Ordol: Obfuscation-Resilient Detection of Libraries in Android Applications

Dennis Titze, Michael Lux and Julian Schütte
Fraunhofer Institute for Applied and Integrated Security (AISEC)
Germany
Email: {firstname.lastname}@aisec.fraunhofer.de

***Abstract*—Android apps often include libraries supporting certain features, or allowing rapid app development. Due to Android's system design, libraries are not easily distinguishable from the app's core code. But detecting libraries in apps is needed especially in app analysis, e.g., to determine if functionality is executed in the app, or in the code of the library.**

Previous approaches detected libraries in ways which are susceptible to code obfuscation. For some approaches, even simple obfuscation will cause unrecognised libraries.

Our approach – Ordol – builds upon approaches from plagiarism detection to detect a specific library version inside an app in an obfuscation-resilient manner. We show that Ordol can cope well with obfuscated code and can be easily applied to real life apps.

1. Introduction

New Android apps emerge every day to make users' life easier. To support and simplify application development, numerous libraries have been released to be included in Android apps. Those libraries provide features like backwards compatibility adapters, parsing, helpers for UI elements and many more.

For the analysis of apps, it can be desirable to recognize or even remove those libraries from apps. On Android, libraries inside an app have the peculiarity that the code of the library is merged into the codebase of the app. The resulting app contains only one file with both the app's code and the code of all libraries.

If a library should be detected, this can be easily done if it is included without modification. In that case, the libraries' classes reside in packages with known names and can therefore be easily identified. However, for acceleration, size reduction and protection of intellectual property, developers can use obfuscation tools which can rename classes, remove unreachable code parts, and optimize the code by performing numerous semantic-preserving transformations.

Once the app's code is obfuscated, contained libraries can not be as easily detectable any more. Renaming of packages is trivial for obfuscators, and is widely used. But even the code of libraries might not remain the same in the app after obfuscation. Depending on which functionality is used by a particular app, obfuscation tools can discard different unused portions of the included libraries. Similarly, the obfuscator can inline different methods of the library into application classes. Further, obfuscators can merge classes created by the app programmer with library classes, and put them into common packages.

Security researchers analysing apps face the problem that they can't tell the developer's code and the code of included libraries apart. Thus, they can neither filter out known library code before analysis, nor can they tell whether a found problem was caused by code of the app itself or by an included library.

Detecting libraries by their package name has another downside: if an app contains classes residing in the package "com.actionbarsherlock", it is reasonable to assume that the library *ActionBarSherlock* is included, but the version is unknown. This is especially problematic if a certain library version contains a vulnerability. Therefore, checking if an app contains a certain version of the library is not possible by only analysing the package names.

Our approach – called Ordol – uses techniques similar to those used for plagiarism detection. Plagiarism detection algorithms typically rely on the semantic properties of applications to detect code plagiarism. Typically, properties are extracted from the code of the app which are robust against semantic-preserving transformations, i.e., obfuscation. Ordol was built with a strong focus on the inheritance of the robustness of these algorithms.

By achieving a high robustness, Ordol gives researchers a method to detect classes belonging to libraries automatically, even if the app has been obfuscated. Furthermore, Ordol does not only detect which library is contained, but is also able to determine the version of the library. The evaluations in Section 5 shows that the detection of library versions are, with few exceptions, accurate.

This paper makes the following contributions:

- Ordol provides a novel approach to detect libraries inside obfuscated apps.
- Its detection algorithm is able to handle commonly used obfuscation techniques, including class and package renaming.
- Ordol is able to detect the version of a library instead of only the name of a library.

2324-9013/17 $31.00 © 2017 IEEE
DOI 10.1109/Trustcom/BigDataSE/ICESS.2017.292

The remainder of this paper is structured as follows: Section 2 shows work related to the topic of library detection in apps. Section 3 details necessary background information to construct an obfuscation resilient detection algorithm. Section 4 explains Ordol's system in detail and all its internal phases. An evaluation against a publicly available library detection tool is shown in Section 5. Finally limitations are discussed in Section 6, and Section 7 concludes this paper.

2. Related Work

LibRadar [1] uses frequencies of Android API calls inside the app to detect libraries. The authors claim that these are "stable code features" which cannot be modified during obfuscation. Whilst this is true for simple obfuscators which only perform renaming, this does not hold for more advanced obfuscators which also perform code inlining and unused code removal. Both obfuscations can break LibRadar's detection approach, whilst our approach is still able to detect those libraries with a high degree of certainty. LibRadar only returns the name of the used library, but does not indicate which version is detected, whereas our approach is also able to determine the version of the included library. Currently, LibRadar is the only tool which is publicly available, which is why LibRadar is used as comparison the evaluation of Ordol.

Backes et. al. present an approach which is resilient against common obfuscation, and is also able to determine the version of the library [2]. Similar to our approach, the authors generated a library database from the original libraries found on the internet and check apps against this library. Their approach and ours have several similarities, e.g., abstracting method calls into a more generic description. But differing to our approach, they still rely on the package structure. If this structure is simply renamed, their approach still works, but once all packages are flattened (i.e., all classes are moved to the default package), their approach fails. As our approach does not rely on any package structure, flattening the package structure is no limitation for Ordol.

Similar to LibRadar, WuKong [3] performs library detection by relying on the total number of API calls per package. Once code inlining or dead code removal is applied, this approach will not be able to detect libraries accurately.

More approaches have been published which rely on identifiers or the package structure. As this information can easily be changed, those techniques cannot reliably detect libraries. Examples for such approaches can be found in [4], [5], [6], [7], [8]

Juxtapp uses a similar approach for feature creation as our approach (i.e., k-grams), but is not able to cope with larger k or small basic blocks. [9]

3. Background

For the functionality of Ordol, it is essential to detect code semantically similar to the code of a set of known libraries. For this purpose, it is necessary to extract features from library code that can be recognized in analysed apps. Those features, known as birthmarks (BMs) [10], should be as unique as possible to avoid confusion with different code, i.e., false positives.

Two primary types of BMs exist: Dynamic BMs and static BMs. Dynamic BMs use characteristics of an application's execution. They are extracted at runtime and have proven to be useful for comparison of executable apps [11]. However, they cannot be directly applied on libraries, as libraries do not have a runtime behaviour by themselves.

Static BMs, on the other hand, are extracted from code in a static manner and form the basis for static detection of similar code in applications. Therefore, whenever the term "birthmark" occurs in this work, it refers to static BMs

A resilient BM will "survive" sophisticated obfuscating and optimizing transformations. However, the quality of a BM is not solely defined by its resilience. Appropriate BMs should also differ for independently created and different classes.

For similarity measures on method level and class level, Ordol does heavily rely on maximum weight bipartite matchings (MWBMs), also known as maximum weight matchings (MWMs). Duan et al. define them in [12] as follows:

> Given a weighted bipartite graph, the MWM problem is to find a set of vertex-disjoint edges with maximum weight. [12]

In Ordol, the vertices representing objects from a library (either methods or classes) are matched to vertices representing the objects of the analysed app. The edge weights $w(e)$ represent the similarity of the objects.

4. System Design

As first step a suitable BM for detection of libraries in obfuscated applications was designed. The following requirements should be satisfied for this BM:

High Resilience: The selected BM should be as robust as possible with respect to all anticipated semantics-preserving obfuscations, especially those that alter the control flow of methods. In particular, the birthmark must not deteriorate when large parts of classes or methods have been removed due to bytecode optimization.

Exhaustive information exploitation: The BM must exploit as much information as possible that is available from any library's bytecode. It is desired that not only information from large methods, but also content of methods with only a few instructions will contribute information to the BM.

Credibility: The BM should provide a high credibility. Particularly, it must have a low false positive rate regarding the matching of classes.

4.1. Analysis Steps

In a preparation step, a library version database has to be generated. For this, each library which should be detected

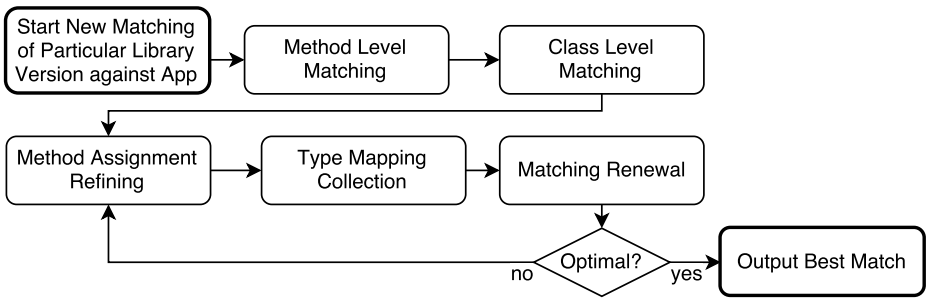

Figure 1: Ordol's library detection flow

has to be analysed by Ordol, and details about the methods and classes of this version are then stored for later detection. This only has to be done once for each library version. The stored details will then be used in the actual analysis of an app.

Once the preparation is finished, an app can be analysed for included libraries. To do so, the app is first translated to an intermediate representation (see Section 4.2). After this step, details about the methods and classes are compared against each stored library version (see Section 4.6), followed by an iterative refinement step to improve the result (see Section 4.7). The result of the analysis is a list of detected library versions and the corresponding similarity score. Figure 1 shows the major steps of the library detection.

4.2. Instruction Representation

Ordol uses Soot [13] for its static analysis. In particular, its default bytecode representation – Jimple – is used. This representation has important beneficial properties over Android Bytecode (represented as Smali code), mainly its strongly typed variables. However, some remaining problematic properties of this representation have to be addressed:

- Local variables are consecutively numbered according to the order of their declaration. If any declaration is removed by an optimization, moved to a different position, or inserted due to code inlining, the numbering of all succeeding variables is changed. This results in a change of all instructions containing these variables.
- Types of non-primitive variables and fields are contained in many instructions, e.g., object instantiations, or a method calls. As these identifiers can be subject to obfuscation, these types cannot be compared directly.
- Method names are contained in several instructions which can be renamed during obfuscation.

To address these problems, Ordol generates an information-reduced instruction representation based on Jimple, by performing the following modifications:

- Every variable is replaced by its respective type in angle brackets
- Any type or method name that can be renamed by obfuscation is replaced by "#". The replaced identifiers are saved as additional information.

The following example shows Jimple code before (Listing 1) and after (Listing 2) the simplification.

```
i1 = i2 + 1
virtualinvoke $r10.<com.test.Headers$1: void foo(
    int, com.test.Obj)>(0, $r9)
```

Listing 1: Jimple before Simplification

```
<int> = <int> + 1
virtualinvoke <#>.<#: void #(int, #)>(0, <#>)
```

Listing 2: Jimple after Simplification

4.3. k-Grams

To overcome the sensitivity to changes in the order of basic blocks, k-grams can be used. k-grams are fixed-length sequences of elements, usually extracted using a sliding window of size k. Listing 4 shows the k-grams extracted from the pseudocode shown in Listing 3 for $k = 3$.

```
A B C if (...) { D E F } else { G } H I
```

Listing 3: Pseudocode Example

```
ABC, DEF
```

Listing 4: k-grams for $k = 3$

Myles et al. [14] suggested k-grams as an alternative or supplement to the birthmarks of Tamada et al. [10] They have shown that k-grams provide significantly better resilience in comparison to the birthmarks of Tamada et al.

When respecting the boundaries of basic blocks, as shown in Listings 3 and 4, the resulting BM is immune against the reordering of the basic blocks.

Nevertheless, the given example also reveals a crucial weakness of the classical k-gram approach: information

contained in basic blocks smaller than k cannot be used, as no k-grams will be formed across the borders of a basic block.

4.4. Ordol's Flow-Based k-Gram Birthmark

For Ordol, a new birthmark combining properties of the k-gram BM and the flow path (FP) BM is used. This BM, henceforth called "flow-based k-gram BM", overcomes the problem of unused information for larger k.

Instead of using the basic blocks separately, the information about successor sets of basic blocks from the corresponding control flow graph (CFG) is used similar to flow paths [15]. When the end of a basic block is reached for an incomplete k-gram, all successor blocks are iteratively traversed until either the k-gram is complete or a return instruction is reached.

```
ABC, BCD, BCG, CDE, CGH, DEF, EFH, FHI, GHI
```

Listing 5: Flow-Based k-grams for $k = 3$

Listing 5 shows the resulting k-grams of this BM for the pseudocode in Listing 3. In contrast to Listing 4, each instruction appears in at least one of the extracted k-grams. Due to their construction, these k-grams do implicitly carry information about the control flow.

4.5. Similarities for k-Gram-Based Birthmarks

The similarity of two sets of k-grams can be measured using different metrics. The most prevalent metric in related work, used for instance by Hanna et. al is the Jaccard similarity [9]. This similarity of the sets A and L is defined as

$$J(A, L) = \frac{|A \cap L|}{|A \cup L|}$$

Instead of the Jaccard similarity, Ordol divides the cardinality of the intersection by the cardinality of L, which is the number of k-grams extracted from the library method:

$$QS(A, B) = \frac{|A \cap L|}{|L|}$$

This metric improves robustness against insertion of code.

4.6. Matching on Method Level and Class Level

The similarity values of classes and the similarity of the analysed application and a particular library are computed using MWBMs.

Weight of Methods: The weight $|m|$ of a method m is defined as the number of instructions found in all basic blocks of m.

Weight of Classes: Let C be a class containing n methods. The weight $|C|$ of C is defined as $|C| := \sum_{i=0}^{n-1} |m_i|$.

4.6.1. Class Similarity Calculation.

Let l_i and a_j be methods of a library class L and app class A, respectively. Ordol computes the MWBM on the weighted graph $G = \langle V, E, w \rangle$ with vertices $V = \{l_i | l_i \in L\} \cap \{a_j | a_j \in A\}$, edges $E = \{e_{ij} | e_{ij} = \langle l_i, a_j \rangle\}$ and weights $w(e_{ij}) = QS(l_i, a_j)$.

Now, let $\{\langle l_{i_0}, a_{j_0} \rangle, \langle l_{i_1}, a_{j_1} \rangle, ... \langle l_{i_n}, a_{j_n} \rangle\}$ be the pairs of matched methods representing the MWBM.
The *class similarity* S_C of L and A is defined as follows:

$$S_C(L, A) = \frac{\sum_k QS(l_{i_k}, a_{j_k}) \cdot |l_{i_k}|}{|L|}$$

Using only the weight of library methods l_i and the library class L for this similarity metric improves the robustness: neither increased method sizes nor additional methods change the resulting similarity.

4.6.2. Library Similarity Calculation.

Let L_i and A_j be the classes of a library $\mathfrak{L}$ and the analysed app $\mathfrak{A}$, respectively. Ordol computes the MWBM on the weighted graph $G = \langle V, E, w \rangle$ with vertices $V = \{L_i | L_i \in \mathfrak{L}\} \cap \{A_j | A_j \in \mathfrak{A}\}$, edges $E = \{e_{ij} | e_{ij} = \langle L_i, A_j \rangle\}$ and weights $w(e_{ij}) = \overline{S_{Ct}}(L_i, A_j)$ for the first iteration.

Now, let $M = \{\langle L_{i_0}, A_{j_0} \rangle, \langle L_{i_1}, A_{j_1} \rangle, ... \langle L_{i_n}, A_{j_n} \rangle\}$ be the pairs of matched classes representing the MWBM.
The *library similarity* S_L of $\mathfrak{L}$ and $\mathfrak{A}$ is defined as follows:

$$S_L(\mathfrak{A}, \mathfrak{L}) = \frac{\sum_k S_C(L_{i_k}, A_{j_k}) \cdot |L_{i_k}|}{\sum_k |L_{i_k}|_M}$$

Additional, the *library coverage* is defined as follows:

$$C_L(\mathfrak{A}, \mathfrak{L}) = \frac{\sum_k |L_{i_k}|_M}{\sum_k |L_{i_k}|}$$

By representing the result using these two figures, it is possible to distinguish between the similarity of detected classes to the actual library classes, and the size of the fraction of the library that was detected.

Shrinking might influence the library coverage, which therefore has only limited expressiveness, whereas the library similarity is the more significant measure for the quality of the result.

4.7. Iterative Detection Refinement

Although MWBMs with similarity-based edge weights deliver good library-to-application-class mappings, there will often be a low number of incorrectly assigned app classes.

These incorrect assignments have two main reasons:

1) Some app classes can contain similar or identical bytecode compared to certain library classes. When the real similarity is high, or the library class in the app has been modified due to optimization or obfuscation, these app classes can be wrongly detected as a library class.

2) When shrinking has been applied to the app, it is likely that some classes that formerly belonged to the detected library have been removed. If the app contains another class that is sufficiently similar to the original one it will take its place in the matching.

In order to correct or remove these incorrect assignments, Ordol uses an iterative heuristic approach which produces a more consistent class matching based on a previous matching. This approach utilizes occurrences of class types in the method bytecode of matched classes to derive new, more accurate type mappings.

Every iteration starts with a MWBM of app classes and classes of a library. For the first iteration, the weights used for the matching are based on the similarity of the BM as defined in Section 4.6. For every succeeding iteration, the following steps describe the procedure to create a new MWBM iteratively:

4.7.1. Method Assignment Refining.

For each class pair in the set $M = \{\langle L_{i_0}, A_{j_0}\rangle, \langle L_{i_1}, A_{j_1}\rangle, ...\langle L_{i_n}, A_{j_n}\rangle\}$ obtained from the previous MWBM all method pairs of $\langle L_{i_k}, A_{j_k}\rangle$ obtained from a MWBM as described in Section 4.6.1 are iterated as follows:

All feasible flow-based k-grams of each method pair which contain at least one call to a class are mapped as illustrated in Figure 2, with two exceptions:

- If k-grams with equal instructions do not reference the same methods, they are excluded to prevent wrong mappings. An example for such an exclusion is given by the app method in Figure 2: Two k-grams with instructions `E, F, G, H` and `R` are given, but the instruction `E` references two different methods, `T2'.a()` and `T2'.b()` for theses k-grams on the app side, and is therefore excluded.
- Any instruction referencing a method of a class not found in any pair $\langle L_{i_k}, A_{j_k}\rangle \in M$ is ignored, as it cannot improve the method mapping of any of these matched class pairs.

Let C_m be the class containing method m, and l', a' methods which are referenced in a method mapping of the methods l_{i_k} and a_{j_k}, respectively.

Further, the mixed similarity $S_M(l, a)$ is defined as $S_M(l, a) = QS(l, a) \cdot S_C(C_l, C_a)$, which is the similarity $QS(l, a)$ of the methods l and a multiplied with the similarity $S_C(C_l, C_a)$ of the classes C_l and C_a containing l and a, respectively.

Further, two variables $m_{l',a'} := 0$ and $w_{l'} := 0$ are defined for all pairs $\langle l', a'\rangle$ and all library methods l', respectively.

For each processed valid mapping, $m_{l',a'}$ and $w_{l'}$ are updated as follows:

$m_{l',a'} = m_{l',a'} + |l_{i_k}| \cdot S_M(l_{i_k}, a_{j_k})$ and $w_{l'} = w_{l'} + |l_{i_k}|$

The weight of the library method, the similarity of the method pair and the similarity of the class pair are used to determine the quality of a mapping (m).

Mappings found in small methods or methods with low similarity on method level or class level therefore have less influence on the final probability of a particular mapping.

The edge weights of the method-level MWBMs are then updated for each class pair, using the mapping probability, as follows:

$$w(e_{ij}) = \frac{m_{l_i,a_j}}{w_{l_i}}$$

4.7.2. Type Mapping Collection.

The next step processes all valid type mappings. This process is essentially identical to the method assignment refinement shown in Section 4.7.1, but processes class mappings instead of method mappings. Analogous to Section 4.7.1, two variables m_{L_i,A_j} and w_{L_i} are used for the class pairs $\langle L_i, A_j\rangle$ and the library classes L_i, respectively.

4.7.3. Matching Renewal.

Finally, the new MWBM on the weighted graph $G = \langle V, E, w\rangle$ is created with $V = \{L_i | L_i \in \mathfrak{L}\} \cap \{A_j | A_j \in \mathfrak{A}\}$, $E = \{e_{ij} | e_{ij} = \langle L_i, A_j\rangle\}$ and

$$w(e_{ij}) = \begin{cases} max\left(10^{-6}, \frac{m_{L_i,A_j}}{w_{L_i}}\right) & \text{if } S_C(L_i, A_j) = 1.0 \\ \frac{m_{L_i,A_j}}{w_{L_i}} & \text{else} \end{cases}$$

The purpose of the first case is to preserve a minimal edge weight for pairs of identical classes. Without this minimal weight, the edge weights for such pairs will become zero if no mapping is encountered for them.

In order to still obtain correct results when groups of pairwise identical classes exist, a minimal edge weight of 10^{-6} was chosen such that any non-zero mapping value will take precedence over this minimal value.

4.8. Termination

Whenever one round ends with the matching renewal, the library similarity and library coverage explained in Section 4.6.2 are calculated and compared against a threshold. If the threshold is not reached, the process terminates for this library, and the library is regarded as not being contained in the app.

If the current mapping passes this test, the resulting class mapping is saved to a set. If the mapping is already contained in this set, the class mapping either reached a stable mapping or a stable cycle of mappings. The process subsequently terminates for that particular library with the latest mapping as detection result for that library.

As the number of class mappings is finite, the algorithm will always terminate. Experiments have shown that a stable state is typically found after less then 10 iterations.

5. Evaluation

In order to verify the resilience and credibility of Ordol, the detection results are compared against LibRadar [1]. Because of the fingerprint uniqueness in LibRadar, it is subsequently assumed that libraries detected by LibRadar

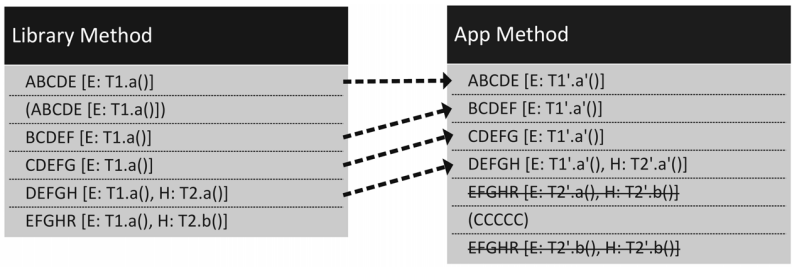

Figure 2: Processing of a method during method assignment refinement

are nearly perfectly credible and that LibRadar does not contain a relevant number of false positives.

In order to create a library signature set, 40 distinct libraries and 1112 unique versions have been downloaded from the Internet. Due to the manual download process, the library database only contains this limited number of distinct libraries. The libraries were selected from the top libraries of LibRadar [16] and Appbrain [17] which were available online.

In few cases, it was not possible to obtain or analyse a sufficiently large number of library versions. The effects of this lack on the evaluation results are separately explained for each evaluation.

For this evaluation, 1000 Android apps, have been randomly chosen from Google Play's most-popular app category in July 2016.

These apps were analysed using the web interface of LibRadar [1], and then analysed with Ordol. As not all libraries can be detected by LibRadar (i.e., if they are not in their database), and also not all libraries detected by LibRadar are in Ordol's database (due to the manual construction of the library database), only the 21 libraries (shown in Table 1 and Table 2) which can be detected by both tools were chosen for the following evaluation.

5.1. Resilience of Ordol Results

The resilience of Ordol's detection approach was evaluated by validating which percentage of LibRadar's findings is confirmed by the results of Ordol (depicted in Table 1).

It can be observed that Ordol provides a very high resilience $\geq 94\%$ for most of the selected libraries, with few exceptions:
Detection rate of Google Mobile Services (GMS) is low because GMS versions < 6.5.87 were not evaluated in Ordol as GMS was bundled differently in older versions.
Signatures were available only for the more recent versions 1.10.3, 1.11.0, 1.12.0 and 1.13.1 of *Parse.com*. It is assumed that this problem is the main cause for the poor detection rate observed for this library.
Similar to Parse.com, only few versions of the *InMobi* Ad SDK were found. The only available versions of that library were 4.0.4, 4.1.1, 4.4.0 and 6.0.0.

TABLE 1: LibRadar Detections confirmed by Ordol

Library	**Confirm. Rate**
Android Support v4	100,00%
Bolts Base Library	100,00%
Bump pay	100,00%
ChartBoost	100,00%
Dagger	100,00%
Facebook	100,00%
Flurry	100,00%
Jsoup	100,00%
OkHttp	100,00%
OkHttp okio Framework	100,00%
retrofit RESTful Library	100,00%
Nine Old Androids	99,21%
Apache Http	98,75%
Fabric	96,88%
ActionBarSherlock	96,72%
Google Gson	96,39%
Google Ads	95,61%
ZXing ('Zebra Crossing')	94,74%
Google Mobile Services	85,28%
Inmobi	76,47%
Parse.com	59,38%

5.2. Credibility of Ordol Results

The credibility of Ordol's detection approach was evaluated by validating the results of Ordol which are not confirmed by LibRadar, and separating them into false positives of Ordol and false negatives of LibRadar.

The separation of false positives and false negatives was achieved using a whitelist of package names for each mapping, which corresponds to common package names that were used for the detected libraries. When a library was detected by Ordol but not by LibRadar, a manual review was conducted to determine if it was an Ordol false positive. Similarly, if a common package name or very high > 90% match and coverage values indicated a correct detection, the detection was regarded as an LibRadar false negative.

The results of the evaluation are depicted in Table 2, with the error rate representing the number of presumed Ordol false positives, whereas the concordance rate represents the

detection rate of LibRadar among the verified, presumably correct detections of Ordol.

TABLE 2: Ordol Detections confirmed by LibRadar

Library	Concordance Rate (Error Rate)
ActionBarSherlock	98,33% (14,29%)
Nine Old Androids	76,69% (0,00%)
Apache Http	75,96% (4,59%)
Google Ads	67,59% (0,00%)
Dagger	65,22% (0,00%)
Android Support v4	64,68% (0,34%)
Google Gson	64,52% (3,88%)
ZXing ('Zebra Crossing')	64,29% (6,67%)
Bump pay	61,18% (0,00%)
Google Mobile Services	56,69% (0,00%)
ChartBoost	50,94% (0,00%)
Parse.com	48,72% (0,00%)
Fabric	47,94% (0,00%)
OkHttp	44,90% (12,50%)
Jsoup	42,86% (0,00%)
retrofit RESTful Library	39,06% (0,00%)
Inmobi	31,71% (0,00%)
Facebook	22,63% (0,00%)
Flurry	18,00% (0,00%)
Bolts Base Library	14,29% (6,67%)
OkHttp okio Framework	2,04% (1,34%)

The error rate shows that Ordol performs well for most libraries. The low concordance rate with low error rate for several libraries shows that our approach is able to detect significantly more libraries correctly.

Manual examination of the mappings with higher error rates revealed the following findings:

- Some older versions (3.x) of *ActionBarSherlock* contain an old version of the Android Support Library v4. This leads to some cases where fractions of the Android Support Library v4 are falsely recognized as the ActionBarSherlock library.
- A large fraction of the *okhttp* library consists of certain caching-related instruction sequences that occur in at least two other commonly used libraries, namely "AndroidAsync" by "koush" and "Disk LRU Cache" by "JakeWharton".
- The *ZXing* library contains specific instruction sequences for creation and handling of barcodes which are common in a number of similar libraries, causing an increased error rate.
- Functionality of the *Bolts Base Library* is also included in the *Parse.com* library. As only few versions of the *Parse.com* library were available for the database, libraries containing these missing versions can be detected wrongly as *Bolts Base Library*.
- Both *Google Gson* and *Apache Http* contain instruction sequences that are often confused with similar libraries for JSON and HTML processing.

5.3. Robustness against Obfuscation

To test the robustness of Ordol against obfuscation, the Open-Source mail app "K-9 Mail" was chosen. As the app can be compiled locally, and all included libraries are known, it is possible to test the detection capabilities of Ordol with and without obfuscation. For obfuscation, ProGuard was used with maximum settings (e.g., code merging, inlining, code removal).

Table 3 shows all libraries detected by Ordol before obfuscation, Table 4 after ProGuard's obfuscation.

TABLE 3: Ordol Detections in Unobfuscated App

Library	Version	Correct Version
apache-mime4j-core	0.7.2	0.7.2
apache-mime4j-dom	0.7.2	0.7.2
bumptech.glide	3.6.1	3.6.1
google.gson	2.3.1	false positive
squareup.okio	1.10.0	1.11.0
android.support-v4	23.1.1	23.1.1

TABLE 4: Ordol Detections in Obfuscated App

Library	Version	Correct Version
apache-mime4j-core	0.7.2	0.7.2
apache-mime4j-dom	0.7.2	0.7.2
bumptech.glide	3.6.1	3.6.1
google.gson	2.3.1	false positive
squareup.okio	1.7.0	1.11.0
android.support-v4	21.0.2	23.1.1

These results show, that Ordol is capable of detecting libraries even if the app itself is obfuscated. Several of the included libraries can be detected perfectly even after obfuscation. For some of the detected libraries (namely android.support and squarup.okio) the detected version changed after obfuscation. As ProGuard's obfuscation removes part of the code and even changes the structure of classes, confusion with other library versions cannot be eliminated, as the resulting obfuscated code can indeed be more similar to another version of the library.

K-9 Mail uses the library *squareup.moshi* which is not contained in Ordol's library database and therefore not detected. But as *squareup.moshi* uses the same streaming and binding mechanisms as *google.gson*, Ordol wrongly detects it as the library *google.gson*.

6. Discussion

Ordol was designed to cope with popular obfuscation techniques. As obfuscation evolves and gets more advanced, Ordol's concept might not be able to handle them. This would e.g., be the case if obfuscation translates Java code to native code, or encrypts all code.

Differing to more simple detection approaches, Ordol has to perform computation heavy analysis of the app.

Detection of libraries can therefore not be performed almost instantly, but e.g., required 70 seconds (on a typical laptop) for the detection of libraries in the K-9 Mail app in Section 5.3.

A downside of detecting the exact version of a library is the need for a library database containing all detectable library versions. If a version of a library is not included in this database, Ordol cannot detect the version correctly. If only minor versions are missing, Ordol will likely detect a different version of the library. But if major versions are not included, Ordol might not be able to detect the library, even if it is not obfuscated.

7. Conclusion

Detecting libraries is difficult in Android apps, as the library's code is merged into the codebase of the app. The detection is further aggravated once the app is obfuscated.

Ordol presents a novel technique and tool for detecting libraries in Android apps. As shown, Ordol is capable of detecting the exact version of a library even if the app is obfuscated, and therefore surpasses previously published detection techniques, which were not able to handle several obfuscation techniques.

An evaluation against a publicly available tool – LibRadar – showed that our new approach is able to detect significantly more libraries correctly.

References

[1] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 653–656. [Online]. Available: http://doi.acm.org/10.1145/2889160.2889178

[2] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 356–367. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978333

[3] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: A scalable and accurate two-phase approach to android app clone detection," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 71–82. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771795

[4] T. Book, A. Pridgen, and D. S. Wallach, "Longitudinal analysis of android ad library permissions," *CoRR*, vol. abs/1303.0857, 2013. [Online]. Available: http://arxiv.org/abs/1303.0857

[5] J. Crussell, C. Gibler, and H. Chen, *Attack of the Clones: Detecting Cloned Applications on Android Markets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 37–54. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33167-1_3

[6] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WISEC '12. New York, NY, USA: ACM, 2012, pp. 101–112. [Online]. Available: http://doi.acm.org/10.1145/2185448.2185464

[7] A. Narayanan, L. Chen, and C. K. Chan, "Addetect: Automated detection of android ad libraries using semantic analysis," in *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, April 21-24, 2014*, 2014, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/ISSNIP.2014.6827639

[8] W. Yang, J. Li, Y. Zhang, Y. Li, J. Shu, and D. Gu, "Apklancet: Tumor payload diagnosis and purification for android applications," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '14. New York, NY, USA: ACM, 2014, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/2590296.2590314

[9] 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 *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA'12. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 62–81. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37300-8_4

[10] H. Tamada, M. Nakamura, A. Monden, and K.-I. Matsumoto, "Detecting the Theft of Programs Using Birthmarks," Japan, 2012. [Online]. Available: http://isw3.naist.jp/IS/TechReport/report/2003014.ps

[11] D. Schuler, V. Dallmeier, and C. Lindig, "A Dynamic Birthmark for Java," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 274–283. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321672

[12] R. Duan and H.-H. Su, "A Scaling Algorithm for Maximum Weight Matching in Bipartite Graphs," in *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '12. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2012, pp. 1413–1424. [Online]. Available: http://dl.acm.org/citation.cfm?id=2095116.2095227

[13] Sable Research Group, "Soot: A framework for analyzing and transforming Java and Android Applications." [Online]. Available: https://sable.github.io/soot/

[14] G. Myles and C. Collberg, "K-gram Based Software Birthmarks," in *Proceedings of the 2005 ACM Symposium on Applied Computing*, ser. SAC '05. New York, NY, USA: ACM, 2005, pp. 314–318. [Online]. Available: http://doi.acm.org/10.1145/1066677.1066753

[15] H.-i. Lim, H. Park, S. Choi, and T. Han, "A Method for Detecting the Theft of Java Programs Through Analysis of the Control Flow Information," *Inf. Softw. Technol.*, vol. 51, no. 9, pp. 1338–1350, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2009.04.011

[16] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Top Libraries," 15.02.2017. [Online]. Available: http://radar.pkuos.org/top_libs

[17] AppTornado, "AppBrain Android library statistics," 15.02.2017. [Online]. Available: http://www.appbrain.com/stats/libraries/