, in the middle facet of the figure) were released *after* Facebook released the fixed SDK v3.16, in some cases even still in July 2016, i.e., more than 1.5 years after the patched SDK. Of the affected apps, 13 apps never released a fixed version and even their latest version on Play is still vulnerable. For 33 of the remaining apps, we can calculate the average vulnerability window and time-to-fix with absolute certainty (i.e., no gap in release dates and version history) and their average time-to-fix is 188 ± 55 days and average vulnerability window is 190 ± 55 days. Those are worrisomely high numbers that expose the end-users to unnecessary long vulnerability periods when considering

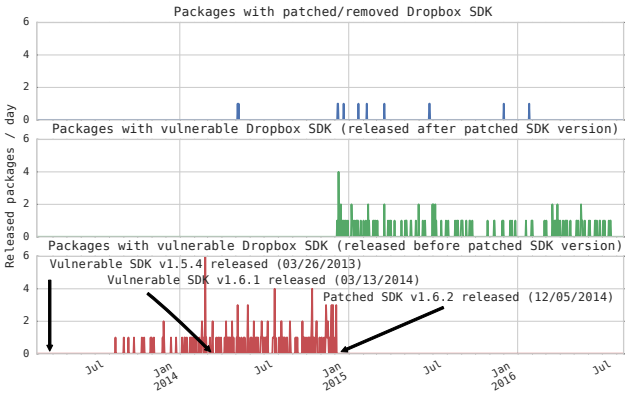

FIGURE 10: DAILY RELEASES OF PACKAGES WITH A VULNERABLE OR PATCHED DROPBOX LIBRARY BETWEEN 04/2014 AND 07/2016

that Facebook released a fixed version only 36 days after the vulnerable version (see Figure 9).

Dropbox data stealing.

For the vulnerability of Dropbox SDK versions 1.5.4 and 1.6.1, we found in our sample set 360 affected packages from 23 distinct apps, when only considering exact library matches. For 11 affected apps our data set contains information on their downloads, which accumulate to 11M. For 301 packages we have the release dates available, which allow us to investigate their time-to-fix and vulnerability windows. Figure 10 depicts the daily releases of patched/vulnerable apps with the Dropbox SDK and from our data set we derive an average time-to-fix of 59 ± 110 days and an average vulnerability window of 196 ± 127 days, where the latter one can be explained by the very delayed release of the patched SDK v1.6.2. Of the 23 affected apps in our data set, 9 did not release a fixed version and their vulnerable versions are still the most recent ones on Play.

7.3 Analysis of security-related APIs

Prior studies [9, 28] have shown that misuse of cryptography APIs is widespread among Android apps and that dynamic code loading by apps can be exploited to hijack apps. However, the original studies reported their results on a *per-app* basis and did not consider the extent to which third-party libraries contribute to this problem. Thus, we were interested in how prevalent such misuse is among third-party libraries and how helpful techniques like LIBSCOUT could be to augment such studies with better accountability. We focus our analysis of security-related APIs on advertising libraries since they are the most popular and widespread type of libraries. For both analyses, we used WALA to create a set of candidate libraries by scanning the bytecode of the 315 ad samples of 39 distinct ad libs (see Table 2) for the relevant API calls. Since the number of samples is suitable for manual review, we refrained from re-implementing the original analysis methods and manually verified whether security properties were violated or not in our candidate libraries.

For the crypto API usage, we performed the same six checks as in prior work (R1-R6 in Table 4). This includes checks for constant encryption keys (R3), salts/seeds (R4+R6), and initialization vectors (IV, R2), as well as checks for the discouraged usage of ECB mode and low number of

Property	#Libs/Ver	Verified
R1: ECB mode for encryption	5/25	5/25
R2: constant IV for CBC	7/32	4/20
R3: constant symmetric keys	13/60	3/7
R4: static salts for PBE	2/2	2/2
R5: <1000 iterations for PBE	2/2	2/2
R6: static seed for SecureRandom	3/7	2/5

TABLE 4: RESULTS FOR CRYPTO API ANALYSIS OF AD LIBS SHOWING CANDIDATE AND VERIFIED LIBS/VERSIONS IN OUR LIBRARY SET.

iterations in password-based encryption (PBE). Table 4 summarizes our findings. The middle column shows the number of candidate libraries and versions. The last column shows the verified violations after manually checking the bytecode of each candidate. Out of all available ad libraries, 10 libs violate at least one of those properties. Several libs violate multiple properties, e.g., *Adrally* (R1,R2,R4,R5,R6), *Leadbolt* (R2,R3,R6), *domob* (R1+R2) and *AppFlood* (R4+R5). In 12 of the 18 verified libs (66%), all versions of those libraries were affected and in 14 cases even the latest available version in our data set was still affected. All of the "fixed" libraries removed the affected code segment, but not one actually implemented a proper fix for previously vulnerable code. The only library (*Leadbolt* SDK) that modified affected code, replaced an empty initialization vector (R2) in versions 5.x with constant IV in version 6.0. We used LIBSCOUT to detect the affected application packages in our data set. In total 2,667 app versions of 296 distinct apps with a cumulative install-base of 3.7bn were affected by those ad libs with verified crypto misuse. In summary, improper usage of cryptography APIs is very common among the widespread ad libs and thus future work should investigate to which extent prior results [9] must be attributed to the library developers instead of app developers.

Second, we study dynamic code loading behavior of ad libs. We follow the approach described in [28] and test whether code loading is performed via package contexts, `Runtime.exec`, or the `DexClassLoader`. Only 9 out of 39 ad libs use any form of dynamic code loading. In fact, only the *HeyZap* lib does code loading via the package context, but it only exposes this functionality to app developers. Only six libraries (33 versions) make use of `Runtime.exec` to execute `logcat` or some shell operations for modifying access rights of files. Only one version of the *ChartBoost* SDK contains a suspicious call to check for the presence of the superuser binary. The `DexClassLoader` technique is only used by the last two *Admob* versions and one version of *Tencent*. Both libs load a supplemental jar/dex file. Hence, in summary, dynamic code loading is not widespread in ad libs in our data set and is rather attributed to other principals (e.g., app developer or other libraries).

8. DISCUSSION

In Section 7 we studied security vulnerabilities in third-party libraries and, using LIBSCOUT, we were the first to show to which extent these problems actually affect the current set of top apps on Google Play. Parts of our analyses re-applied prior approaches [9, 28] and our results show that third-party libraries are a contributing factor to those original results, which reported only *per-app* results or could only exclude a small set of libs from their results based on unobfuscated

library package names. Thus, we argue that a lightweight approach for third-party library detection, like LibScout, which is resilient against common obfuscation techniques, can greatly enhance static analysis approaches (e.g., [12, 45, 21, 2, 42, 13, 28, 25, 11]) by allowing them to attribute their results to the correct principals (app or library developers).

Moreover, our results show that app developers adapt new library versions only very slowly, even when the currently used version contains severe security or privacy vulnerabilities. This work does not attempt to uncover the exact reasons for this slow adaption, but its results surely warrant further investigation of those root causes and the development of new ways to motivate or support app developers in adapting library updates. Potential alleys for this future work could be the distribution channel of libraries (e.g., central library repositories like Maven, integration into IDEs, on-device library services), compatibility of public APIs of libs, or better communication of library updates to developers (e.g., we noticed that in many cases there is no CVE entry for library vulnerabilities and even lib providers do not report/warn about such incidents in an adequate manner).

Current limitations and future work.

The design choice to extract profiles from the original libraries comes with the inherent limitation of completeness. Therefore, complementary techniques such as *WuKong* [40] or *Libradar* [22] can be useful to detect potential libraries that are not in LibScout's database. Currently, our approach does not detect code-only changes in libraries, however, we can limit the number of candidate versions to a small set. To pinpoint the exact version whenever the profiles are ambigious (see Section 6.1), we are experimenting with secondary, code-based profiles that are generated from dex bytecode operation types (e.g. invoke or move) after compiling the library jars to dex bytecode. These secondary profiles are still resilient to identifier renaming, but could be influenced by code-based obfuscation such as API hiding.

Although being robust against a larger number of obfuscation techniques than related work, more extreme forms like flattening the package hierarchy would still defeat our static approach (and any related approach). In this case the structure of the hierarchy is modified and the boundaries between app and library code become blurred. Since such techniques introduce severe side-effects, they are rarely used in practice. Another open problem is the manual or automatic dead-code elimination of library code. The complete library code is no longer statically included but only a (small) subset thereof. Our partial matching algorithm covers this problem up to the point where only so little library code is left in the app that the similarity score drops below a certainty threshold. In general, this is a hard problem that can not be completely solved with static analyses techniques. Instead, only solutions in which dependencies have to be declared explicitly would remedy this problem.

A few libraries, e.g., the Baidu SDK that recently had a severe vulnerability [35], are provided as native library. Such libraries could be detected based on the hashes of their shared object files included in the app packages. Providing a database for native libraries would complement our approach.

Lastly, our repository contains only the top apps on Google Play. Thus, the results of our study might shift when we also consider the "long tail" of apps. We deliberately decided for this approach, since those top apps account for the bulk of all downloads and the largest user-base on Play. Another technical reason is that building complete app version histories is also a highly time-consuming task (in our experience, one Google account can query one app version per 2–5 seconds).

9. CONCLUSION

There is a trend to include more and more libs into apps, while at the same time app developers slowly adapt to new library versions (if at all). This puts millions of users at risk if security vulnerabilities remain unfixed in current top apps. Similar to the Android fragmentation problem, our results show strong indications for a library version fragmentation problem. Even in top apps severly outdated library versions were found, implying that library providers can not act on the assumption that end-users may use the latest features or have the latest bugfixes. We conclude that there is a need for additional research on third-party libraries.

Ethical considerations. We inform the developers of vulnerable libs and report the affected apps on Google Play.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) (FKZ: 16KIS0345, 16KIS0656) and the project SmartPriv (FKZ: 16KIS0377K).

10. REFERENCES

[1] T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net, 2006.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI'14*, 2014.

[3] M. Backes, S. Bugiel, E. Derr, S. Gerling, and C. Hammer. R-Droid: Leveraging Android App Analysis with Static Slice Optimization. In *ASIACCS '16*. ACM, 2016.

[4] T. Book, A. Pridgen, and D. S. Wallach. Longitudinal analysis of android ad library permissions. In *MoST'13*. IEEE, 2013.

[5] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *ICSE'14*. ACM, 2014.

[6] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *ESORICS'12*. Springer, 2012.

[7] J. Crussell, C. Gibler, and H. Chen. Andarwin: Scalable detection of semantically similar android applications. In *ESORICS'13*. Springer, 2013.

[8] Dropbox Blog. Security bug resolved in the dropbox sdks for android. https://blogs.dropbox.com/developers/2015/03/security-bug-resolved-in-the-dropbox-sdks-for-android. Last visited: 04/27/16.

[9] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *CCS'13*. ACM, 2013.

[10] W. Enck, D. Octeau, P. McDaniel, and C. Swarat. A study of android application security. In *USENIX Security'11*. USENIX, 2011.

[11] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: an analysis of android ssl (in)security. In *CCS'12*. ACM, 2012.

[12] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *TRUST '12*. Springer, 2012.

[13] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow analysis of Android applications in DroidSafe. In *NDSS'15*, 2015.

[14] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WISEC'12*. ACM, 2012.

[15] GuardSquare. Dexguard android obfuscator. https://www.guardsquare.com/dexguard.

[16] GuardSquare. Proguard java obfuscator. http://proguard.sourceforge.net.

[17] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *DIMVA'12*. Springer, 2013.

[18] Licel Corporation. Dexprotector android obfuscator. https://dexprotector.com.

[19] Licel Corporation. Stringer java obfuscator. https://jfxstore.com/stringer.

[20] B. Liu, B. Liu, H. Jin, and R. Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. In *MobiSys'15*. ACM, 2015.

[21] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *CCS'12*. ACM, 2012.

[22] Z. Ma, H. Wang, Y. Guo, and X. Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. In *ICSE'16*. ACM, 2016.

[23] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO'87*. Springer, 1988.

[24] A. Narayanan, L. Chen, and C. K. Chan. Addetect: Automated detection of android ad libraries using semantic analysis. In *ISSNIP'14*. IEEE, 2014.

[25] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl. To pin or not to pin—helping app developers bullet proof their tls connections. In *USENIX Security'15*. USENIX, 2015.

[26] Parse Blog. Discovering a major security hole in facebook's android sdk. http://blog.parse.com/learn/engineering/discovering-a-major-security-hole-in-facebooks-android-sdk. Last visited: 04/27/16.

[27] P. Pearce, A. Porter Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *ASIACCS'12*. ACM, 2012.

[28] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *NDSS'14*, San Diego, CA, 2014.

[29] PreEmptive Solutions. Dasho java obfuscator. http://www.preemptive.com/products/dasho.

[30] J. Seo, D. Kim, D. Cho, T. Kim, and I. Shin. FlexDroid: Enforcing In-App Privilege Separation in Android. In *NDSS'16*, 2016.

[31] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Security'12*. USENIX, 2012.

[32] Smardec Inc. Allatori java obfuscator. http://www.allatori.com.

[33] S. Son, G. Daehyeok, K. Kaist, and V. Shmatikov. What mobile ads know about mobile users. In *NDSS'16*, 2015.

[34] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *MoST'12*. IEEE, 2012.

[35] The Hacker News. Backdoor in baidu android sdk puts 100 million devices at risk. http://thehackernews.com/2015/11/android-malware-backdoor.html. Last visited: 04/27/16.

[36] The Hacker News. Facebook sdk vulnerability puts millions of smartphone users' accounts at risk. http://thehackernews.com/2014/07/facebook-sdk-vulnerability-puts.html. Last visited: 04/27/16.

[37] The Hacker News. Warning: 18,000 android apps contains code that spy on your text messages. http://thehackernews.com/2015/10/android-apps-steal-sms.html. Last visited: 04/27/16.

[38] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. In *SIGMETRICS'14*. ACM, 2014.

[39] Vungle Support. Security vulnerability in android sdks prior to 3.3.0. https://support.vungle.com/hc/en-us/articles/205142650-Security-Vulnerability-in-Android-SDKs-prior-to-3-3-0. Last visited: 05/02/2016.

[40] H. Wang, Y. Guo, Z. Ma, and X. Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *ISSTA'15*. ACM, 2015.

[41] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating sdks: Uncovering assumptions underlying secure authentication and authorization. In *USENIX Security'13*. USENIX, 2013.

[42] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *CCS'14*. ACM, 2014.

[43] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *USENIX Security'15*. USENIX, 2015.

[44] W. Yang, J. Li, Y. Zhang, Y. Li, J. Shu, and D. Gu. Apklancet: Tumor payload diagnosis and purification for android applications. In *ASIACCS'14*. ACM, 2014.

[45] Z. Yang and M. Yang. Leakminer: Detect information leakage on Android with static taint analysis. In *WCSE'12*. IEEE, 2012.

[46] N. Zhong and F. Michahelles. Where should you focus: Long tail or superstar?: An analysis of app adoption on the android market. In *SA'12*. ACM, 2012.

[47] W. Zhou, Z. Wang, Y. Zhou, and X. Jiang. Divilar: Diversifying intermediate language for anti-repackaging on android platform. In *CODASPY'14*. ACM, 2014.

[48] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *CODASPY'12*. ACM, 2012.