CodeMatch: Obfuscation Won't Conceal Your Repackaged App

Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini
Technische Universität Darmstadt
Germany
{glanz,amann,eichberg,reif,hermann,mezini}@cs.tu-darmstadt.de,lerch@st.informatik.tu-darmstadt.de

ABSTRACT

An established way to steal the income of app developers, or to trick users into installing malware, is the creation of *repackaged apps*. These are clones of – typically – successful apps. To conceal their nature, they are often obfuscated by their creators. But, given that it is a common best practice to obfuscate apps, a trivial identification of repackaged apps is not possible. The problem is further intensified by the prevalent usage of libraries. In many apps, the size of the overall code base is basically determined by the used libraries. Therefore, two apps, where the obfuscated code bases are very similar, do not have to be repackages of each other.

To reliably detect repackaged apps, we propose a two step approach which first focuses on the identification and removal of the library code in obfuscated apps. This approach – *LibDetect* – relies on code representations which abstract over several parts of the underlying bytecode to be resilient against certain obfuscation techniques. Using this approach, we are able to identify on average 70% more used libraries per app than previous approaches. After the removal of an app's library code, we then fuzzy hash the most abstract representation of the remaining app code to ensure that we can identify repackaged apps even if very advanced obfuscation techniques are used. This makes it possible to identify repackaged apps. Using our approach, we found that ≈ 15% of all apps in Android app stores are repackages.

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; • **Software and its engineering** → *Software libraries and repositories*; • **Applied computing** → System forensics;

KEYWORDS

library detection, repackage detection, obfuscation, code analysis

ACM Reference format:
Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. 2017. *CodeMatch*: Obfuscation Won't Conceal Your Repackaged App. In *Proceedings of ESEC/FSE'17, Paderborn, Germany, September 04-08, 2017,* 11 pages.
https://doi.org/10.1145/3106237.3106305

ESEC/FSE'17, September 04-08, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106305

1 INTRODUCTION

Popular apps in the Google Play Store are installed on millions of devices. This attracts malicious actors to create altered, repackaged versions of those apps to steal the original owner's revenue, or to trick users and infect their mobile devices with malware. Detecting such repackaged apps is therefore necessary for a secure and viable app market.

Several techniques for repackage detection have already been proposed and can be broadly classified as being code-agnostic [20, 42, 43], graph-based [10, 15, 16, 25, 47], user-interface-based [17, 41], and code-signature-based [9, 22, 39, 45, 46]. The *Code-agnostic* approaches hash internal files of an app without considering the file content or type; as a result, the hashes could be evaded by single bit changes. *Graph-based* techniques derive the control-flow, data-flow or call graph of the analyzed app and measure the similarity by comparing isomorphic sub-graphs of the derived properties. Given that graph matching is a hard problem, these approaches potentially suffer from scalability issues [15]. Those approaches which try to abstract from the concrete graphs to achieve scalability, e.g., by using metrics, suffer from high false positive rates [10]. *User-interface-based* techniques also construct a graph, but use views as nodes and the transitions from one view to another as edges. These graphs can easily be fooled by changing or introducing pseudo-views. *Code-signature-based* approaches create signatures based on an apps' code to address the weaknesses of the graph-based approaches; the proposed approach also belongs to this category.

Challenges. A challenge for all existing repackage detection techniques are code transformations. Developers regularly minify and optimize their apps to increase performance. Additionally, they obfuscate their apps to protect their intellectual property. However, attackers also apply obfuscation to hide malicious code and to evade signature-based detectors, such as anti-virus software.

Current repackage detection techniques can only handle basic forms of obfuscation such as one-by-one identifier renaming, replacing types, and reordering of fields and methods [7, 31]. More sophisticated obfuscation techniques, such as moving classes between packages or changing Android API calls are not supported. Our evaluation of Google Play Store apps revealed that 60% [21] are at least partially obfuscated and that at least 20% use more advanced techniques. The effectiveness of repackage detection is further inhibited through the prevalent reuse of libraries in apps. Wang et al. [39] reported that more than 60% of the sub-packages in Android apps belong to library code. Hence, separating the library code from the app code is necessary. Otherwise, apps which use (nearly) the same libraries automatically share a large portion of the overall code base and are always identified as repackages – even if

the apps' code is completely different. A basic approach to filter out *non-obfuscated library code* is to use package white-lists [10, 46].

Another challenge for repackage detection tools are apps generated by App Makers, e.g., apps-builder[8]. In that case, the vast majority of the code base – the generator's libraries – will be the same and the rest will still be very similar. Current approaches, will generally flag such apps as repackages.

Proposed Approach. To address the identified challenges, we propose a method for repackage detection that consists of a library detection technique *LibDetect* and an app matcher *CodeMatch*, whereby the latter uses the former.

LibDetect uses five hierarchically organized representations. The first one is the original bytecode. The other representations each abstract over some additional aspects, such as the used identifiers or the control-flow. Hence, each higher level is less precise but potentially enables a higher recall. As a result of the precision/recall trade-off our library detection internally uses the five representations step-by-step. If a library method is only marginally obfuscated, our approach will identify the method using a less abstract representation when compared to methods that are more effectively obfuscated. After identifying the library methods we regroup them to match the potential original library classes. This enables us to identify methods and classes which were moved across class/package boundaries. To evaluate the robustness of our representations, we extracted roughly 200 sample library APKs from Maven Central [32], obfuscated them with a state-of-the-art obfuscator (*DexGuard* [33]), and tried to reidentify the original library methods. For the evaluation of *LibDetect*, we randomly selected 1,000 apps and identified the libraries manually to establish a gold standard.

Our app matcher *CodeMatch* uses fuzzy hashing [28] of an app's code to withstand various sophisticated obfuscation techniques and optimizations, including; class relocating, slicing, duplication of Android APIs in the app, code changes and code optimizations that affect the detection. Additionally, it orders the app's packages based on the size of their classes. This addresses the challenges faced by *DroidMOSS* [46] due to reorderings of classes and packages.

To test if an app is repackaged, *CodeMatch* performs the following steps: First, it filters apps which were generated using App Makers; these 2.4% of all apps can reliably be filtered using a white-list of the main-package prefixes; they are required by Android's signing process and cannot be obfuscated. Second, it filters the library code of apps using *LibDetect*. Third, it filters apps which have less than $\approx$ 300 lines of code; such apps can not be classified reliably by our approach. Fourth, it generates for each app the most abstract/obfuscation-resilient representation and fuzzy hashes it. Fifth, it compares the fuzzy hashes and if the similarity exceeds a predefined threshold, the apps are marked as repackaged.

We prepared the evaluation of *CodeMatch* by fuzzy hashing the descriptions of downloaded apps and randomly selecting 1,000 app pairs, whose fuzzy-hashed descriptions are at least 90% similar; we considered very similar descriptions as a first indicator for repackaging. Afterwards, we installed and executed each app pair to reconfirm their similarity manually. We used these results as the ground truth, to evaluate the findings of *CodeMatch*, *ViewDroid* [41], *DroidMOSS* [46], *FSquaDra* [42], and an repackage detection that uses the centroid concept from physics [10].

Additionally, we evaluated the effects of *CodeMatch* independent of *LibDetect*. To evaluate the effect that library detection has on repackage detection, we executed two of the repackage detection tools (DroidMOSS and Centroid-Based) additionally with *LibDetect* and *LibRadar* as pre-filters and compared the results with the other tools. To evaluate the effect of *CodeMatch* in isolation, we additionally run it with a library white-list and with *LibRadar*. We show that *CodeMatch* enables us to identify, in all library detection configurations, up to 50% more obfuscated and repackaged apps than the other approaches. In summary, we make the following contributions:

- Five abstract code representations that enable library and repackage detection with different precision/recall trade-offs
- *LibDetect*, a technique to detect library code on a class basis that outperforms the current most advanced library detection tool *LibRadar* by 70%.
- *CodeMatch*, a technique that detects app repackaging, which uses *LibDetect* to filter out libraries before measuring the similarity of the apps.
- The first quantitative comparative evaluation of available repackage detection approaches (*CodeMatch*, *ViewDroid*, *DroidMOSS*, *FSquaDra* and a *centroid-based* approach)

The remainder of this paper is structured as follows. Section 2 presents the attacker model. Section 3 gives an overview of obfuscation techniques. Section 4 describes the state of the art. Section 5 presents the proposed approach. Section 6 discusses the results of our evaluation. Section 7 examines threats to validity. Section 8 concludes the paper.

2 ATTACKER MODEL

We identified three kinds of attackers who create repackaged apps:

Attackers from the first category only apply basic changes/obfuscations of an app that do not require a deep understanding and configuration of obfuscators. Their primary goal is to avoid that the repackaged app is identified by hash-based approaches. The second category is able to make full usage of existing advanced obfuscators to effectively hide their apps even if state-of-the-art repackage detection approaches are used. The third category of attackers are experts who are able to apply custom obfuscations.

Current repackage detection tools are able to identify repackaged apps created by attackers from the first category. The proposed approach – *CodeMatch* – is additionally able to identify repackaged apps of attackers from category two.

3 CODE OBFUSCATION

We briefly introduce known obfuscation techniques, which were seen in the wild [12, 36, 38] or are performed by known obfuscators [6, 13, 14, 26, 33, 37, 44]. Optimization techniques are also included, because they introduce variance similar to obfuscation and cause similar issues. Throughout the paper we will refer to both techniques as obfuscation techniques

Name Mangling. In general, meaningful identifiers, such as field, method, class, and package names, are replaced by meaningless, small strings; e.g., "`Person`" → "`aa`". Package identifiers can even be reduced to the empty string; this *just* puts all classes in

the default package. Shortening names also improves the overall performance due to the smaller code size [6].

Modifier Changes. Field, method, and class modifiers can be changed most of the time without affecting the semantics of a program. The modifications range from basic changes, e.g., raising the visibility of classes or class members (e.g., "`package private`" → "`public`") or adding/removing the `final` modifier, to more complex ones. E.g., transforming an instance method into a static one requires an extra parameter to make the `this` reference explicit.

Structural Changes to a Method's Implementation. A very basic technique is to add `NOP`s - i.e., instructions which have no effect on the method's semantics, but modify the structure of the code. The primary effects are a larger method body and shifted jump targets of jump instructions, such as, `if`, `switch` or `goto`. More involved changes, such as changing the kind of an `if`-instruction ("`if>`" → "`if≤`"), generally affect the method's control-flow graph.

Code Slicing. Most applications do not use all features of the libraries they include. Therefore, it is possible to remove unused library code by creating a slice of essential functionality.

Code Restructuring. Common obfuscaters move classes and methods between packages, classes and methods. Such changes affect all call sites related to the changed class structure.

Method Parameters Manipulation. Reordering or removing/adding unused method parameters affects both: the signature and body of the method. This generally requires a corresponding update of all call sites.

Constant Computation. Constant values are replaced by expressions that compute the constant; e.g. the constant `100` is replaced by the computation `10*10`. A more advanced technique is the encryption of strings which are then decrypted on demand.

Fake Types. Already existing classes, in particular from libraries such as the Android SDK, are duplicated and used within the program instead of the original class; e.g., "`java.util.HashSet`" → "`com.MySet`". In more advanced cases a field's primitive type is changed; e.g., from `int` to `long`.

Code Optimization. Classic code optimizations also influence the code's structure whenever methods are inlined, values propagated, unused variables are removed, or control-flow is modified.

Hide Functionality. Sophisticated obfuscators hide functionality by encryption, recompilation, compression, and virtualization of selected classes (see Sharif et al. [36]). These techniques are currently the most effective protection mechanisms, but generally slow down the app's execution time, need advanced knowledge of the app's internal structure, or require manual code changes and are, therefore, rarely used in practice.

4 STATE OF THE ART

This section presents the state of the art in the area of library and repackage detection for Android applications.

4.1 Library Detection

Different repackage detection approaches [10, 46] use common library *white lists* to detect and filter out library code. White lists contain package names of known libraries and are compared with package names contained in Android apps. Currently the largest white list is collected by Li Li et al. [29]; it contains over 5,000 different names of library packages. The problem with using white lists is that changing just one character of a library's package name can completely evade the library detection.

LibD [30] uses the sub-/super-package relation (*inclusion*) and the inheritance relation between classes across packages (*inheritance*) to construct one reference graph per library. These graphs can then be compared with graphs extracted from an app. A graph is constructed by using (sub-)package names as nodes and inheritance or inclusion relations as directed edges. While this approach reduces the information needed for comparing libraries to the package level, it is vulnerable to changes that split or merge packages. If the package hierarchy is changed the graph has a different number of edges per node and cannot be compared with this approach.

LibRadar [31] is an approach for detecting library code in Android apps. Given an app's code, it extracts for each package a feature vector consisting of the observed Android API calls. These vectors are then hashed to get a fingerprint per package. These fingerprints can then be compared against fingerprints of known library packages. *LibRadar* is therefore resilient against the renaming of packages, but cannot handle obfuscations that merge or split packages that affect the vector of API calls.

The goal of *LibScout* [7] is to identify the version of an Android API that is used in obfuscated code. It generates merkle-tree-hash profiles in three steps. First, it replaces all types of a method signature that do not belong to the Android API with an "X" and then hashes the transformed signature. Second, the method signature hashes are sorted and hashed at the class level (class hashes). Third, class hashes are again sorted and hashed per package (package hashes). Finally, all hashes are used to identify library code. If the package hash does not match, the class and the method hashes are used. If no hash matches the code element is declared as non-library code. The hashes are generated for different library versions to make it possible to identify a specific version. Since the computation of the tree hash inherently reflects the implicit tree structure between packages, classes and methods, *LibScout* is not robust against cross-class/-package *code restructurings*.

To recap, the discussion above indicates the need for better library detection that is able to handle library instances with a changed package hierarchy. Additionally, beyond white lists, *LibRadar* is the only approach that can directly be used as a pre-step to repackage detection. Both white lists and *LibRadar* have their specific limitations mentioned above. Moreover, neither they nor the other approaches exploit information about the method bodies. These limitations lead to poor recall, as our empirical comparison of these techniques against our new approach will reveal in Sec. 6.

4.2 Repackage Detection

DroidMOSS [46] uses an app's bytecode instruction names (mnemonics) without arguments to compute a fingerprint by fuzzy hashing the entire mnemonic sequence in the given order. A white list is used for library filtering. *DroidMOSS* is vulnerable to *Code Restructurings* because the fingerprint depends on the code order.

ViewDroid [41] detects repackaged code by building view graphs. It extracts all *Activity* classes as view nodes and all actions as edges e.g., button pushes or intent execution. The tool performs a subgraph similarity measurement to compare view graphs, which is

time consuming. *ViewDroid* is vulnerable to insertion of libraries with own views, because it does not filter libraries. Furthermore, it can only compare apps with more than three views. In a random sample of 1,000 apps, 44.3% had less than 3 views.

Cuixia et al. [17] developed a tool that represents code in the same way as *ViewDroid*, but uses fixed size vectors of API call counts per UI widget instead of whole views. The tool achieves a more robust representation, but has the same drawbacks as *ViewDroid*.

FSquaDra [42] computes for all files referenced in the `MANIFEST.MF` of an APK the hash values and compares them with the extracted hashes of a potentially repackaged app. *FSquaDra* has no library detection and the insertion of library code would change the entire hash value of the code file. However, even the addition of some (useless) resources (e.g., sound files or images) would change the computed hashes and reduce the similarity with other apps.

Wukong [39] detects repackaged apps in three steps: First, it uses *LibRadar* [31] to filter out library code. Second, it filters equivalent apps by comparing two different fingerprints. Whereas the first is generated the same way as *LibRadar*'s fingerprints, the second is based on the frequency of API calls per package. Third, it generates feature matrices which contain occurrence frequencies of all variables in different contexts, to compare them for repackage detection. As *LibRadar* is used in the first two steps, the approach also suffers from the same drawbacks. Libraries that remain undetected during step one potentially cause a high false positive/negative rate in the detection results. When one of the fingerprints is falsely matched in step two, the repackage analysis is not executed.

DNADroid [15] filters out libraries by using package names and class hashes. After that, *DNADroid* extracts the data-dependency graph (DDG) of each app method and identifies repackaged apps by comparing the app's DDGs with the DDGs of other apps. The library filtering step of *DNADroid's* is vulnerable to renamings affecting package names or class hashes. Determining sub-graph isomorphism is generally computationally expensive [10], which makes *DNADroid* unsuitable for repackage detection in huge app stores, such as "Google Play Store".

AnDarwin [16] detects repackaged apps in four steps: First, comparable to *DNADroid*, it computes the DDG for each method. Second, it computes for each DDG the frequencies of each underlying instruction (e.g., assignments or additions). Third, to filter out library code it prunes vectors that occur more often among different apps than a predefined threshold. Finally, *AnDarwin* hashes each remaining vector and compares these hashes with hashes of potentially repackaged apps [5]. The approach fails to filter out libraries that are not used frequently enough according to the threshold.

The approach by Kai Chen et al. [10] constructs a control flow graph per method and represents it as a centroid. The algorithm filters out 73 popular libraries with a white list and matches the centroids of the remaining methods pairwise. The advantage of this approach is that similar centroids can be found efficiently, due to the sorting capability of the centroids. However, it also has two drawbacks. First, the detection depends on the sorting order, e.g., if we filter first by the instruction count, we possibly miss methods that are very similar by the invocation count. Second, when library code is not filtered properly (see 4.1), it is considered as app code, which renders compared apps artificially more similar.

To recap: Each repackage detection approach has its own specific drawbacks; all share problems due to limitations of the library detection in use. To address these problems, we designed *CodeMatch*, which we evaluate against *DroidMOSS*, *ViewDroid*, *FSquaDra*, and the centroid-base approach in Section 6. For *FSquaDra* and *ViewDroid*, the software was either available online or was made available to use upon request. The code for *DroidMOSS* and for the *centroid*-based approach was not available; but we were able to re-implement them based on the information available in their publications [10, 46]. The remaining approaches could not be acquired from their authors and we were not able to re-implement them based on their publications.

5 THE APPROACH

Our approach consists of two parts: First, an approach – *LibDetect* – for the identification and removal of library code from a given Android app (APK). Second, an approach that takes the app's code – after library removal – to find repackages. Both parts rely on abstract representations of the app's code to handle obfuscation. In Section 5.1, we first present the different code representations before we discuss *LibDetect* in Section 5.2, and *CodeMatch* in Section 5.3.

5.1 The Abstract Representations

To deal with obfuscation of methods and classes (see Section 3), we use five different abstract representations of methods. The representations build upon one another, each abstracting over some additional elements of the original bytecode compared to its predecessor. Table 1 shows which representation addresses which obfuscation techniques and Table 2 shows an example method `compare(int, int)` in the first four representations.

We use the **Bytecode (BC)** of a method as is, to reliably identify non-obfuscated library methods.

In the **Addressless Representation (AR)** we remove `NOP`s and program counters and abstract over jump targets. In the latter case, we replace forward jumps by *"along"* and backward jumps by *"back"*. Taken together, this addresses respective *Structural Changes to a Method's Implementation*. Furthermore, we remove all method modifiers to address *Modifier Changes*.

In the **Nameless Representation (NR)** we address *Name Mangling* and *Fake Types*. For that, we remove method names from the method signatures and invocation instructions and replace non-Android-API type references in return, parameter, field, array, and invocation instructions by lists of the types' Android-API supertypes. These lists represent those parts of the type information that cannot be obfuscated. We obtain them by walking up the type hierarchies, collecting all interface and class types defined in the Android SDK/Java. After that, we order them alphabetically.

Table 3 compares the method signature's AR and NR of `Object get(Key)` declared by the app class `MyHashMap`, which is a clone of Android's `HashMap`. We assume that the app class `Key` inherits only from `Object` and that `MyHashMap` inherits from `AbstractCollection`, `Map`, and `Object`. While the signature's AR contains the app-specific type information, its NR is identical to that of the `get()` method from Android's `HashMap`.

Table 1: Comparison of Obfuscation to Handling Entities.

Obfuscation	BC	AR	NR	SPR	Fuzzy SPR	LibDetect	CodeMatch
Name Mangling	–	–	×	×	×	×	×
Modifier Changes	–	×	×	×	×	×	×
Structural Changes to a Method's Implementation	–	×	×	×	×	×	×
Code Slicing	×	×	×	×	×	×	×
Code Restructuring	–	–	–	–	–	×	×
Method Parameters Manipulation	–	–	–	×	×	×	×
Constant Computation	–	–	–	×	×	×	×
Fake Types	–	–	×	×	×	×	×
Code Optimization	–	–	–	×	×	×	×
Hide Functionality	–	–	–	–	–	–	–

Table 2: A `compare(int, int)` Method in BC, AR, NR, and SPR Representation.

BC	AR & NR	SPR
`0:iload_0`	`iload_0`	`load`
`1:iload_1`	`iload_1`	`load`
`2:if_icmpne→9`	`if_icmpne→along`	`if→along`
`5:iconst_0`	`iconst_0`	`const`
`6:goto→19`	`goto→along`	`if→along`
`9:iload_0`	`iload_0`	`load`
`10:iload_1`	`iload_1`	`load`
`11:if_icmpge→18`	`if_icmpge→along`	`if→along`
`14:iconst_m1`	`iconst_m1`	`const`
`15:goto→19`	`goto→along`	`if→along`
`18:iconst_1`	`iconst_1`	`const`
`19:ireturn`	`ireturn`	`return`

Table 3: A Method Signature's AR and NR.

	AR	NR
Declaring Type	`MyHashSet`	`[HashSet, Object, Set]`
Method Name	`get`	
Parameter	`Key`	`[Object]`
Return Type	`int`	`int`

In the **Structure-Preserving Representation (SPR)** we address *Method-Parameters Manipulation* by sorting parameters alphabetically by the parameter type lists from NR. We also address *Fake Types* by removing all type information and the indexes from load and store instructions. For example, the instructions `astore` and `dstore_2` for storing an `object` or a `double`, are both represented by `store`. We unify size-dependent instructions, such as `ldc` and `ldc_w` and also drop all string constants, e.g., log messages, to address *Constant Computation*s that exchange these. We provide an exhaustive mapping from bytecode instructions to their SPR representation as supplementary material[21]. Furthermore, we improve our handling of *Structural Changes to a Method's Implementation* by representing all compare and jump instructions by `if`. However, the jump direction, i.e., "along" or "back" is kept.

In the **Fuzzy SPR** we address stronger *Code Optimization*s, *Constant Computation*s and *Code Slicing* by fuzzy hashing the token sequence from our SPR with *SSDEEP* [28]. This enables us to uncover similarity in the presence of such variation. SSDEEP chunks the input sequence depending on the total sequence length, abstracts each chunk to a single character, and concatenates all characters to a hash. It then repeats this process with doubled block size. The resulting signature consists of the two hashes and the block size.

5.2 *LibDetect*

LibDetect is a code-signature-based library-detection approach which can detect a library even if the library was sliced down to the required parts, library classes were moved to other packages, and/or instances of app classes were put into library packages. This addresses shortcomings of existing approaches [10, 29, 31, 46] which only identify complete library packages; searching for copies of the entire library code may miss library fragments and simply removing library packages may miss individual library classes and may accidentally remove app code. *LibDetect* searches for copies of library methods and later aggregates potential matches to classes, i.e., *LibDetect* identifies library code at the granularity of classes.

To match individual methods, we need to deal with obfuscation of both methods and classes, as both are referenced from within other methods. Since obfuscation introduces variation in the method's code, we use our abstractions (Section 5.1) to counter its effects, when we find library code with respect to a library database. In this process, the degree of abstraction becomes a tradeoff between precision and recall: If we match an app method to a library method using a more concrete representation, it is more likely that the match is correct and that we have fewer matches. However, we might miss better obfuscated library methods. Using a more abstract representation, we are more likely to find potential matches for an app method, but may also falsely match the method with a larger number of library methods. The overall process is depicted in Figure 1.

Preparation. While APKs contain Dalvik Bytecode, our tooling operates on Java Bytecode. Therefore, we use Enjarify [24] to transform the Dalvik executable file (DEX) from the APK into a Java archive (JAR). Enjarify is the most advanced DEX-to-Java-Bytecode transformer currently available.

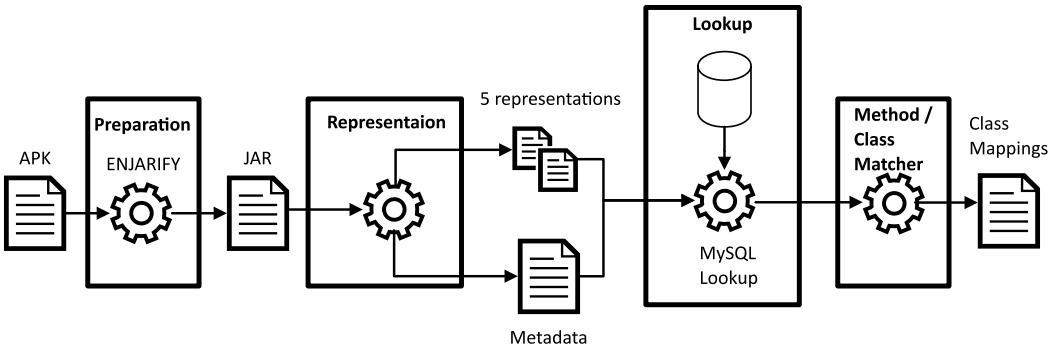

Figure 1: Toolchain to Identify Library Classes

Representation. We use the OPAL framework [19] to process the JAR file. For each method, we generate the five representations and extract the following information: the fully-qualified name of the method, its instruction count, its enclosing class and its defining package. This information is subsequently used to improve the precision when identifying library classes.

Lookup. To find library methods to which an APK's method might correspond, we need a database of known library methods which enables an efficient lookup. Therefore, we hash all five abstract representations of known library methods using the SHA-1 function [18] (the two fuzzy hashes of the SPR are processed individually) and build an index over the hashes of each representation. These indexes point to the methods' metadata.

Algorithm 1: Best-Matching Methods

```
Data: Method m
Result: Matching methods
matches ←lookup(m.FQN, m.BC);
if matches ! = ∅ then return matches;
matches ←lookup(m.FQN, m.AR);
if matches ! = ∅ then return matches;
for repr in m.{AR, NR, SPR, FuzzySPR1, FuzzySPR2} do
    matches ←lookup(repr);
    if matches ! = ∅ then return matches;
return ∅;
```

Given an APK's method, we use Algorithm 1 to lookup potentially-matching library methods in the reference database. The algorithm looks for matches using our abstract representations in increasing order of abstraction. The first two lookups search the database for methods with the same fully-qualified name (FQN) and the same BC or AR as the APK's method. This allows us to precisely identify library methods that are not or only slightly obfuscated. All subsequent lookups ignore the declaring classes' FQNs, to address *Name Mangling*. We perform another lookup with the AR this way and proceed with the other representations, until we find at least one match or otherwise declare the method as non-library code. This way, we find potential matches even in the presence of strong obfuscation, but identify only the matches with the highest match confidence. Additionally, we avoid unnecessary large sets of method matches on more abstract representations.

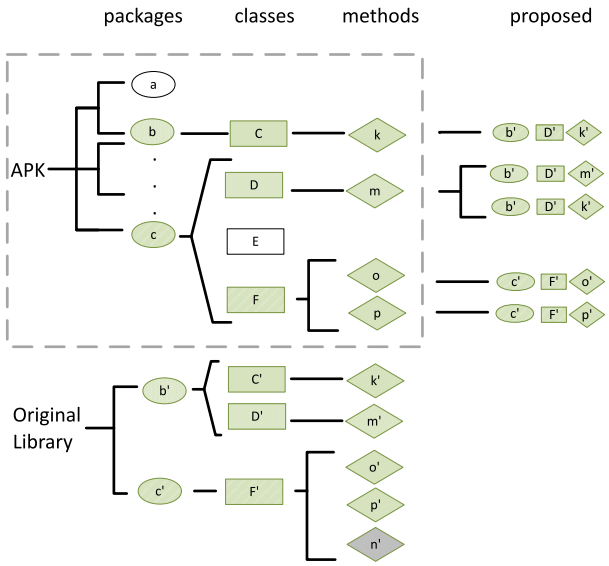

Figure 2: Aggregating Potentially-Matching Methods to Library Classes. The ellipsis are packages, rectangles are classes and diamonds are methods. Green indicates library and grey missing elements.

Method/Class Matcher. In the last step, we aggregate the APK methods for which we found potentially-matching library methods to library classes. Figure 2 shows the app's structure, i.e., the packages, classes, and methods it contains. The code of each included library corresponds to a fragment of this structure (b, c ellipse, C, D, F rectangles and k, m, o, p diamonds). Due to *Code Restructuring*, this fragment may have a different structure than the respective library originally had. Also, due to *Code Slicing* (n' diamond), elements of the original library might be missing.

The aggregation first searches for app packages that contain code from library packages. For each app package it collects all the app methods with at least one matching library method. For each

unique library package from any such matching library method, it counts the app methods that match a library method from that package. We refer to this count as the app-to-library-method count.

The aggregation then continues for each such library package individually, in descending order of the app-to-library-method count. For example, from app package `c` in Figure 2, two app methods `o,p` match methods from the library package `c'` and one app method match methods from the library package `b'`. Therefore, we first process library package `c` and then `b`.

For each library package, the aggregation computes a mapping from app classes to library classes. To this end, it considers only potentially-matching library methods from that library package. It maps each app class with the library class to which the highest number of methods match. If multiple classes match equally many methods, the aggregation picks the library class with the more-similar size in terms of bytecode instructions. Each library class is mapped only once. For example, the app class `F` in Figure 2 is mapped to the library class `F'`, because two of its methods potentially match methods from `F'`.

Finally, we filter out class mappings where the app class has less than half as many bytecode instructions as the mapped library class, to avoid false positives due to a few methods that occur very frequently, especially in our abstract representations. The app classes that remain in the mapping are reported as library classes.

5.3 *CodeMatch*

Figure 3 depicts the workflow of *CodeMatch*.

Preparation. This step is the same as for *LibDetect* (see Section 5.2), except that we additionally extract the developer's public key (as her unique identifier) for the APK.

Library Slicing. After preparation, *CodeMatch* uses a library-detection tool, which reports detected libraries at either package- or class-level granularity. It then removes the respective elements from the APK's codebase. If the library detection reports multiple library classes, *CodeMatch* can be configured to abstract the reported class names to whole packages and remove these instead.

Filtering. In addition to the library code removal, we filter two kinds of apps that cannot reliably classified as repackages using our approach. First, those that consist mainly of library code, plus at most $\approx$ 300 lines of glue code [21]. We refer to such apps as *library apps*. Second, apps generated using "App makers". App makers generate apps by processing user created ui designs. Apps generated by the same App maker generally share a similar code base without necessarily being repackages. We filter these apps using a white list of 40 common used prefixes of known App maker frameworks. This list is the result of a web search for Android App maker frameworks.

Representation. In contrast to library code, which may be sliced when only a part of the library's functionality is used, an app's code is likely completely included in a repackage, as slicing would break its functionality. Since we already removed library code from our target APK, we assume that the remaining code almost entirely corresponds to the potentially repackaged app's code. Therefore, we can use the identified app code for the comparison with other apps' code. To this end, *CodeMatch* represents each app class by its own Android-API type list (as defined for our NR), the type list of all its fields, and our most-abstract representation, SPR, for all its methods. We chose the SPR because we do not want to miss methods that were similar beyond *Code Optimization*.

To address *Code Reordering*, we sort the fields according to the type lists and the methods according to their instruction count. To address *Name Mangling* of package and class names, i.e., to have a name-independent order of classes, we also sort the entire classes by their size, i.e., the sum of sizes of each field, the number of methods, and the instruction count. To address remaining smaller differences that might have been introduced by obfuscation, we fuzzy hash [28] the entire representation.

Comparison. Before we compare potentially repackaged apps, we establish a threshold on the fuzzy-hash similarity, above which we report analyzed apps as repackaged. We determine this threshold by executing *CodeMatch* on 1,000 apps and searching for the best F1-score (harmonic mean between precision and recall). We get the best F1-score with a threshold of 30%.

To efficiently find potential repackages of an app, we build a database of known apps which we processed as described in the previous steps. Additionally, we indexed the respective developers' public keys, to avoid reporting apps from the same developer as repackages. We compare apps from different developers by the fuzzy-hashed representation of the apps' code using F2S2 [40], which efficiently finds similarity matches based on the edit distance between fuzzy hashes. If the similarity score between the target app and a database app exceeds our threshold, we report a potential repackage.

6 EVALUATION

The evaluation answers the following research questions:

- **RQ1** How robust is our code representation against state of the art obfuscation? (Section 6.1)
- **RQ2** How effective is *LibDetect* compared to other library detection approaches? (Section 6.2)
- **RQ3** How effective is *CodeMatch* compared to other repackage detection approaches? (Section 6.3)

In addition, we quantify the repackage share found by *CodeMatch* in the wild (Section 6.4).

6.1 Robustness of Code Representation

We assess the robustness of our code representations against obfuscation by applying *LibDetect* to obfuscated apps for which we know the libraries they use.

Setup. We downloaded all 193 APKs from Maven Central [32] for which the build file (POM file) documents the used libraries. We obfuscated these APKs with *DexGuard* [21], an extension of *ProGuard* [6]; the obfuscator integrated into the Android development environment and recommended by the Android developer board [23]. Compared to *ProGuard*, *DexGuard* adds more advanced obfuscation techniques that we have also seen; in particular, string encryption and fake types. We use *DexGuard* with the four preset configurations: *Renaming*, *Optimization*, *String encryption*, and

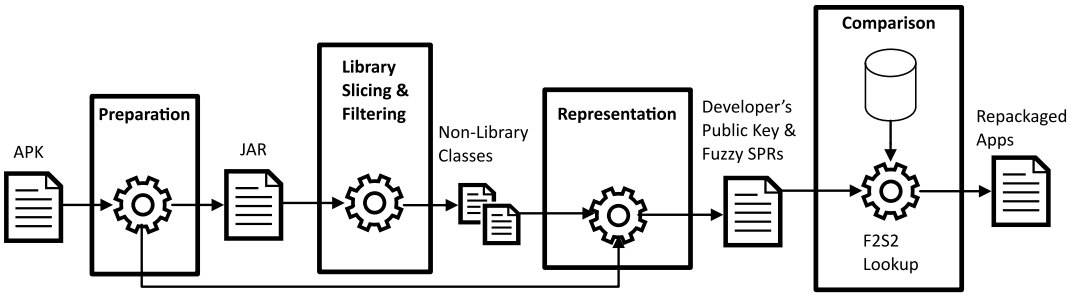

Figure 3: Toolchain to Identify Repackaged Apps

all of them *Combined*. DexGuard's default configuration enforces *Renaming*, because of that *Renaming* is always enabled.

During the obfuscation process *DexGuard* generates a detailed mapping which identifies the origin of every method. Using this information, along with the information about the libraries stored in the APK's POM file, we are able to assess whether a method is correctly classified as belonging to a library.

Evaluation. We filtered all methods (≈ 26%) that have less than 10 instructions (e.g., simple getters and setters or default constructors); such methods are indistinguishable after name mangling. After that, we checked for each method whether it is a library method or not and at which abstraction level (cf. Algorithm 1) the method is classified as a library method.

Results. Figure 4 shows how often *LibDetect*'s method classification was able to correctly classify a method as belonging to a library. For each configuration of *DexGuard*, the percentage of found methods to total methods per project is visualized using standard box-plots. In case of *Name Mangling* we are able to correctly classify more than 95% of all methods. And even if all obfuscation techniques are combined we still identify over 70% of all methods correctly. These results show that the approach provides a good starting point for a library detection.

Figure 5 shows the importance of each representation. It depicts at which abstraction level/representation an obfuscated library method was detected. As expected, BC and AR could not find any library methods. Both rely on names, which are changed by the applied obfuscations in all configurations. However, these representations are still valuable to identify methods that were not obfuscated – which are still common in real obfuscated apps.

Interestingly, even if all obfuscation techniques are combined the nameless representation (NR) already enables us to correctly classify a method in the vast majority of cases (> 95%); this makes NR the most relevant representation. Nevertheless, there are also cases, where SPR and fuzzy SPR are needed to identify obfuscated library methods and these cases increase by 0.5% per obfuscation technique. Overall, the results indicate that our design decision to consider the representations in increasing order of their level of abstraction is helpful.

6.2 Library Detection

In this section, we present the results of comparing *LibDetect* with other library detection approaches.

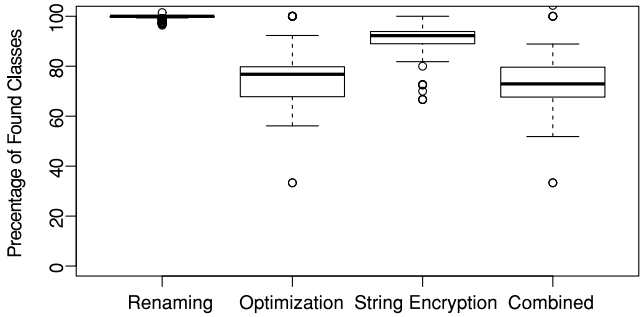

Figure 4: Detection Rates for Different *DexGuard* Configurations.

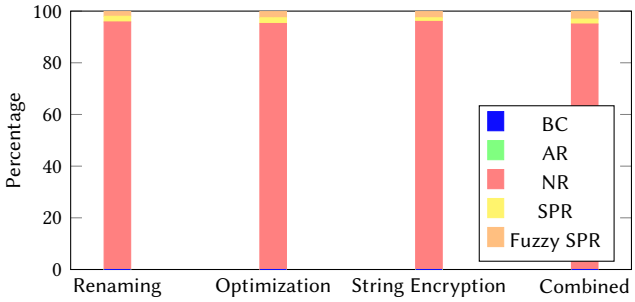

Figure 5: Relevance of Representations

Setup. We collected 8,000 Android related libraries: ≈7,000 from Maven Central and ≈1,000 additional JARs collected manually by searching for package names from the common-library list of Li Li et al. [29] and the package names of *LibRadar*'s database [31] (cf. short description of *LibRadar* in 4.1). The Maven Central JARs were collected by analyzing the latest versions of the POM files with dependencies to Android APIs (keyword "android" in the group ids of the dependency). Using all 8,000 libraries we build the reference database as described in Section 5.2.

For the evaluation of *LibDetect* in the wild, we randomly selected 1,000 apps (99% confidence level; 5% confidence interval) from five app stores (Anzhi, Google Play, App China, HiApk, and Freewarelovers) and measured the precision and recall of *LibDetect*, the common library white list (*Common Libraries*) by Li Li

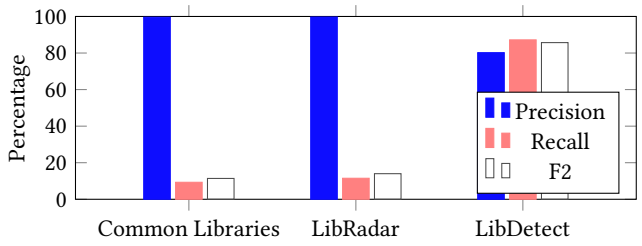

Figure 6: Average Precision and Recall of the Different Library Detection Approaches

et al. [29] and *LibRadar* [31]. We chose five different app stores to avoid biases such as only small apps (Freewarelovers) or only language-dependent apps (Anzhi). To determine the ground truth, we first identified the libraries used by the apps through manual code inspection. This enables us to assess both precision and recall.

Results. Figure 6 shows the average precision and recall of each approach. *Common Libraries* has an average precision of 99.6 % and a recall close to 9.3 % *LibRadar* has an average precision of 99.7 % and recall of 11.5 %. *LibDetect* has an average precision of 80.2% and the average recall is 87.2 %.

Discussion. Given that library code is the main reason for false positives of code-based repackage-detection approaches, identifying as many library classes as possible (i.e., a high recall) is very important. To reflect this, Figure 6 shows the harmonic-balanced F_2-measure, which weights recall higher than precision. We find that *LibDetect* identifies the majority of library code (F_2-measure of 85.7%) and that it significantly outperforms the state-of-the-art tool *LibRadar* (**RQ2**). *LibRadar*'s low recall 11.5% is due to its package-level abstraction. As discussed above, the tool misses library classes that are moved across package boundaries.

A careful analysis of the false positives/negatives of *LibDetect* revealed that the false positives are primarily due to `Activity` and `Listener`-classes that occur often in UI-intensive apps and which can be found in app code as well as in library code. These classes often have very similar functionality and differ only in their names. Hence, they are indistinguishable to *LibDetect*, because the (original) names are no longer available. *LibDetect*'s false negatives are caused by the filtering of potentially obfuscated data-container classes of libraries; in general, a container class primarily defines several fields along with respective getters and setters and, as discussed, such short methods are filtered.

6.3 Repackage Detection

For the evaluation of *CodeMatch*, we collected all app descriptions and the respective developer's public keys (to differentiate between them) from an archive of the Google Play Store [27] and fuzzy-hashed the descriptions of each app with SSDEEP [28]. Afterwards, we compared all fuzzy-hashed descriptions pairwise and randomly selected 1,000 app pairs, which had at least 90%-similar descriptions and were signed with different public keys. To identify which of these pairs are actual repackages, we installed and executed the app pairs on the emulator LeapDroid [35]. Subsequently, we manually tagged them (as truly repackaged or not) by checking their similarity in the following process:

(1) We checked whether the loading screen and main view have the same structure and the same icons.
(2) In case of doubts, we then checked the actions, which could be performed from the main view.
(3) If we were still not sure about the tag, we generated fake accounts, installed needed additional software, and performed all actions that were possible.
(4) If the above steps were insufficient, we also performed a visual inspection of the de-compiled code. If the code of both apps was similar, we classified the apps as repackages; otherwise as "me-too" products.

Following this process, we manually identified 377 app pairs as actual repackages (true positives) and used all 1,000 app pairs to evaluate the precision and recall of *CodeMatch*, *FSquaDra* [42], *ViewDroid* [41], *DroidMOSS* [46], and a centroid-based approach [10] to which we refer as *Centroid*.

To assess the effect of the different library-detection approaches on the overall repackage detection, we first removed all *library apps* using *LibDetect*, as described in Section 5.3. These apps are generally falsely identified as repackaged apps by the other approaches as they have no specialized support for this kind of apps and we want to avoid to bias the results. We exclude library apps (mostly < 300 LOC excluding library code), because it is practically impossible to determine whether such apps are illegitimate repackages. For example, most wallpaper apps only differ in the image, but are generated apps, rather than repackages.

After that, we executed all repackage detectors that use some library detection with all library detection approaches described in Section 6.2: *LibRadar* (LR), *LibDetect*(LD), and the *Common Libraries* white list (WL). If a repackage detector was unable to classify a repackaged app, we counted it as false negative.

Results. Figure 7 presents the average precision and recall for the different combinations of repackage- and library detection tools. The results are grouped by the repackage detection approaches and sorted by the average recall.

Combining *LibDetect* with any previous repackage-detection approach results in an average precision of 86%, which is better than all combinations of the respective repackage-detection approach and any other library detection.

All combinations of *CodeMatch* with an existing library-detection approach performed at least as good as the combinations of the respective library detection with any other repackage-detection approach. However, *CodeMatch* + *LibDetect* achieves the highest average recall and F_1-score.

Overall, we can conclude that *LibDetect* leads to significant better repackage detection results and significantly improves the results of *Centroid* when compared with the original results. Nevertheless, *CodeMatch* + *LibDetect* gives the best precision and recall.

6.4 App Data-Provision & Insights

We used *CodeMatch* to assess the problem of repackaged apps in the wild. For that, we downloaded 46,537 apps from five different Android app stores and analyzed how many apps are repackaged

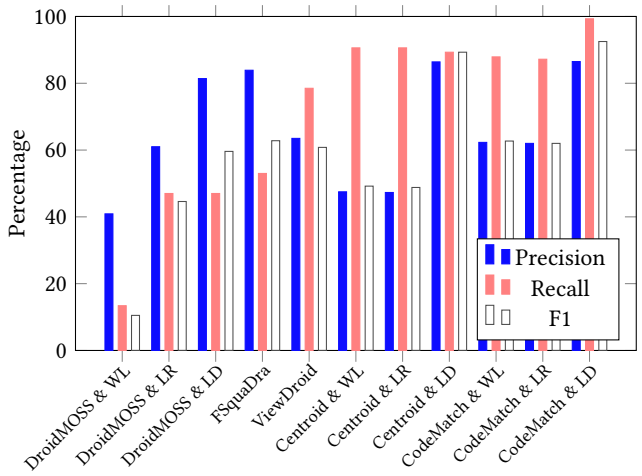

Figure 7: Average Precision, Recall and F_1-Measure of the Different Repackage Detection Approaches

Table 4: Analyzed Android Apps from Five App Stores

App Store	Apps	Lib Apps		App Maker		Repackaged	
Anzhi [1]	18,889	1,707	9.0%	72	0.4%	2,757	14.6%
Google Play [27]	17,751	371	2.1%	1,018	5.7%	3,510	19.8%
App China [2]	4,577	1,260	27.5%	21	0.5%	396	8.7%
HiApk [4]	4,472	1,106	24.7%	6	0.1%	608	13.6%
Freewarelovers [3]	848	583	68.8%	0	0%	20	2.4%
Total	**46,537**	**5,027**	**10.8%**	**1,117**	**2.4%**	**7,291**	**15.7%**

across the individual stores. Table 4 shows the distribution of the apps across the stores. We obtained the APKs from AppChina, HiApk, and Freewarelovers using DroidSearch [34].

Table 4 reveals that up to 5.7% of the apps are created using App makers (see Section 5.3) and—depending on the store—between 2.1% and 68.8% of the apps are *library apps* with less than 300 lines of app (glue) code. We filtered these apps before computing the number of repackaged apps. Overall, 7,291 (15.7%) of the 46,537 apps are repackages. The problem seems to be most relevant for the Google Play store; nearly 20% of the apps are repackages.

7 THREATS TO VALIDITY

In the following, we discuss threats to validity related to our library-detection and repackage-detection experiments.

We construct *LibDetect*'s reference database from all libraries in *LibRadar*'s database, the latest Android related libraries from Maven Central, all libraries from a public white lists [29] and a popular-libraries list [11]. Nevertheless, the database may not contain all libraries used by apps in our evaluation datasets. In this case, *LibDetect* may fail to identify some library code and *CodeMatch* may include this library code in the repackage detection. Both could lead to other results than the reported ones, but – given the size and quality of the data set – the overall error should be negligible.

Assembling a large, up-to-date reference database for *LibDetect* might be unpractical. We argue, since we include only public libraries listed in public databases and lists, that assembling the database could be fully automated, which would allow frequent updates without manual effort.

The evaluated apps from Maven Central that we chose for library detection may not be representative for apps in general. We chose these apps because they document the libraries that they depend on. This allowed us to construct a ground truth for the evaluation of the library-detection tools (see Section 6.2). The experiment shows, how well *LibDetect* can discover (library) code embedded in an app, after the entire app has been obfuscated. The impact of specific apps on the results of these experiments should be rather small.

To evaluate the impact of obfuscation on library detection we used the obfuscator *DexGuard*, which applies renaming, optimization, shrinking, and string encryption. To the best of our knowledge no existing library detection handles more advanced techniques.

Our ground-truth dataset for repackage detection may not be representative for apps in general. We chose our sample from the apps of five different app stores. Therefore, we first filter the set of all pairs of apps from the stores for pairs with similar descriptions, as described in Section 6.3. Then we selected a random sample of 1,000 app pairs, which is representative at a confidence level of 99% and a confidence interval of 5%. It is possible that this sampling strategy introduces a bias, because repackages with dissimilar descriptions are left out. However, the intent behind repackaging is to get users to install the repackaged app instead of the original app, which makes it likely that a similar description is used. Furthermore, the app candidates were classified as repackaged or not-repackaged through a manual review by one of the authors (cf. Section 6.3). It is possible that our primary criterion, the similarity of the apps' user interfaces, may lead to some wrong classifications. Due to the high effort of reviewing 1,000 app pairs, it was infeasible to confirm the review results by additional reviewers.

8 CONCLUSION & FUTURE WORK

We presented an approach to detect repackaged apps that relies (1) on a new advanced library detection approach and (2) the fuzzy hashing of the app's code to handle advanced code obfuscations. In both cases, we rely on a hierarchy of five different code representations. Each higher-level representation abstracts over additional parts of the code to counter more advanced obfuscation techniques. The evaluation demonstrated the effectiveness of the approach in matching code obfuscated using advanced obfuscators. The evaluation further revealed that – depending on the level of obfuscation – the different representations are necessary to match code.

When we applied our tool *CodeMatch* to real world apps taken from Android App Stores, we were able to determine that 15% of the apps can be found in repackaged form across different app stores.

Our implemention is available for download [21].

ACKNOWLEDGMENTS

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) within the Software Campus project *Eko*, grant no. 01IS12054, the DFG as part of CRC 1119 CROSSING, as well as the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP.

REFERENCES

[1] 2017. Anzhi App Marketplace. (2017). Retrieved 01/11/2017 from http://www.anzhi.com/

[2] 2017. App China App Marketplace. (2017). Retrieved 01/11/2017 from http://www.appchina.com/

[3] 2017. Freeware Lovers App Marketplace. (2017). Retrieved 01/11/2017 from http://www.freewarelovers.com/

[4] 2017. HiApk App Marketplace. (2017). Retrieved 01/11/2017 from http://www.hiapk.com/

[5] Alexandr Andoni and Piotr Indyk. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*. IEEE, 459–468.

[6] Eric Lafortune at GuardSquare. 2017. ProGuard. (2017). Retrieved 01/11/2017 from http://proguard.sourceforge.net/

[7] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 356–367.

[8] Apps Builder. 2017. Apps Builder. (2017). Retrieved 01/11/2017 from http://www.apps-builder.com

[9] Jian Chen, Manar H Alalfi, Thomas R Dean, and Ying Zou. 2015. Detecting Android Malware Using Clone Detection. *Journal of Computer Science and Technology* 30, 5 (2015), 942–956.

[10] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 175–186.

[11] CodePath. 2017. Must-Have Libraries. (2017). Retrieved 02/24/2017 from https://github.com/codepath/android_guides/wiki/Must-Have-Libraries

[12] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 184–196.

[13] Licel Corporation. 2016. DexProtector Android Obfuscator. (2016). Retrieved 01/20/2017 from https://dexprotector.com

[14] Licel Corporation. 2016. Stringer Java Obfuscator. (2016). Retrieved 01/20/2017 from https://jfxstore.com/stringer/

[15] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security–ESORICS 2012*. Springer, 37–54.

[16] Jonathan Crussell, Clint Gibler, and Hao Chen. 2013. Scalable semantics-based detection of similar android applications. In *Proc. of Esorics*, Vol. 13. Citeseer.

[17] Yang Cuixia, Zuo Chaoshun, Guo Shanqing, Hu Chengyu, and Cui Lizhen. 2015. UI Ripping in Android: Reverse Engineering of Graphical User Interfaces and its Application. In *2015 IEEE Conference on Collaboration and Internet Computing (CIC)*. IEEE, 160–167.

[18] D Eastlake 3rd and Paul Jones. 2001. *US secure hash algorithm 1 (SHA1)*. Technical Report.

[19] Michael Eichberg and Ben Hermann. 2014. A software product line for static analyses: the OPAL framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. ACM, 1–6.

[20] Parvez Faruki, Vijay Ganmoor, Vijay Laxmi, Manoj Singh Gaur, and Ammar Bharmal. 2013. AndroSimilar: robust statistical feature signature for Android malware detection. In *Proceedings of the 6th International Conference on Security of Information and Networks*. ACM, 152–159.

[21] Leonid Glanz. 2017. CodeMatch Artifacts. (2017). Retrieved 06/30/2017 from http://www.st.informatik.tu-darmstadt.de/artifacts/codematch/

[22] Hugo Gonzalez, Natalia Stakhanova, and Ali A Ghorbani. 2014. Droidkin: Lightweight detection of android apps similarity. In *International Conference on Security and Privacy in Communication Systems*. Springer, 436–453.

[23] Google. 2017. Android Developers http://developer.android.com/tools/help/proguard.html. (2017). Retrieved 01/11/2017 from http://developer.android.com/tools/help/proguard.html

[24] Google. 2017. Enjarify. (2017). Retrieved 01/11/2017 from https://github.com/google/enjarify

[25] Wenjun Hu, Jing Tao, Xiaobo Ma, Wenyu Zhou, Shuang Zhao, and Ting Han. 2014. Migdroid: Detecting app-repackaging android malware via method invocation graph. In *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–7.

[26] Smardex Inc. 2017. Allatori Java Obfuscator. (2017). Retrieved 02/20/2017 from http://www.allatori.com

[27] Jake J. 2017. PlayDrone Archive Snapshot 10/31/2014. (2017). Retrieved 01/11/2017 from http://archive.org/download/playdrone-snapshots/2014-10-31.json

[28] Jesse Kornblum. 2006. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation* 3 (2006), 91–97.

[29] Li Li, Jacques Klein, Yves Le Traon, et al. 2016. An investigation into the use of common libraries in android apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 403–414.

[30] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. Libd: Scalable and precise third-party library detection in Android markets. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 335–346.

[31] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: fast and accurate detection of third-party libraries in Android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 653–656.

[32] Service mark of Sonatype Inc. 2017. Maven Central. (2017). Retrieved 01/11/2017 from http://search.maven.org/

[33] GuardSquare nv. 2017. DexGuard Android Obfuscator. (2017). Retrieved 02/20/2017 from https://www.guardsquare.com/en/dexguard

[34] Siegfried Rasthofer, Steven Arzt, Max Kolhagen, Brian Pfretzschner, Stephan Huber, Eric Bodden, and Philipp Richter. 2015. Droidsearch: A tool for scaling android app triage to real-world app stores. In *Science and Information Conference (SAI), 2015*. IEEE, 247–256.

[35] Huan Ren and Huihong Luo. 2017. LeapDroid. (2017). Retrieved 01/11/2017 from http://www.leapdroid.com

[36] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic reverse engineering of malware emulators. In *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 94–109.

[37] PreEmptive Solutions. 2017. DashO Java Obfuscator. (2017). Retrieved 02/20/2017 from http://www.preemptive.com/products/dasho

[38] Xabier Ugarte-Pedrero, Igor Santos, Pablo G Bringas, Mikel Gastesi, and José Miguel Esparza. 2011. Semi-supervised learning for packed executable detection. In *Network and System Security (NSS), 2011 5th International Conference on*. IEEE, 342–346.

[39] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. 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*. ACM, 71–82.

[40] Christian Winter, Markus Schneider, and York Yannikos. 2013. F2S2: Fast forensic similarity search through indexing piecewise hash signatures. *Digital Investigation* 10, 4 (2013), 361–371.

[41] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, 25–36.

[42] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. 2014. FSquaDRA: fast detection of repackaged applications. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 130–145.

[43] Min Zheng, Mingshen Sun, and John CS Lui. 2013. Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 163–171.

[44] Wu Zhou, Zhi Wang, Yajin Zhou, and Xuxian Jiang. 2014. Divilar: Diversifying intermediate language for anti-repackaging on android platform. In *Proceedings of the 4th ACM conference on Data and application security and privacy*. ACM, 199–210.

[45] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. 2013. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*. ACM, 185–196.

[46] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM, 317–326.

[47] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets.. In *NDSS*, Vol. 25. 50–52.