RC25649 (WAT1612-020) December 13, 2016
Computer Science

IBM Research Report

Identifying Android Library Dependencies in the Presence of Code Obfuscation and Minimization

Salman A. Baset[1], Shih-Wei Li[2], Philippe Suter[1], Omer Tripp[3]
[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

[2]Columbia University

[3]Google

Research Division
Almaden – Austin – Beijing – Brazil – Cambridge – Dublin – Haifa – India – Kenya – Melbourne – T.J. Watson – Tokyo – Zurich

Identifying Android Library Dependencies in the Presence of Code Obfuscation and Minimization

Salman A. Baset
IBM Research
sabaset@us.ibm.com

Shih-Wei Li
Columbia University
shihwei@cs.columbia.edu

Philippe Suter
IBM Research
psuter@us.ibm.com

Omer Tripp
Google
trippo@google.com

***Abstract*—The fast growth of the Android app market motivates the need for tools and techniques to analyze and improve Android apps. A basic capability in this context is to identify the libraries present in a given Android app, including their exact version. The problem of identifying library dependencies is made difficult by two common build-time transformations, namely code minimization and obfuscation. Minimization typically incorporates used library fragments into an app, while obfuscation renames symbols globally across an app.**

In this paper, we tackle both of these challenges via a unified approach, which abstracts app and library classes into summaries of their interactions with system libraries. The summarization technique is resistant to obfuscation, and is amenable to efficient similarity detection (matching). We lift the class-wise matches into a set of library dependencies by encoding this problem as a global constraint/optimization system across all app classes and available libraries.

We have implemented our approach as the MOBSCANNER system. We report on the evaluation of MOBSCANNER against 20 Android apps along with a randomly chosen database of over 10K library versions belonging to 1K unique libraries. MOBSCANNER is able to pinpoint the *exact library versions* present across apps without and with obfuscation/minimization with recall scores of 98% and 85%, respectively.

I. INTRODUCTION

The Android platform has gained tremendous popularity since it was unveiled in 2007. Currently it holds a global market share of 78.4% with 1.5M activations of Android devices per day. In the US alone, there are 76M Android users. The main Android app market, Google Play, offers over 2M apps with a history of over 50B downloads to date.

These impressive statistics provide strong motivation to understand, and improve the quality of Android apps. In this paper, we focus on a key question in this space, which to our knowledge has not been previously addressed; namely:

> *What are the library dependencies of a given Android application?*

We emphasize that library dependencies include the *exact library version*, which is crucial information for many interesting use cases, as we motivate below.

Motivation. There is strong motivation for Android developers to reuse software components. First, Android apps are written in Java, which is known for its rich set of libraries. Second, the mobile app market is highly dynamic, mandating short development time and fast release cycles. Finally, there is great potential for reuse given the common needs of many apps, such as internet communication, UI widgets and login workflows.

There are also compelling reasons to uncover the library dependencies of a given application, including the following:

- Security: An application may use a library (version) with known security vulnerabilities, or even worse, a library that is confirmed to be offensive if not malicious (e.g., an advertising library that make use of sensitive user information without proper authorization) [7].
- Quality: Many libraries have known functional bugs, such as performance and memory problems [10]. Knowing the library dependencies of an application can help isolate bad behaviors exhibited by the application.
- Optimization: There is also potential for optimization, either manually (by notifying the developer) or automatically (via app rewriting or instrumentation), if it is discovered that an app depends on an older version of a library and a safe transformation is possible [17].
- Mining: The ability to automatically extract the library dependencies of an application opens the door for large-scale studies on the use of libraries by Android apps, allowing insight into questions such as what the most popular libraries are, how their popularity changes across app categories, and how often a recent version of a library is used [25], [6].
- Analysis: Finally, we note the relevance of extracting library dependencies to automated testing and verification tools. For verification, this provides a means to apply reusable summaries across library calls rather than reanalyzing such calls from scratch each and every time [13]. For testing, knowledge of the libraries used by the app is a valuable form of fingerprinting for guided selection of tests (e.g., inputs for security of functional testing that correspond to known bugs in those libraries) [16].

Indeed, all of these use cases benefit from, if not depend on, knowing the exact version of a library dependency. For most, if not all, of these use cases, complete resolution — even if imprecise — is preferable to missed dependencies. There are two main reasons. First, false positives can be addressed by downstream filters, whereas false negatives cannot be corrected.

Second, some of the clients require conservative reasoning. As an example, in security analysis the scenario of detecting a false vlunerability (to be eliminated through manual audit) is better than missing a vulnerability.

Challenges. Interestingly, though it may appear straightforward to compute the dependencies of an Android app, there are two serious challenges that complicate this task, sometimes to the point of rendering nonambiguous identification impossible:
Minimization: Though mobile devices are growing increasingly more powerful, there is still strong incentive to reduce the size of Android apps (and mobile apps in general), not only because of the remaining gap from laptops and PCs but also because many users still own legacy devices. Hence, Android apps typically do not include libraries in their entirety, but rather rely on reachability analysis to determine which parts of the library are used by the app (at the granularity of class members, i.e. fields and methods), such that only those class fragments are incorporated into the image, blurring the boundary between app and library code.
Obfuscation: To complicate the problem further, the resulting image often undergoes obfuscation. The motivation is that unlike web applications, whose server-side code is inaccessible to users, mobile apps are downloaded onto the end-user's device. This aspect lends the user the ability to disassemble and inspect the app, which might violate sensitive intellectual property and perhaps even uncover security weaknesses that render other users vulnerable. Notice, importantly, that obfuscation applies not only to symbols originating from the source code written by the developer but also to library symbols, since the boundary between app and library classes is blurred due to minimization.

By the end of minimization and obfuscation, the app's image, in the form of an `.apk` file, is largely a blob of obfuscated code with no immediate hints as to which libraries were incorporated into it. As an illustration, we refer the reader to Figure 1, which we later discuss in more detail. Notice in particular the static call site highlighted in red with method identifier La/a/a/a/a/c; .a. This is in fact a call to another method from the same Apache Commons Codec class (cf. the clear version in blue).

In this paper, we present a solution to the challenges highlighted above in the form of MOBSCANNER, a tool for identifying the precise versions of libraries used in the construction of an app. MOBSCANNER employs a combination of information-retrieval and constraint-solving techniques to report an effective set of candidate library dependencies (including their version), even when the app is obfuscated and/or when the set of potential libraries is very large.

Contributions. This paper makes the following principal contributions:

1) Identifying library versions: We formulate and address the problem of identifying, given an Android app, the library version contained in its image, even in the presence of code obfuscation and minimization. We are not aware of previous attempts to address this problem, which — as we motivate above — has important applications.
2) Technical solution: To address the problem stated above, we utilize techniques from the area of information retrieval (Section III), and combine them with constraint solving and optimization (Section IV). There are potentially other applications for these tools, and information retrieval algorithms in particular, which we hope this work will inspire.
3) Implementation and evaluation: We have implemented our technique as MOBSCANNER, a system for identification of the library/version pairs that an Android image contains. We report on a series of experiments to evaluate the accuracy of MOBSCANNER (Section V). Our initial results are promising, and highlight directions for future research.

II. OVERVIEW

In this section, we explain the challenge in identifying the library dependencies of Android apps, namely because code obfuscation and minimization are by now an integral part of the Android build process. We then outline the main steps of our technique.

ProGuard and the Android Build Process. The Android build process packages an app into a single application package (.apk) file. The .apk file contains all the information necessary to run the app on either an emulator or a physical device. It includes compiled .dex files (Java .class files converted to the Dalvik bytecode format), a binary version of the AndroidManifest.xml file, compiled resources (resources.arsc) and uncompiled resource files.

Recently the ProGuard tool, available as an open-source project,[1] has been assimilated into the build process. ProGuard shrinks, optimizes and obfuscates the code of Android applications. Code shrinking is achieved via reachability analysis, such that classes and class members (fields, methods, etc) that are found not to be reachable from the application entry points are removed. Obfuscation is accomplished via renaming of the surviving classes, methods and fields with semantically obscure names.

Enabling ProGuard for an application is seamless and painless. With Android Studio or the Gradle build system, for example, the `minifyEnabled` switch controls whether ProGuard is enabled in release builds. The ease of enabling ProGuard, and the benefit of obtaining a more optimized and protected app with a smaller image, are the reasons why many of the apps featured on Google Play are minimized and obfuscated.

The ProGuard reachability analysis is relatively aggressive in deciding which parts of the code are accessible from entry points, operating at a level as low as class members. As a simple illustration, consider the following synthetic example:

class C { void f() {...} void g() {...} }

If f() is seen by ProGuard to be transitively invoked by one or more of the entry-point methods, but the same is not true of g(), then C will be minimized into

class C { void f() {...} }

[1] http://proguard.sourceforge.net

An analogous transformation is applied by ProGuard in the case of unused fields.

To ensure that minimization does not eliminate reachable code from the binary image, thereby causing the app to crash at runtime, developers can add a manual specification of reachable code via the `-keep` annotation. For example, the directive

```
-keep public class * extends android.app.Service
```

specifies that all public classes extending the built-in Android `Service` class should be treated as reachable.

Identifying Library Dependencies. Both obfuscation and minimization complicate the task of identifying the exact library versions that an application is dependent on. Due to obfuscation, symbols are renamed. Due to minimization, classes and class members deemed unused are eliminated.

We tackle these challenges via a unified approach that consists of two main steps, which we discuss in turn. The input is an app A, where we assume the existence of a (comprehensive) database $\mathbb{D}$ of candidate library/version pairs. Repositories such as Bintray or Maven Central[2] enable the construction of such a database with relative ease, as we explain in Section V-C.

The first step, described in Section III, is to compute a "signature" for each of the classes in A, and compare the signature against those extracted from the classes of the libraries contained in $\mathbb{D}$. Intuitively, the signature is a feature vector, where the features correspond to elements of the code that are resistant to obfuscation, such as instantiation of a core (i.e. java.$\star$) or platform (i.e. com.android.$\star$) class or use of a string constant.

While identification and extraction of obfuscation-resistant features is relatively straightforward, a key question that remains is what weights to assign to the different features. As an intuitive example, we refer the reader to Figure 3. Like the MD5Crypt class in that example, many other classes are likely to allocate objects of type java.lang.StringBuilder, which the compiler instantiates to implement string concatenation. However, it is much less likely for another class to define a string constant with value $apr1$. Hence, this feature should be assigned a much higher weight for the purpose of signature matching.

To obtain an effective weighting scheme, we build on results in the area of information retrieval and text mining, specifically the tf-idf algorithm [20]. This algorithm computes the weight of a given feature in terms of (i) its intra-class frequency (i.e., its number of occurrences in the given class) vs (ii) its inter-class frequency (i.e., the number of other classes where it occurs). Intuitively, a feature has high discriminative power, and thus high weight, if it occurs frequently in the given class (e.g., the class makes multiple uses of the constant $apr1$) but infrequently in other classes (e.g., no other class makes use of this constant).

Given the ability to weight features, and thus perform class-level matching, the second main challenge — discussed in Section IV — is to lift the class-wise results to the level of library/version pairs. Simply declaring all libraries with classes that match strongly against app classes as dependencies, or doing so only for libraries with at least k strong matches, are overly coarse heuristics that are hard to justify. Indeed, the problem of selecting which library/version pairs to treat as dependencies calls for global reasoning (rather than e.g. greedily accounting for one class at a time).

To perform such reasoning efficiently and effectively, we leverage the power of modern constraint solvers in specifying a constraint/optimization system that captures the essential considerations. These include (i) the constraint that two versions of the same library cannot simultaneously act as dependencies of the same app, (ii) the goal of "covering", via the set of dependencies, a maximal number of app classes for which there exist strong matches (other classes are assumed to originate from the app), as well as (iii) the goal of minimizing the overall imprecision of matching for selected libraries.

These requirements and objectives, and in particular the goal of maximizing coverage of appropriate app classes, bias toward high recall, which is preferable to high precision as a design choice given clients like security analysis and optimization. Indeed, in Section V we report on our ability to achieve almost perfect recall for clear apps, and still high recall for obuscated/minimized apps (98% and 85%, respectively).

III. Class Matching

Code minimization obviates any attempt to match the code of an app against complete libraries. On the other hand, though minimization may eliminate individual members inside a class (fields or methods), in most cases the majority of class members is retained due to the coarseness of (static) reachability analysis. Reasoning at the resolution of single class members is overly granular, missing the signal due to co-occurrence of multiple members within the same class.

In light of these observations, we decided to define as the atomic unit for matching Dalvik classes. A Dalvik class, much like a Java class, carries metadata on its access flags, parent class and implemented interfaces, corresponding source file, annotations, etc. Beyond these, the class definition points to static and instance fields as well as direct and virtual methods.

Obfuscation-resistant Features. Some of the information defined via a class is subject to obfuscation transformations, e.g. the names of classes, methods and fields originating from either the source code or third-party libraries. However, there are several categories of symbols that are resistant to obfuscation.

We illustrate the effects of obfuscation in Figure 1, which presents the bytecode representation of the source code in Figure 3, from the Apache Commons Codec library, with and without obfuscation respectively. Quick comparison between the two versions reveals, for example, that md5Crypt, a static method from the library, is renamed to a.

Still, as this example highlights, there are two main categories of symbols that are resistant to obfuscation:

- Symbols from core libraries: By core library we simply mean any type under either java.$\star$ or com.android.$\star$. There are different contexts in which such symbols

[2]https://bintray.com and https://search.maven.org

```
if-eqz v3, 001d
const-string v0, "$apr1$"
invoke-virtual {v3, v0}, Ljava/lang/String;.startsWith:(Ljava/lang/String;)Z
move-result v0
if-nez v0, 001d
new-instance v0, Ljava/lang/StringBuilder;
invoke-direct {v0}, Ljava/lang/StringBuilder;.<init>:()V
const-string v1, "$apr1$"
invoke-virtual {v0, v1}, Ljava/lang/StringBuilder;.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
move-result-object v0
invoke-virtual {v0, v3}, Ljava/lang/StringBuilder;.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
move-result-object v0
invoke-virtual {v0}, Ljava/lang/StringBuilder;.toString:()Ljava/lang/String;
move-result-object v3
const-string v0, "$apr1$"
invoke-static v2,v3,v0,Lorg/apache/.../digest/Md5Crypt;.md5Crypt:([BLString;LString;)LString;
invoke-static v2,v3,v0, La/a/a/a/a/c;.a:([BLString;LString;)LString;
move-result-object v0return-object v0
```

Fig. 1. Dex bytecode to which the method in Figure 3 compiles without obfuscation, and delta due to obfuscation (in blue). (Some type signatures have been simplified for readability.)

manifest. We consider the following contexts for symbol s defined by a core library:

1) use of s in an extends clause (e.g., Exception in class C extends Exception);
2) invocation of method s (e.g., StringBuilder.append in sb.append(str)); and
3) instantiation of an object (e.g., StringBuilder in the new-instance instruction in Figure 1.

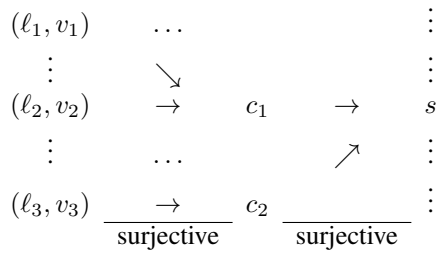
- String constants: It is nontrivial either for obfuscation, for minimization or for optimization passes to alter string constants while ensuring semantic equivalence to the previous version. The same is not true of numeric constants, which often undergo folding during optimization passes. Hence we focus on string constants alone. These provide useful signal when matching between app and library classes, especially if the string is unique (as with "$apr1$" in the example).

Each of the above (sub)categories yields a set of features for class-wise matching. That is, given a database of libraries, each string constant arising in any of the libraries is defined as a feature; similarly, each core-library class incident in an `extends` clause is mapped to a unique feature, etc.

As an illustration, from the example in Figure 3 — with the clear and obfuscated bytecodes in Figure 1 — we derive the features in Figure 2. The leftmost "Feature" column lists the symbol, and the "Category" column indicates the feature type. We return to the remaining columns shortly.

Henceforth, we refer to the feature vector extracted from a class as the *class signature*. Note, importantly, that class signatures are not unique. Two distinct classes, c and c', may have the exact same signature, or put differently, the class-to-signature mapping is a surjective yet not bijective function. The class signature abstracts away most of the class implementation, preserving only obfuscation-resistant symbols, and so agreement among different classes on the signature is possible, and is more likely in the case of small classes involving few (if any) obfuscation-resistant symbols.

This observation, combined with the observation that different library/version pairs may share classes in common (as often happens), are addressed by adding appropriate levels of indirection, as we illustrate schematically here:



That is, different library/version pairs (ℓ_i, v_i) may map to the same class c_j per the containment relationship. Furthermore, different classes may point to the same class signature if they agree on features and feature values.

Matching Method: the tf-idf Algorithm. Intuitively, matching at the class level proceeds in two steps. First, given a database $\mathbb{D}$ of libraries, we extract obfuscation-resistant features, as explained above, from the classes $c[\ell]$ of each of the libraries $\ell \in \mathbb{D}$. Then, given an app A, we check for pairwise similarity between the class signature F_c extracted from each of the classes $c \in A$ and the signature $F_{c[\ell]}$ (for each of the library classes $c[\ell]$) as an indication whether c is due to ℓ.

The key consideration is to decide how to weight the different features that we extract. Intuitively, if some class c defines a unique constant string s, but also invokes `StringBuilder.append`, then s provides greater signal than `append`, which is used by almost every Java class. Hence, we would like to assign a higher weight to s.

This judgment has a parallel in the field of natural language processing, and in particular in the area of text mining. There the problem is to decide, given term t and document d in corpus D, how important t is to D. By analogy, the class signatures are the documents, and the terms are the individual features.

A simple yet effective method to decide the weighting scheme for features is the *term frequency-inverse doument frequency (tf-idf)* algorithm [20]. Intuitively, the tf-idf value of a feature increases proportionally to its number of occurrences in the class signature, but is offset by the frequency of the feature in the class-signature database, which adjusts for the fact that certain features are more frequent in general (as illustrated with `StringBuilder.append`).

More formally, given feature f, class signature c and database $\mathbb{D}$ of class signatures s.t. $c \in \mathbb{D}$:

$$\mathsf{tf}(f, \mathbb{D}) = \begin{cases} 0 & \text{if } f \notin c \\ \log(1 + \mathsf{freq}(f, c)) & \text{otherwise} \end{cases}$$

$$\mathsf{idf}(f, \mathbb{D}) = \log\left(\frac{|\mathbb{D}|}{|\{c \in \mathbb{D} : t \in c\}|}\right)$$

$$\mathsf{tf\text{-}idf}(f, c, \mathbb{D}) = \mathsf{tf}(f, c) \times \mathsf{idf}(f, \mathbb{D})$$

In the above, the tf function computes the log frequency of feature f in class c, where log scale is used to adjust for highly frequent features (like `StringBuilder.append`), which would otherwise dominate any comparison. tf provides an *intra*-class view of the features.

Feature	Category	Frequency		tf Score	
		Clear	Obf.	Clear	Obf.
$apr1$	string constant	3	3	0.65	0.65
java/lang/String: startsWith	virtual invocation	1	1	0.32	0.32
java/lang/StringBuilder	new instance	1	1	0.32	0.32
java/lang/StringBuilder: append	virtual invocation	2	2	0.51	0.51
java/lang/StringBuilder: toString	virtual invocation	1	1	0.32	0.32

Fig. 2. Features derived from clear and obfuscated bytecodes in Figure 1, and their corresponding frequencies and (normalized) tf scores

```
static final String APR1_PREFIX = "$apr1$";
public static String apr1Crypt(final byte[] keyBytes, String salt) {
  // to make the md5Crypt regex happy
  if (salt != null && !salt.startsWith(APR1_PREFIX)) {
    salt = APR1_PREFIX + salt;
  }
  return Md5Crypt.md5Crypt(keyBytes, salt, APR1_PREFIX);
}
```

Fig. 3. Source code of apr1crypt method from class MD5Crypt in the Apache Commons Codec library

	(ℓ_1, v_1)	(ℓ_2, v_2)	(ℓ_3, v_3)
c_1	0.98	–	–
c_2	–	0.98	0.99
c_3	–	0.96	0.91

Fig. 4. Synthetic example with scores for library/version pairs for classes $c_1 - c_3$

idf complements tf with an *inter*-class view of the features by accounting for the frequency of the feature across all class signatures. The less frequently a feature occurs across the entire database, the more discriminative it is (as with the example of a unique constant string).

Finally, the tf-idf function reduces tf and idf, via multiplication, to a numeric weight. The more frequently a feature f occurs in the class signature, and/or the less frequently it occurs elsewhere (i.e., in other signatures in the database), the higher f's weight becomes. As a final step, we normalize the class signature into a unit vector.

An important observation that we exploit is that the idf value of a feature is not affected by its frequency within a class signature, yet it requires us to consider all other class signatures. When adding a given library into the database, and while the database is still being populated, we cannot yet compute the idf value of a feature.

Hence, as is standard in the area of text mining [20], we persist only the tf value of features into class signatures due to libraries, and subsequently normalize the vector. Later, when performing the matching against an app, we compute the (complete) tf-idf value for features in app-induced class signatures, thereby factoring in frequencies.

Revisiting Figure 2, we have this data for the symbols extracted from the code in Figure 3 as the "Frequency" and "tf Score" columns for both the clear and the obfuscated bytecodes. Note that there is no difference across clear and obfuscated in the frequency of occurrence of symbols, and so they also share the same tf score.

Computing Match Scores. Having explained how features are extracted and how weights are assigned to the different features, we are now ready to explain how match scores are computed. Given application A and class signatures $\{s_1^A, \ldots, s_n^A\}$ induced by the classes of A, and class signatures $\{s_1, \ldots, s_m\}$ due to the library database (where normally $m \gg n$), score computation reduces to matrix multiplication.

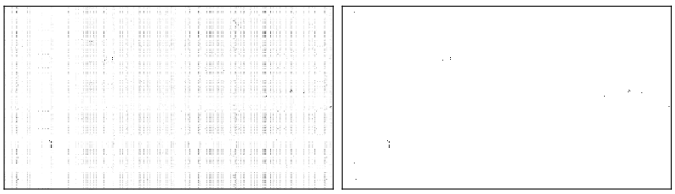


Fig. 5. Raw (left) and thresholded (right; $v \geq 0.95$) tf-idf matches for 450 application classes (horizontally) and 250 library classes (vertically), where white/black indicates a dot-product value of 0/1

Denote the arity of a class signature by k. Then we multiply an $n \times k$ matrix containing the signatures s_i^A as rows with a $k \times m$ matrix containing the signatures s_j as columns, i.e.: $S =$

$$\begin{bmatrix} s_{1,1}^A & \cdots & s_{1,k}^A \\ & \cdots & \\ s_{n,1}^A & \cdots & s_{n,k}^A \end{bmatrix} \times \begin{bmatrix} s_{1,1} & & s_{m,1} \\ \vdots & \vdots & \vdots \\ s_{1,k} & & s_{m,k} \end{bmatrix} = \begin{bmatrix} & \cdots & \\ \vdots & s_i^A \cdot s_j & \vdots \\ & \cdots & \end{bmatrix}$$

Recall that both the s_i^A vectors and the s_j vectors are normalized, and so in cell (i, j) in the score matrix S, we obtain a value $v \in [0, 1]$ that indicates the degree of similarity between class signatures s_i^A and s_j, i.e., $v = s_i^A \cdot s_j$. This dot product is commonly known as the *cosine distance*: a value of 1 denotes perfect agreement, which means that the class signatures contain the exact same obfuscation-resistant features, while a value of 0 denotes orthogonal vectors, i.e., class signatures that share no common features.

In practice, a large proportion of values in S are non-zero, as many features are relatively common across class signatures. To make the score matrix S actionable, we apply as a final step a point-wise thresholding function:

$$t(x) = \begin{cases} 0 & \text{if } x < T \\ x & \text{otherwise} \end{cases}$$

In our experiments, we set $T = 0.95$. Discarding all values below the threshold lets us concentrate on a smaller set of likely matches. As a visual illustration, Figure 5 shows (a fraction of) a matrix S before and after thresholding, where darker dots represent higher values. Thresholding is useful for manual interpretation of results, but is also a crucial step before the automated processing of the result matrix, which we describe in Section IV.

Discussion. In Section V, we provide empirical validation for the efficacy of tf-idf. Still, there are obvious limitations to tf-idf, and most notably the lack of modeling for order between features.

As an intuitive example, if a class contains a method m, such that in m there is first reference to string constant c_1 and then to c_2, then the ordered pair (c_1, c_2) provides stronger signal than set $\{c_1, c_2\}$. tf-idf, with the feature set described above, fails to exploit this additional signal.

Doing so leads naturally to the notion of n-grams, where our current features are unigrams, ordered pairs are modeled as bigrams, triplets as trigrams, etc. The number of features grows exponentially with n, but beyond this source of complexity, the remaining details of our technique remain unchanged.

In practice, we expect the performance hit due to switching from unigrams to say bigrams to be tolerable, since class signatures are already sparse and would become even much more sparse. There are efficient methods for representation and multiplication of sparse matrices [27]. Given our successful experience with unigrams, however, we defer this investigation to future work.

IV. Library Matching

In the previous section, we addressed the challenge of matching at the class level. Given this capability, the remaining challenge is to process the per-class matches into a set of library/version pairs that the app is hypothesized to depend on.

Constraint and Optimization System. We rely on the expressive power and efficiency of modern SMT solvers. We encode the problem of lifting class-level matches to library-level matches as a constraint/optimization system with a well-defined set of restrictions and goals.

The problem we are solving is related to the set-cover problem, and so we adopt similar terminology. Intuitively, we aim to determine which selection of library/version pairs results in the best *cover* of the classes in the application with high similarity scores. In the following, we also use "cover" to denote the set of library/version pairs that are selected as the optimal solution.

In Figure 6, we present the constraint/optimization system defined for an app A and database $\mathbb{D}$ of library/version pairs. The main semantic constraint is that two versions of the same library cannot simultaneously serve as dependencies of A. The optimization goals are that (i) as many of A's classes (surviving the similarity filter) as possible are covered by the libraries selected from $\mathbb{D}$ and (ii) the overall "cost" of the cover is minimal. We clarify the exact meaning of these statements in the following.

First, as Figure 6 specifies, we define three types of variables. For each class c, we define a variable v_c to denote whether that class is covered. Naturally classes that are due to the app itself, and not any of the third-party libraries, should not be covered. Hence, we create the set V of per-class variables only for classes surviving the thresholding filter:

$$V = \{\ \mathsf{v}_c \colon \exists (\ell, v) \in \mathbb{D}.\ \mathsf{match}(c, (\ell, v)) > T\ \}$$

where T denotes the thresholding value as described in Section III. We also define variables $\mathsf{v}_{(\ell,v)}$ and $\mathsf{c}_{(\ell,v)}$ to denote whether (ℓ, v) is in the cover and what the cost is for including it in the cover, respectively.

Moving to the constraints, the first constraint specifies that a class is covered iff at least one of the library/version pairs matching against it beyond threshold T is in the cover. Next, for the cost $\mathsf{c}_{(\ell,v)}$ of (ℓ, v), it is either 0 if (ℓ, v) is not in the cover or equal to $\mathsf{cost}(\ell, \mathsf{v})$, which we soon return to. The final constraint is that pairs (ℓ, v) and (ℓ, v') s.t. $v \neq v'$ cannot coexist in the cover.

Finally, there are two optimization goals. The first is to maximize the number of boolean variables v_c set to true. Naively, we would expect all classes matching against a library beyond threshold T to be covered, which would be a constraint rather than a maximization objective. However, there are corner cases (which we have encountered in practice) that counter this reasoning. In particular, if class c is covered *only* by (ℓ, v) and c' is covered *only* by (ℓ, v'), then obviously — due to the constraint that two library versions cannot coexist in the cover — one of these classes will not be covered. In reality, this happens when a spurious match occurs, typically if the application class is small and models few features. The second goal is to minimize the overall cost of the cover, which we explain by describing the cost function. The two goals are solved in order: From the entire set of solutions that maximize the cover, the one with the lowest cost is selected.

The* cost *Function. To illustrate and motivate the cost function, we make reference to the small synthetic example in Figure 4. In this example, there are different possibilities how to cover classes $c_1 - c_3$. All of these necessarily involve (ℓ_1, v_1), which is the only library/version pair covering c_1, and the remaining question is whether to also select (ℓ_2, v_2), (ℓ_3, v_3) or both.

Intuitively, though (ℓ_3, v_3) matches against c_2 marginally better than (ℓ_2, v_2) (0.99 vs 0.98), its match score against c_3 is much lower (0.91 vs 0.96), and so overall (ℓ_2, v_2) is a better choice for the cover with little gain out of choosing both (ℓ_2, v_2) and (ℓ_3, v_3). A simple heuristic to capture this reasoning is to define cost as the aggregate error due to selection of a library/version pair:

$$\mathsf{cost}(\ell, v) = \sum_c\ (\ell, v) \text{ covers } c\ ?\ 1 - \mathsf{match}(c, (\ell, v))\ :\ 0.0$$

For the example in Figure 4, we obtain the following:

$$\begin{array}{lcll} \mathsf{cost}(\ell_1, v_1) & = & 1 - 0.98 + 0.0 + 0.0 & = 0.02 \\ \mathsf{cost}(\ell_2, v_2) & = & 0.0 + 1 - 0.98 + 1 - 0.96 & = 0.06 \\ \mathsf{cost}(\ell_3, v_3) & = & 0.0 + 1 - 0.99 + 1 - 0.91 & = 0.1 \end{array}$$

In light of the (separate) goal of maximizing coverage of classes, (ℓ_1, v_1) is selected, and the optimal decision for c_2 and c_3 is to select (ℓ_2, v_2).

There is useful intuition behind our cost heuristic. While it biases toward selection of fewer library/version pairs (in an attempt to minimize a summation expression), balancing against the optimization goal of maximizing class coverage, the choice of which library/version pairs to pick considers the global (rather than local) consequences of selecting a given combination of library/version pairs — as the example illustrates — by computing error w.r.t. all classes.

Variables	
v_c : boolean	true iff class c covered by ≥ 1 library
$\mathsf{v}_{(\ell,v)}$: boolean	true iff version v of library ℓ is in the cover
$\mathsf{c}_{(\ell,v)}$: real	contribution of version v of library ℓ to overall cost
Constraints	
$\forall\ c.\ \mathsf{v}_c \Leftrightarrow \bigvee_{\{(\ell,v):\ (\ell,v)\ \text{covers}\ c\}} \mathsf{v}_{(\ell,v)}$	c is covered iff a library/version pair covering c is in the cover
$\bigwedge$	
$\forall\ \ell, v.\ \mathsf{c}_{(\ell,v)} = \mathsf{v}_{(\ell,v)}\ ?\ \mathsf{cost}(\ell,v) : 0.0$	version v of lib. ℓ contributes to overall cost iff selected
$\bigwedge$	
$\forall\ \ell, v \neq v'.\ \neg\mathsf{v}_{(\ell,v)} \vee \neg\mathsf{v}_{(\ell,v')}$	two ver.s of same lib. can't be in cover
Objectives	
$\max \sum_c\ (\mathsf{v}_c\ ?\ 1 : 0)$	maximize coverage of app classes
$\bigwedge$	
$\min \sum_{(\ell,v)} \mathsf{c}_{(\ell,v)}$	minimize overall cost

Fig. 6. Constraints and optimization goals

Overcoming Nondeterminism Due to Library Versions. Our definition of the constraint system suffers from nondeterminism if different library/version pairs yield the exact same cost. A simple example of this is if the app utilizes classes that are common (i.e., fully identical) across different versions of a given library, in which case there is inherent ambiguity in resolving the actual library/version dependency.

Observe that nondeterminism arises if the output is a set of library/version pairs. To combat nondeterminism, we have adopted a solution whereby library/version pairs that are identical w.r.t. to their cost across all app classes are grouped into an "equivalence class". The output is then in terms of groups of library/version pairs rather individual library/version pairs, where the meaning (naturally) is that exactly one of the libraries in the group is hypothesized to be a dependency. Applying this revision to the formal system in Figure 6 is straightforward and entails minor changes.

Complexity Analysis. In conclusion of this section, we analyze the complexity of the constraint system. Given set $C = \{\ c, c', \dots\ \}$ of classes such that $|C| = m$ and database $\mathbb{D}$ of library/version pairs such that $|\mathbb{D}| = n$, we make the following observations:

- There are m class coverage constraints (i.e., $\mathsf{v}_c \Leftrightarrow \dots$).
- There are n cost constraints (i.e., $\mathsf{c}_{(\ell,v)} = \dots$).
- Assuming finite bound k on the number of versions a library has, there are $O(k^2 \cdot n)$ conflict constraints (i.e., $\neg\mathsf{v}_(\ell, v) \vee \neg\mathsf{v}_(\ell, v')$).

In total, since k is a constant, the size of the constraint system, C, is linear in m and n: $|C| = O(m + n)$.

While the encoded problem is in theory NP-hard, we observed that instances in practice were amenable to efficient solving. (See Section V-D.) The hard constraints are propositional formulas, and the optimization objectives reduce to unweighted MaxSAT/MinSAT problems (as the weights of all soft constraints are set to 1), for which there exist solving strategies that are efficient in practice [15].

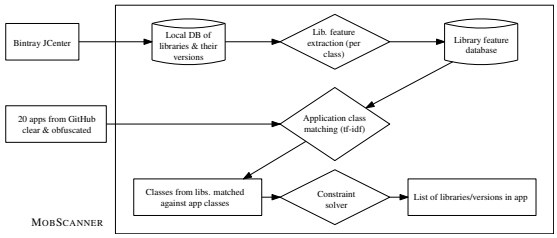


Fig. 7. MobScanner architecture

V. Implementation and Evaluation

In this section, we first describe our prototype implementation of our technique, and then present the experiments we conducted to evaluate our approach.

A. Prototype Implementation

We have instantiated our approach as the MobScanner system. MobScanner implements the class and library matching algorithms described earlier. Figure 7 visualizes the MobScanner architecture.

MobScanner makes use of several existing tools for these tasks, which we discuss in turn. The glue code, integrating between the different tools and converting between formats, is a set of Python scripts. In the future, as we scale up the system to handle a dramatically larget set of libraries, we intend to transition to Apache Spark.

Per-class obfuscation-invariant features are extracted from each library upon downloading it, and persisted into a Redis database. As a simple method to extract features, we use the textual representation of Dalvik bytecode (as exemplified in Figure 1), which we obtain by running the Android tool dexdump on apps. For libraries without a precompiled version in Dalvik format, we convert them first using the Android dx tool. From the textual representation, the features are matched using regular expressions.

There are two reasons for having libraries go through the intermediate step of dx as opposed to extracting features from the Java bytecode directly: First, this ensures that the extraction phase is identical regardless of the original format. Second,

extraction from the Dalvik format is orders of magnitude faster (though the conversion is costly). This may not affect users who only make use of the libraries once, but was an important factor in helping us iterate fast on our experiments.

When an app is submitted to MOBSCANNER, its features are extracted analogously to the way library features are extracted. We then utilize the NumPy Python library for matrix multiplication [24] to compute the tf-idf scores, and the Z3 solver [12] (as obtained from https://github.com/Z3Prover/z3 on Mar. 07, 2016) to compute a solution for the constraint/optimization system.

B. Experimental Design

Our experimental design is informed by the following main research question:

> *Can our technique identify the set of libraries present in the application (including their version) with high accuracy?*

To investigate this question, we have designed our experiments such that there are two main parameters:

- Library database: We experiment with two library databases. One version contains only the libraries the apps are dependent upon, whereas the other contains (many) other libraries, which exposes more opportunities for accuracy loss due to false matches.
- Clear vs obfuscated: We apply MOBSCANNERto both the clear and the obfuscated version of each of the apps. In doing so, we are able to assess the impact of obfuscation and minimization on our ability to identify library dependencies.

Overall, these parameters yield 4 different configurations, which we evaluate and analyze to respond to our main research question.

C. Benchmark Applications and Libraries

For our experiments, we aimed for a set of twenty apps, in clear and obfuscated form, as well as a set of several thousand library/version pairs. These goals reflect a tradeoff between the requirement for statistical significance on the one hand, and the (nontrivial) time and effort expended on each application to build a clear version, configure ProGuard, and build the obfuscated version on the other hand.

Clear Applications. We browsed through two publicly available listings of open-source Android apps.[3] Throughout this process, we downloaded more than 30 apps (the first batch, in order, under each of the listings), out of which we proceeded with the first 20 that we were able to successfully compile and build in both clear and obfuscated form.

Next, we determined the set of library versions in the clear and obfuscated versions of these apps. Here we note an important subtlety: While dependency information is specified by the developer in a dependencies file, that file does not always contain the precise library version, and may instead contain a regular expression defining a range of admissible versions. Furthermore, in certain cases, where different modules require different versions of the same library, the dependencies specified by the developer are overridden by the build system, which — as a default conflict resolution strategy — picks the most recent of the requested versions. To determine the ground truth with certainty, we used the Gradle build system.[4] Querying Gradle for dependencies returns unique and precise results as to which library versions are included in the build.

Obfuscated Applications. To ensure that a ProGuard configuration is available for each of the apps, we applied the following methodology:

1) If the app has an existing ProGuard configuration, and complies in release mode with that configuration, then we adopt the existing configuration without any modifications.
2) Otherwise, we create a fresh configuration according to the following steps:
   a) We list all the packages contained in the clear app (via a shell command utilizing the `dexdump` tool).
   b) We select, based on manual analysis, the package names that correspond to application (rather than library) classes. All classes under these packages are marked as reachable via the `-keep` syntax. (The `-keep` directive supports the wildcard expression `<pkgname>.*`, which has this exact effect.)
   c) We check whether the app compiles in release mode, and also if it can be launched and run in an emulator. If any of these checks fails (and in particular, if the reason why the app cannot run in an emulator is an unresolved symbol), then we continue to the next step. Otherwise we have obtained a valid ProGuard configuration.
   d) Finally, we specify additional packages as reachable via further manual analysis (typically one package at a time), and loop back to the previous step.

For all apps, we managed to produce a configuration that obfuscates all the dependencies. In all cases but two, all the libraries defined as dependencies were preserved in the obfuscated/minimized build. In cases where a library was entirely removed by the minimization pass, we made sure to reflect that in the ground truth for sound accuracy measurement.

Library Sets. The next challenge is to download libraries belonging to the apps, as determined by Gradle, as well as a large set of additional libraries and versions thereof.

In total, Gradle identified 124 unique library/version pairs across all of our benchmark apps. The versions belong to 87 distinct libraries. To download these libraries, we proceeded as follows: first, we downloaded all versions we could find on Maven Central [5] and Bintray JCenter [1]. Then, for library dependencies absent from these repositories, we searched for additional repositories specified by the developer(s) in their build configuration, and obtained all available versions from there. The additional repositories included Eclipse [3], Crashlytics [2], and JitPack [4]. Finally, we obtained all versions of the frequently used Android and Google support

[3] https://github.com/pcqpcq/open-source-android-apps and https://f-droid.org/

[4] https://gradle.org/

libraries from the locally installed Maven repositories that come with the Android SDK. The set comprising all available versions of the 87 libraries present in the applications contains 1,161 library/version pairs.

To form a larger database, we randomly selected 1,000 libraries from the entire list of libraries available on Bintray JCenter, for a total of 9,649 library/version pairs. We then added the 87 libraries present in the apps (all 1,161 versions of them). This library set forms the data set labeled "Local DB of libraries & their versions" in Figure 7.

From this set of of 1,087 libraries and 10,810 library/version pairs, we were able to successfully extract features out of 882 libraries (8,900 library/version pairs) and store them in the database labeled "Library feature database" in Figure 7. Note that many library/version pairs were compiled against versions of Java that are not supported on Android, and therefore could not be processed with the dx tool. Hence, these library versions cannot, by definition, be dependencies of Android apps.

Henceforth, we refer to the database containing the 87 libraries used by the apps (all versions thereof), and only those libraries, as the *restricted* database. The database containing all 882 libraries and their versions is dubbed the *complete* database.

D. Experiments and Results

As noted earlier, the MOBSCANNER workflow derives — given the per-class match scores — a set of candidate library dependencies (with no prior knowledge of the number of dependencies or any other hints). We evaluate the precision, recall and F-score values achieved by our constraint/optimization-based technique w.r.t. the restricted and complete database configurations.

The results are listed in Table I. The column "Deps." denotes the total number of (i) library dependencies of the clear app, and (ii) library dependencies contained in the obfuscated/minimized version of the app. The column "Classes" denotes the size of the app expressed as the number of classes, before and after minimization. The remaining "Pr.", "Rec." and "F-1" columns, grouped under "Clear" and "Obfuscated" (for both settings: 87 and 882 libraries), specify the precision, recall and F-score values achieved by our technique w.r.t. the clear and obfuscated versions of each app.

The reason why we maintain two separate columns, "Clr." and "Obf.", when counting the number of libraries present in the clear and obfuscated versions of the app, respectively, is minimization. As motivated earlier, in Section II, ProGuard can potentially eliminate a library from the app if liveness analysis deems that none of its classes are used in the app. We expect this to happen rarely in practice, since mostly libraries specified by the developer as dependencies are actually used, and indeed the deltas between the "Clr." and "Obf." columns are minor (where it holds invariably that the "Clr." columns is greater or equal to the "Obf." column).

We make the following observations:

Recall: Our technique achieves an impressive recall rate of 98% for the clear version of the apps and 85% for the obfuscated version across both the restricted and the complete databases. In fact, with the exception of 5 apps, recall is perfect for the clear apps. Under obfuscation, recall remains perfect for 6 of the 20 apps.

Precision: While precision is not as high as recall, we consider it tolerable, and in particular, well above the 50% mark. The fact that precision is lower than recall is to be expected; our constraint system is geared toward maximizing coverage of app classes by the available libraries. Thus, given a total of n classes in the app, m of which coming from libraries, there are $n - m$ classes that are potential false positives. There is no equivalent of this problem for recall.

Note also that, consistent with this observation, the constraint system can be revised to gear toward precision, e.g. by favoring quality matches to coverage of app classes. We haven't explored this design given our deliberate choice of biasing towards high recall.

We also point out a minor improvement in precision in the presence of obfuscation. A likely explanation is that this comes as a byproduct of minimization, which eliminates certain classes from the clear app that contributed to imprecision.

An interesting and pleasing observation, stemming from our manual analysis of false positives, is that at least in the case of the app Jiandan, MOBSCANNER was able to detect that the authors of the app included fragments from a library by copy/pasting some of its source code directly into the application. This not only means that MOBSCANNER is more precise than the numbers in Table I suggest (which are based solely on the dependencies reported by Gradle), but also that it is sufficiently robust to detect partial library inclusion via copy/pasting.

Resilience to redundant libraries: A final observation, summarizing findings w.r.t. precision and recall, is that the impact on accuracy of transitioning from the restricted to the complete database is tolerable. Though there is a decrease of approximately 5% in precision, and consequently also in the F-score value, recall remains unaffected and the F-score value is still acceptable (around 70%).

Our approach is also efficient. On average, matching an app against libraries took 6.0 and 11.1 seconds with the restricted and complete databases, respectively, on a commodity machine with 16 GB of RAM and Intel Core i7 processor. The time spent on constraint solving was 0.5 seconds and 1.7 seconds on average, respectively, with a maximum of 6.7 seconds for the K9 application in the complete database setting.

Discussion. The experimental results are consistent with our design goal biasing towards high recall. This is in view of potential clients, like security and optimization, for which overapproximation is preferable to underapproximation. Still, even with this bias, precision is tolerable.

VI. RELATED WORK

As stated earlier, we are not aware of existing techniques to identify the library versions present in an Android app. We report in on similarities with existing technique developed for other purposes.

Application	Deps.	Classes	Restricted Library Set (87 unique libs)						Complete Library Set (882 unique libs)					
			Clear			Obfuscated			Clear			Obfuscated		
	Clr./Obf.	Clr./Obf.	Pr.	Rec.	F-1	Pr.	Rec.	F-1	Pr.	Rec.	F-1	Pr.	Rec.	F-1
Android-Ctch.-Detector	9/9	1786/1782	82%	100%	90%	69%	100%	82%	82%	100%	90%	69%	100%	82%
Android-Dev.-Toolbelt	4/4	1423/775	67%	100%	80%	75%	75%	75%	57%	100%	73%	60%	75%	67%
AwkwardRatings	11/11	3436/1846	53%	91%	67%	56%	82%	67%	45%	91%	61%	53%	82%	64%
Beebo	11/10	4142/2821	58%	100%	73%	56%	90%	69%	48%	100%	65%	47%	90%	62%
CameraColorPicker	4/4	1675/977	57%	100%	73%	50%	75%	60%	57%	100%	73%	50%	75%	60%
ChaseWhisplyProject	5/5	3029/1052	63%	100%	77%	50%	60%	55%	63%	100%	77%	50%	60%	55%
Clip-Stack	3/3	1362/815	60%	100%	75%	67%	67%	67%	60%	100%	75%	67%	67%	67%
GivesMeHope	17/14	3114/1526	89%	94%	91%	80%	86%	83%	80%	94%	86%	75%	86%	80%
Jiandan	6/6	2184/1549	40%	100%	57%	43%	100%	60%	41%	100%	57%	43%	100%	60%
K9	11/11	2836/1823	59%	91%	71%	57%	73%	64%	50%	91%	65%	47%	73%	57%
Mizuu	16/16	5494/4601	64%	88%	74%	60%	75%	67%	54%	94%	68%	55%	75%	63%
MultiROMMgr	2/2	1302/749	67%	100%	80%	100%	100%	100%	67%	100%	80%	100%	100%	100%
Netguard	3/3	1742/1104	50%	100%	67%	60%	100%	75%	43%	100%	60%	50%	100%	67%
numixproject	1/1	62/41	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
PocketHub	16/16	4426/2991	63%	94%	75%	65%	81%	72%	54%	94%	68%	52%	81%	63%
Remindly	6/6	1466/880	75%	100%	86%	71%	83%	77%	75%	100%	86%	71%	83%	77%
S-Tools	2/2	1250/681	50%	100%	67%	67%	100%	80%	33%	100%	50%	40%	100%	57%
SoundRecord	5/5	1361/694	71%	100%	83%	80%	80%	80%	71%	100%	83%	80%	80%	80%
StockTicker	15/15	4970/4155	68%	100%	81%	59%	87%	70%	68%	100%	81%	54%	87%	67%
superCleanMaster	9/8	2908/2048	53%	100%	69%	44%	88%	58%	50%	100%	67%	39%	88%	54%
Average			64%	98%	77%	65%	85%	73%	60%	98%	73%	60%	85%	69%

TABLE I
PRECISION AND RECALL STATISTICS FOR RESTRICTED DATABASE OF 87 UNIQUE LIBRARIES/1,161 LIBRARY VERSIONS AND 882 UNIQUE LIBRARIES/8,900 LIBRARY VERSIONS.

Clone detection [9], [22] addresses the problem of identifying fragments of code that have been duplicated (and possibly modified) in source code. Some of the statistical techniques employed are often similar, such as the idea of abstracting code fragments into vectors that can be compared for similarity. Our work differs from existing solutions for clone detection in several important ways. First, we focus on larger units of code (classes in libraries as opposed to sometimes even fragments of methods); second, the app/library relation is different from the relation between clones, which often belong in the same code base; and third, we must account for (obfuscation) transformations that deliberately make the code harder to recognize. Finally, our main challenge is to pinpoint the specific version of the library that was used rather than identifying code reuse.

The technique of Software Bertillonage [11], applied also in the context of a study on code reuse in Android apps [23], is closely related to our signature construction and matching. However, because the features ("Objects of Interest" in the Bertillonage terminology) include many symbols that are not resistant to obfuscation, it is not directly applicable to our use case. The techniques proposed for detection of reused components (similarity and inclusion) are also unlikely to be sufficient to identify fragments of libraries after minimization, which we pose as a global constraint/optimization problem.

An existing tool that can partially retrieve obfuscated symbols is JSNice [21]. From a corpus of unobfuscated code, JSNice builds a statistical model of the interaction of symbols, and can query that model to retrieve names most likely to match obfuscated symbols. While JSNice can counteract some of the effects of obfuscation, it cannot point to specific sources, as the statistical model doesn't differentiate between the different contributors.

A main challenge that we address, and is particular to mobile apps, is obfuscation and dead-code elimination. This complicates detection of third-party dependencies, which is not a concern for Dependency-Check as it starts from the source project rather than the packaged app.

Viennot et al. present a scalable crawler for Google Play called Playdrone [25], [6]. They use it to index and analyze over 1M Android apps. They decompile the indexed apps and characterize trends across a large volume of Google Play apps. In particular, they identify libraries commonly used in Android apps; for this they compare class names in the apps with a whitelist of known libraries. While this technique is sufficient to establish trends given the very large numbers of apps considered, it (i) fails to extract any library dependencies from obfuscated apps and (ii) cannot distinguish between different versions of a library.

Many researchers have adopted static analysis techniques, such as data-flow analysis (taint analysis in particular), to identify integrity and privacy violations in Android apps [28], [19], [18], [14], [8], [26]. As noted earlier, we expect the above work can benefit from our approach to better pinpoint the library versions present in the app, creating opportunities to apply reusable data-flow summaries.

VII. CONCLUSION

We have presented a technique to identify Android library dependencies in the presence of code obfuscation and minimization at the granularity of exact library versions. Our technique is based on (i) extraction of obfuscation-resistant features from app classes and libraries, where features are assigned weights according to the tf-idf algorithm, followed by (ii) conversion to a hypothesized set of library dependencies by solving a (global) constraint/optimization problem. We have implemented our technique as the MOBSCANNER system, and report on experiments involving 20 apps from the wild, on which MOBSCANNER achieves a recall rate of 98% when the apps are clear, and 85% when they are obfuscated.

REFERENCES

[1] Bintray. https://bintray.com.

[2] Crashlytics. http://download.crashlytics.com/maven.

[3] Eclipse. https://repo.eclipse.org/content/groups/releases.

[4] Jitpack. https://jitpack.io.

[5] Maven central. https://repo1.maven.org/maven2.

[6] PlayDrone APKs on archive.org. https://archive.org/details/playdrone-apks. Accessed Nov. 2015.

[7] sourcedna. https://sourcedna.com/.

[8] 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 *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269. ACM, 2014.

[9] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377. IEEE Computer Society, 1998.

[10] Zhezhe Chen, Qi Gao, Wenbin Zhang, and Feng Qin. Flowchecker: Detecting bugs in mpi libraries via message flow checking. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[11] J. Davies, D.M. German, M.W. Godfrey, and A. Hindle. Software Bertillonage: finding provenance of an entity. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, 2011.

[12] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, 2008.

[13] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 567–577, New York, NY, USA, 2011. ACM.

[14] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: semantics-based detection of Android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '2014, pages 576–587. ACM, 2014.

[15] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Theory and Applications of Satisfiability Testing - SAT 2006*, Lecture Notes in Computer Science, pages 252–265. 2006.

[16] Xun Gong, Negar Kiyavash, and Nikita Borisov. Fingerprinting websites using remote traffic analysis. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 684–686, New York, NY, USA, 2010. ACM.

[17] Alden King and Scott B. Baden. Reducing library overheads through source-to-source translation. In *Proceedings of the International Conference on Computational Science, ICCS 2012, Omaha, Nebraska, USA, 4-6 June, 2012*, pages 1930–1939, 2012.

[18] J. Lerch, B. Hermann, E. Bodden, and M. Mezini. FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '2014, pages 98–108. ACM, 2014.

[19] S. Liang, M. W. Keep, M. Might, S. Lyde, T. Giltray, P. Aldous, and D. Van Horn. Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, SPSM '13, pages 21–32. ACM, 2013.

[20] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[21] Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from "big code". In *POPL*, pages 111–124. ACM, 2015.

[22] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *SCP*, 74(7):470–495, 2009.

[23] I.J.M. Ruiz, M. Nagappan, B. Adams, and A.E. Hassan. Understanding reuse in the Android market. In *2012 IEEE 20th International Conference on Program Comprehension*, pages 113–122. IEEE, 2012.

[24] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science and Engineering*, 13(2):22–30, 2011.

[25] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of Google Play. In *SIGMETRICS*, pages 221–233. ACM, 2014.

[26] 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 *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, PLDI '14, pages 1329–1341. ACM, 2014.

[27] Raphael Yuster and Uri Zwick. Fast sparse matrix multiplication. *ACM Transactions on Algorithms*, 1(1):2–13, 2005.

[28] Y. Zhongyang, Z. Xin, B. Mao, and X. Li. Droidalarm: an all-sided static analysis tool for android privilege-escalation malware. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, ASIA CCS '13, pages 353–358. ACM, 2013.