ApkDiff: Matching Android App Versions Based on Class Structure

Robbe De Ghein*
robbedeghein@gmail.com
Unaffiliated
Ghent, Belgium

Bert Abrath
Bert.Abrath@UGent.be
Ghent University
Ghent, Belgium

Bjorn De Sutter
Bjorn.DeSutter@UGent.be
Ghent University
Ghent, Belgium

Bart Coppens
Bart.Coppens@UGent.be
Ghent University
Ghent, Belgium

ABSTRACT

Reverse engineering an application requires attackers to invest time and effort doing manual and automatic analyses. When a new version of the application is released, this investment could be lost completely, if all the analyses had to be re-done. The gained insights into how an application functions might be transferred from one version to the next, however, if the versions do not differ too much. Diffing tools are thus valuable to reverse engineers attempting to transfer their knowledge across versions, as well as to defenders trying to assess this attack vector, and whether or how much a new version has to be diversified. While such diffing tools exist and are in widespread use for binary applications, they are in short supply for Android apps.

This paper presents ApkDiff, a tool for diffing Android apps based on the semantic features of the class structure. To evaluate our tool we selected 20 popular financial apps available in the Google Play Store, and tracked their version updates over eight months. We found that on average 79% of all classes had a unique match across version updates. When we consider only classes for which we detect explicit obfuscations being applied (by applying heuristics on their identifiers), we still find that we can find a match for 56% of the classes (ranging from 23% to 85%), suggesting that these obfuscated apps are not resilient to our matching strategies. Our results suggest that ApkDiff provides a valuable approach to diffing Android apps.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**.

KEYWORDS

reverse engineering, matching program versions, bytecode, classes

*Work performed while at Ghent University.

Checkmate '22, November 11, 2022, Los Angeles, CA, USA

ACM Reference Format:
Robbe De Ghein, Bert Abrath, Bjorn De Sutter, and Bart Coppens. 2022. ApkDiff: Matching Android App Versions Based on Class Structure. In *Proceedings of the 2022 ACM Workshop on Research on offensive and defensive techniques in the context of Man At The End (MATE) attacks (Checkmate '22), November 11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3560831.3564257

1 INTRODUCTION

Reverse engineering an obfuscated application to acquire insights can take a substantial amount of time and effort. While analysing the application, new versions can be released, potentially making obsolete the original reverse engineering results and insights. There are many different ways in which the code can change across versions: not only can code change because features are added or bugs are fixed; but obfuscations can also be applied, potentially in a non-deterministic way. In the latter case, all the insights acquired by the reverse engineer might be rendered useless because the equivalent code cannot easily be found in the new version. From a defender's point of view, ideally the reverse engineer cannot transfer any insights from the outdated version, forcing them to redo all their work on the new version.

Nevertheless, techniques exist to aid reverse engineers in transferring knowledge across different versions [17, 32, 34]. Such techniques try to match certain objects (functions, basic blocks, instructions, ...) from one program version to another program version, after which annotations on these objects (such as potential function names, recovered semantics, ...) can be transferred through these matches. However, these techniques are typically focused on matching binary code, and often even unobfuscated binary code. A related technique is clone detection, but such techniques focus on identifying re-used larger code fragments.

Although diffing tools based on such techniques are valuable to reverse engineers, they are valuable to defenders as well. When protecting an application, a security expert might use such a tool to assess diffing as an attack vector, to decide whether or how much a new version has to be diversified. While diffing tools for binary applications exist and are widely used, they are in short supply for Android apps.

In this paper, we design and implement a technique we call ApkDiff, named so because it diffs APKs (Android Packages), that tries to match bytecode versions of apps, even when these versions have been obfuscated and/or diversified. This technique can be used to

evaluate how much—if any—information a reverse engineer can transfer across app versions. Our technique tries to link *classes* between two versions. We will try to match classes based on their class structures: the field signatures, the method signatures, interfaces, modifiers, etc. We evaluate the application on a sample set of real Android apps: 20 financial apps in Belgium and the Netherlands. These apps were tracked for over 8 months and every update was extracted, allowing us to evaluate the attack strategy discussed above.

The contributions of this paper are as follows:

- We design and implement ApkDiff, a technique to match classes in bytecode between different versions of an app;
- We evaluate the effectiveness of ApkDiff on 20 financial apps for Android from the Google Play store;
- We release ApkDiff as an open source tool, available at https://github.com/csl-ugent/apkdiff

.

The rest of this paper is structured as follows. Section 2 provides some background on the bytecode obfuscations our technique will be able to circumvent. Next, Sections 3 and 4 describe the design and implementation of our technique, respectively. This is followed by the evaluation, where we first describe how we gathered the application data in Section 5, and subsequently the evaluation itself in Section 6. This is followed by a discussion of related work in Section 7, after which Section 8 provides a conclusion.

2 BACKGROUND

We briefly discuss some basic anti-reverse engineering protections that are commonly used in Android applications. These basic protections are obfuscations and consist of identifier renaming and modifying the package hierarchy.

2.1 Identifier Renaming

Renaming symbolic identifiers is one of the most basic forms of protection. The idea is the same as that behind stripped binaries: remove metadata that is not necessary for execution, but that is useful for other purposes such as debugging. The Android Runtime (ART) does not care if a class is called "PrintWriter", "a" or "Ö!Üé⊠|⊠â". Semantically meaningful identifiers are only useful for developers. There are a couple of ways to rename identifiers. The most popular methods create shrunken names, random names, and misleading names. Shrinking a name changes the identifier to a short and meaningless name (such as "a"). Random names are usually a bit longer, but are still meaningless. They have no semantic meaning and often contain unusual Unicode characters. Misleading names are normal-looking identifiers that replace the original names, e.g. a class named Account is renamed to Biometric. This is a hard case to deal with, both as a human analyst as well as when using automated tools, as it is challenging to detect when this kind of renaming has happened. The renamed identifiers can be package names, class names, function names, etc.

2.2 Modifying Package Hierarchy

A second basic obfuscation is to modify the package hierarchy. Like identifiers, the package hierarchy is insignificant to code execution. Most information that the package hierarchy provides, is the semantic grouping of different classes. A tree-like structure makes it easier for a developer to work with a large number of classes by grouping related classes together. This is mostly a feature for the developers. As such, it is possible to remove or change this information to make it harder for an attacker to determine the semantics of every class. There are two major methods to modify a package hierarchy: hierarchy flattening and repackaging.

In hierarchy flattening, a part of the package tree is flattened by removing all subpackages and recursively moving all classes inside this part of the tree up to the closest parent package that is not removed [37]. Note that no classes are removed, only packages.

Repackaging classes is similar to hierarchy flattening. While repackaging classes, a part of the package tree is completely removed and all classes inside this part are relocated to a new package, often a root level package. The biggest difference between repackaging and hierarchy flattening is that repackaging does not happen in-place, but creates a completely new package and actually removes a part of the hierarchy instead of flattening it.

Many Java/Android/bytecode obfuscators exist, ranging from relatively simple ones that restrict themselves to shrinking names, to complex ones which also modify the bytecode itself to make it harder to reverse engineer. Examples include, ProGuard [12], R8 [1], DexGuard [4], DashO [3], SecureIt [13], JFuscator [10], KlassMaster [11], Obfuscapk [14], GUJTO [8], ...

3 MATCHING ALGORITHM DESIGN

We designed and implemented a bytecode-based matching approach which we call ApkDiff. In this section, we give an overview of ApkDiff's design. Our goal is to match classes between two versions. The data available in the bytecode can be classified into two categories: the metadata and the bytecode instructions. We will not try to match instructions directly, and instead focus on features based in part on the metadata of classes. The exact features our implementation of ApkDiff uses will be detailed in Section 4 on the implementation of ApkDiff, but it includes information on field types and the package hierarchy.

We optionally allow the use of identifiers (such as class names) as a feature, but when these are included, we employ heuristics to ignore obfuscated identifiers. The exact heuristics we employ are described in the implementation section as well. Note that while these specific rules could always be extended and fine-tuned, to increase their accuracy, being able to detect at least identify the obfuscations as applied by ProGuard will already be useful, as Wang et al. [39] showed that about 88% of obfuscated apps chose ProGuard as their only obfuscator in 2016, and 70% of the apps used the default ProGuard configuration in 2016.

To match classes, ApkDiff will *not* compare all classes in a pairwise fashion to look for exact matches on all features. This would not work, because more often than not, it is impossible to know with certainty whether or not two classes are an exact match due to obfuscations. Some classes have an identical class structure and thus cannot be distinguished on their class structures alone. Instead, ApkDiff works by a process of elimination: it finds *differences*, which are easier to find. For example, when two classes have a different number of functions, the classes cannot be an exact

match. If ApkDiff cannot find any distinct differences, it will mark the classes as a *potential match*.

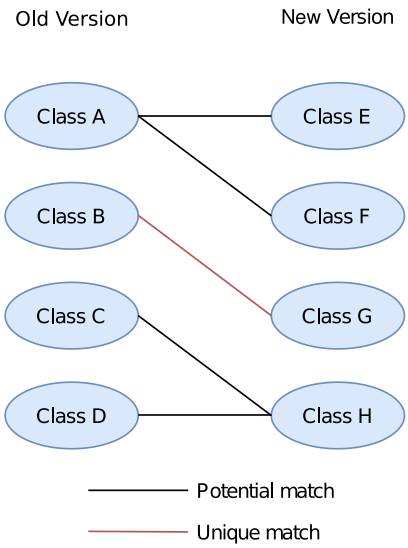

Figure 1: Example of a unique match between potential matches.

When, after comparing all classes, there is a pair of two classes which are potential matches with one another, and only each other, but are distinct from all other classes, we call this potential match a *unique match*. A visual representation of this is shown in Figure 1. Thus, to find the unique matches, all classes from one version are compared to all classes from the other version and the potential matches are stored. When all potential matches are found, unique matches are then those where a single class of the old version is a potential match of a single class in the new version. This is similar to how BinDiff operates on binaries [24]: BinDiff tries to find the same pieces of code in two binaries by mapping functions to a 3-tuple feature and then checking per tuple if exactly two functions are mapped: one from each binary. If two functions are mapped, they must refer to the same function and thus they are a match. To increase the probability of finding unique matches, we use package tree information to split up the classes into different sets.

A high-level overview of the algorithm is shown in Figure 2. First, classes are partitioned in different sets based on the package tree information. Then, the algorithm consists of an iterative approach. The core loop consists of finding unique matches, and then propagating this information to extend the matches. This is described in more detail in Sections 3.1 and 3.2, respectively. The outer loop of the algorithm consists of iteratively reducing the amount of sets the (remaining) classes are partitioned into. Finally, as a last-ditch effort, all classes are put into a single remaining set, after which we perform a last iteration of our algorithm. This handling of sets, and how package information is used, is described in Section 3.3.

3.1 Potential and Unique Matches

A potential match is a link between two classes. These classes are marked as potentially equal. For example, consider matching a class which has only an `int` field to a class which has only a `float` field. As these are two distinct primitive types, we know with certainty that these classes are different. However, when we compare two classes that each contain a single object reference field, these fields might possibly refer to the same class, but we are not certain. Thus, this pair of classes is marked as potentially the same.

Note that we assume that the number of functions in a class does not change across versions. This is of course not the case if functionality was added/removed or if advanced obfuscation techniques were used such as function merging or function inlining. However, even if the number of functions does regularly change, the algorithm can be altered to keep this into account, e.g., we could match classes with added functionality by also matching class structures that are completely encapsulated within other class structures. We did not implement such extensions for this paper, however.

3.2 Extending Matches

We can leverage knowledge of existing matches to find new matches. This is similar how BinDiff can extend matches. In the case of BinDiff, for two functions, it compares incoming and outgoing edges of the call graph using the same scheme as before, but now only the functions related to the edges of the graph are compared. By using the call graph information, the set of functions that have to be mapped is significantly reduced and the probability of a unique mapping is increased. For ApkDiff, we take a similar approach to extend matches. As our approach is class-based, rather than focus only on the call graph, we extend matches across all features related to classes. For example, the field types are references to other classes. We leverage these features to try and find new matches, thus extending the match.

Consider the example in Figure 3. Here, ApkDiff finds five potential matches, one of which is unique (classes A and D). This is shown on the left of the figure. Classes B and E are superclasses of classes A and D respectively. This gives more confidence in the potential match between classes B and E as opposed to the potential match between classes B and F. Because class hierarchy modifications such as Class Hierarchy Flattening [25–27] are possible, ApkDiff does not match the superclasses without also checking the class structure: a potential match must still be found before ApkDiff considers these superclasses as a match. A match is extended when both extended classes (these are the superclasses in this example) are also a potential match. This new match is directly considered a unique match. Any other potential match does not have this extra relationship. This is shown on the right of Figure 3.

Whenever a new match is found by extending a unique match, this new match can possibly affect the remaining potential matches. We can see this our previous example in Figure 3. Here, the potential match between classes C and F becomes unique after the extended match between classes B and E is found. Extended matches

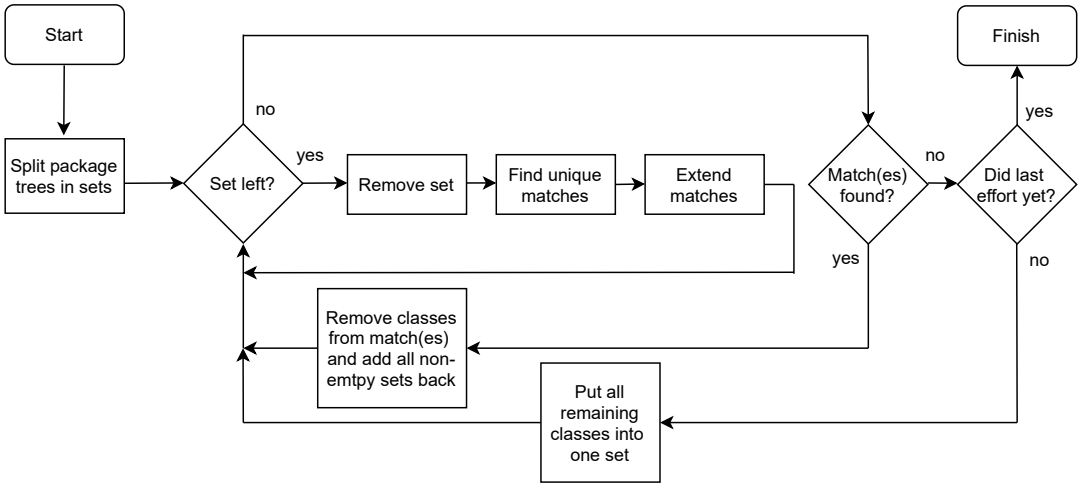

Figure 2: A high level flowchart of the complete algorithm.

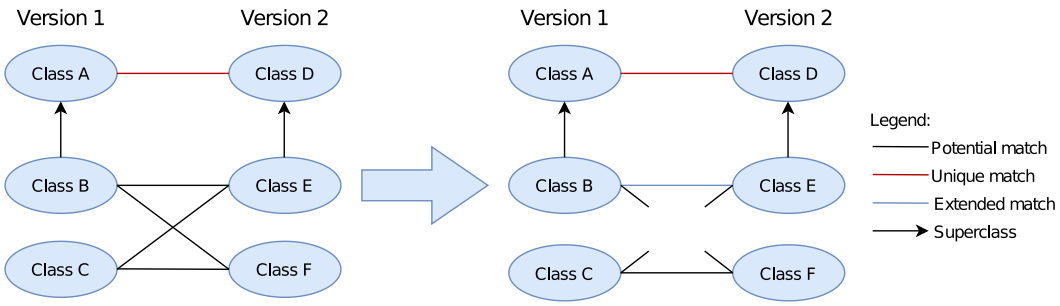

Figure 3: Example of an extended match creating new unique matches. Here a unique match is found between classes A and D while the other potential matches are not unique. However, from the extra information regarding the superclasses, the match between classes B and E can be marked unique too. When we scan for unique matches again, the match between classes C and F becomes unique.

bypass the uniqueness requirement because of more valuable information: the connection with an existing unique match. By bypassing this requirement, this new unique match can influence the remaining pool of potential matches. Removing all potential matches related to this new unique match could possibly create new unique matches within the remaining potential matches. In our example in Figure 3, we see that the new unique match between classes B and E removes the potential matches between B and F and between C and E, which makes the potential match between C and F unique once we add the extended match between B and E.

Whenever a match extends to find a new unique match, ApkDiff scans for new unique matches in the remaining potential matches. When this results in new unique matches, these new unique matches can be extended again. This leads to a fixed-process iteration process which repeats itself until no more matches are found.

3.3 Package Information

One of the features we employ is the package hierarchy, which provides a semantic grouping for the classes. When ApkDiff tries to find a match for a certain class, it will start by looking at the classes from the other version that are in the same package. An example is shown in Figure 4. Here we can find a similar package hierarchy for both versions. When trying to find a match for class A in the package `com.google` from version 1, ApkDiff starts by looking at the classes inside package `com.google` from version 2: classes F and G. This strategy can be applied to all packages and subpackages. By using the package information, we limit the possible classes that can match which increases the probability for a unique match. The package in the other version is only a starting point because the package hierarchy can be modified by standard tools such as ProGuard [12]. Packages can be flattened in-place or repackaged to another part of the hierarchy. The only feature for a package is its identifier. Thus, package information can only be used if identifiers are used in our matching algorithm. If identifiers are not used, the package information is discarded.

3.3.1 Package Squeezing. Like any other identifier, the package identifier can also be obfuscated and renamed to something else. If

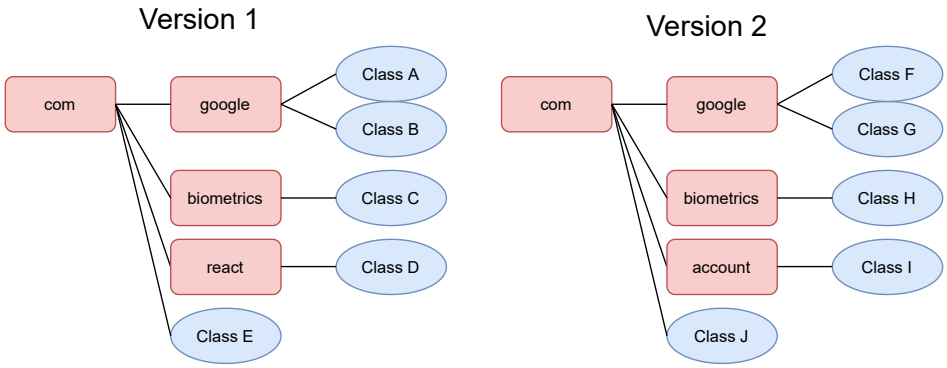

Figure 4: An example of package information that serves as a starting point when looking for matches. Using the package information, we see that a match for class A is likely to be found in the com.google package from version 2, as class A is also part of this package in version 1. The same idea can be applied for packages com.biometrics and com.

identifiers *are* used, this can be problematic when the same package identifier has been renamed to different identifiers across versions. When this happens, ApkDiff will look for matches in the wrong packages. To deal with these renamed identifiers, ApkDiff depends on heuristics to determine if an identifier is obfuscated, and if so, will not use the package identifier. The way ApkDiff deals with renamed packages is by removing the package from the package hierarchy and moving all its classes up one level in the package hierarchy. This is similar to ApkDiff itself applying hierarchy flattening obfuscation. The difference is that here, the subpackages are not removed but only the root package that is renamed/moved. We do this to only discard the information that is obfuscated. If the parent package is not obfuscated, this can still serve as a starting point to look for matches. We will refer to this technique as *package squeezing*.

Using the package information, we can split all the classes into smaller sets. Each set contains two groups of classes with one group per version. These sets of classes are semantically meaningful due to their package information and thus we have a higher probability of finding the correct match.

3.3.2 Last-Ditch Effort. When we have found all possible unique matches in the sets created from the package information, we will discard these sets and create one large set with all remaining classes from each version. This is a last effort to find more unique matches. We do this last effort because classes could have been repackaged and moved to somewhere else in the package hierarchy. When matching happens without using identifiers, the package information cannot be used and this last effort is done immediately.

4 IMPLEMENTATION

We implemented ApkDiff using the Soot framework, and made it publicly available at https://github.com/csl-ugent/apkdiff. In this section, we discuss in more detail some of the more interesting implementational aspects. First, in Section 4.1 we describe the features ApkDiff uses for its matching steps. Next, in Section 4.2, we describe the heuristics we employ to determine whether or not we consider an identifier to be obfuscated. Finally, in Section 4.3 we describe some steps we took to increase the performance of ApkDiff.

4.1 Available Features

The features we employ related to classes are based mostly on metadata which is mainly used by the JVM. It includes the identifiers (which is needed for reflection), access modifiers (private, public, etc.), the class package, etc. We also incorporate some features that are not directly available in the bytecode, but which we generate ourselves, based on the instructions. For example, ApkDiff constructs a call graph by examining the function calls across the bytecode instructions. Figure 5 gives an overview of all features that we use in ApkDiff. In blue ellipses we indicate the classes and interfaces which we try to match. The high-level subdivisions of features we can assign to them are indicated in purple rounded rectangles, which themselves are connected to the actual features we will try to match.

Some of these class features are also used to extend matches: the superclass, the implemented interfaces, the field types, the function parameter types and the function return type.

4.2 Heuristics for Identifiers

Perhaps the most obvious and interesting of these available features are the identifiers, and we will go into more detail on how we try to match these. Identifiers are present for classes, functions, fields, packages, etc. These identifiers are names and originate from the source code. They can offer a lot of information as they are semantically meaningful. However, identifiers should be handled carefully, as altering identifiers is one of the most basic forms of obfuscation in the Android ecosystem as we already discussed above. Ideally, we want a way to distinguish renamed identifiers from original ones. There is no perfect solution that captures all of these cases, in particular the misleading names, and as such, we designed two modes for matching classes: one where identifiers are used and one where they are not.

To support the matching mode that does utilize the identifiers, we propose a heuristic that tries to distinguish the renamed identifiers from the original ones. As most code is written in English [15],

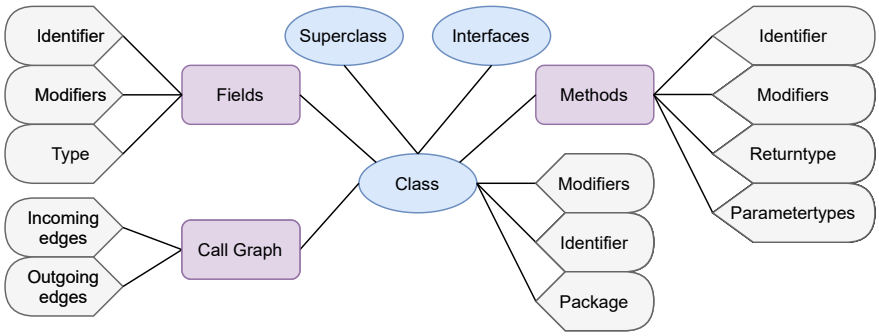

Figure 5: Available features in the bytecode of a class that form the class structure.

we assume that only ASCII symbols are used in the identifiers. Other symbols are usually from Unicode characters used in renaming identifiers or symbols that are not valid in source code but are valid in bytecode. This is of course not always true, as code in a foreign language does exist [2, 5, 9], but this is rather rare, and can be manually detected by a reverse engineer if needed. The use of ASCII symbols is also recommended by coding style guides such as the Google Java Style Guide [6] which states "identifiers use only ASCII letters and digits" or the Oracle coding convention [35] that says: "the prefix of a unique package name is always written in all-lowercase ASCII letters".

Short names also usually indicate obfuscations, e.g. a, b, ..., aa, ab, etc. as these are rarely semantically meaningful. However, we need to have some exceptions as some short names are actually common, like ID, IO or R, the latter being used in the context of Android resources. In our implementation, we marked every identifier consisting of only one or two symbols as obfuscated, except for these exceptions. The English language also does not have words with three or more repeating consonants [36], which means that repeating letters also indicate obfuscation. Another indication of a renamed identifier is reserved keywords, as these are not valid in source code but can occur in bytecode. We also detect some custom patterns, which we added to our heuristic after encountering them in applications from our dataset. For example, some class names we observed were of the form "CRC64+hash" where "hash" is an actual hash of some input (presumably the real name). Other hashes were also found in the applications, such as MD5 or SHA2.

In summary, we mark an identifier as obfuscated if it is:

- *not* a hardcoded exception (e.g. ID, IO, R), regardless of capitalization
- is a Java keyword
- is a known hash pattern
- is only one or two symbols
- has three or more repeating consonants
- has symbols that are not ASCII, the dollar sign (used in compiled inner class names) or an underscore

Everything else is marked as unobfuscated.

Note that being marked as unobfuscated by this heuristic does not necessarily mean that classes that were marked as unobfuscated are not obfuscated. The obfuscations could also not be detected by our heuristic.

4.3 Increasing Performance

The design as described earlier favours precision over speed. This makes ApkDiff slower but more accurate than signature-based matching schemes. ApkDiff performs a more fine-grained analysis by comparing *all* features individually, instead of comparing a single signature per class. Still, even though signatures are not used for the matching itself, they can be used to speed up the matching process. We do this by using signatures to exclude classes that would not have matched in the fine-grained analysis anyway. For example, with our approach, matches need to have the same number of fields and methods. For every class, we can compute a signature consisting of these two numbers and insert all classes into a hash table based on their signatures. When we are looking for the potential matches of a class that has two fields and four functions, we can index the hash table using its signature to find all other classes that also have two fields and four functions, skipping the fine-grained analysis for many of the classes. This saves a substantial amount of time.

ApkDiff uses the following features in its hash: field types, the parameter types and their modifiers for the signature (private, protected, public, static, final, etc.), the return type, the number of parameters, and the parameter types.

Note that this signature implementation is specific to what is considered a match. If classes with a different number of functions can match, the example above will throw away classes that could still match. As such, the signature should adapt to the exact matching specifications. To deal with uncertainties, it is possible to use a placeholder. For example, when type information is used, we can keep the primitive types while using a placeholder for Object types. This way no classes are thrown away based on uncertainties. The function parameter types are also sorted so that the signatures do not rely on their exact order. The order of parameters is a change that can be made to the class structure without changing the semantics of that function.

Finally, it is worth noting that large parts of the algorithm can occur in parallel. The largest portion of the time is spent looking for potential matches doing a fine-grained analysis. Since no data is altered when looking for potential matches, we can and do analyse multiple classes at once without synchronization issues.

5 DATA GATHERING AND DATA SET

Rather than evaluate ApkDiff on some self-created toy examples, we wanted to evaluate it on real Android applications. In this section, we describe how we gathered this data set.

There are a variety of ways to get access to an APK of an Android application and this section will list the two main methods used: using the Android Debugging Bridge (ADB), and using the Google Play Store API.

Android Debugging Bridge. When developer mode is enabled on an Android device, the end user can extract APKs using a tool called Android Debugging Bridge (ADB). ADB is developed by Android itself and is included in Android Studio. In this paper, ADB was the main method of gathering APKs. Automatic updates were disabled and APKs were extracted before and after every update.

Google Play Store API. A more automated way to obtain APKs is by using the Google Play Store API. Even though there is no official documentation for the Google Play Store API, there are some third-party tools like GPlayCli [7] that can query the API. The API also has access to older versions that cannot be directly downloaded from the Google Play Store. These types of third-party tools are often deprecated or broken. Unfortunately, it turns out these were somewhat unreliable for our purposes: some versions which we extracted from a phone with ADB were unavailable through the API afterwards. Furthermore, there is a limit on the number of API interactions. Still, the Google Play Store API was used where possible to augment our dataset.

Data Set. For about 8 months starting from the summer of 2020, we tracked 20 Android financial apps from Belgium and the Netherlands. Some of these apps released frequent updates, while others almost had no updates during our observation period. In total, we gathered a 229 different app versions for these 20 apps. As our goal is to evaluate ApkDiff and to draw some more general observations, we do not refer to specific apps or vendors, but rather pseudonymize all references to them.

Note that while we initially gathered the apps, some apps could not be analysed by soot, the analysis framework on which ApkDiff relies. We observed both soot throwing an exception, or the analysis of soot taking too long. We excluded those apps from our analyses.

6 RESULTS

In this section, we present our results. We start off by describing the relevant statistics of the dataset on which we will evaluate ApkDiff. Next, we describe how well ApkDiff works, to match classes in this dataset. Finally, we will show how our algorithm can sometimes be used to deobfuscate classes. Note that all our measurements were performed with ApkDiff's conditional name matching enabled.

6.1 Statistics of the Data Set

ApkDiff works on bytecode rather than native code. Thus it is interesting to not only measure how large the code base for each app is, but also how this is split across bytecode and native code. We use file sizes as a proxy for the size of the code base. These file sizes allow us to deduce quite some information, for example whether there is native code at all. As apps often have native code for multiple ABIs, average over all of the available ABIs rather than report absolute sizes. Table 1 shows the distribution of bytecode and native code for the apps in our dataset.

Table 1: Bytecode (.dex files) and Native code (lib folder) sizes in MB per app (averaged over all available versions and ABIs).

App	Byte	Native	App	Byte	Native
appA	28.1	88.1	appK	19.3	0
appB	29.4	69.1	appL	21.2	0
appC	22.7	0	appM	3.5	9.2
appD	12.9	22.8	appN	13.8	4.7
appE	45.0	0	appO	27.5	0
appF	8.5	6.4	appP	5.5	3.5
appG	14.7	42.8	appQ	4.6	119.7
appH	7.5	26.6	appR	8.8	5.3
appI	10.5	13.5	appS	25.7	3.6
appJ	11.3	4.3	appT	9.0	5.3

Table 2: Percentage of unobfuscated classes, based on applying our heuristic (averaged over all available versions).

App	%	App	%	App	%	App	%
appA	23.67	appF	34.73	appK	24.03	appP	85.09
appB	52.66	appG	57.95	appL	37.12	appQ	95.96
appC	90.35	appH	45.06	appM	97.25	appR	32.86
appD	91.39	appI	65.74	appN	89.08	appS	57.91
appE	67.48	appJ	45.84	appO	16.57	appT	32.78

The distribution of bytecode and native code varies a lot between the applications. On the one hand, apps like appC, appE, appK, and appL have no native code whatsoever. All functionality is implemented in bytecode, including the potentially security-critical functionality. On the other hand, apps like appH, appQ, appG, and appA have more native code than bytecode. This would suggest that most functionality is implemented in the native code, which could range from core functionality, to the whole-sale inclusion of third-party libraries such as OpenCV and OpenSSL. As some of our results also depend on whether or not classes are considered to have been obfuscated, according to our heuristic, Table 2 provides an overview per app of the percentage of unobfuscated classes, based on our heuristic.

Finally, we need to consider how realistic our use case is: are apps updated often enough, so that reverse engineers would be working on a version while a new version is released? If apps release new updates sporadically, matching classes between versions might not be as important. We can get a feeling by look at the update frequencies of the applications in our dataset. We have an average of 0.99 and a median of 0.86 updates per month. In other words, the average app has an update frequency of (slightly above) one month. There are some outliers in both directions, ranging from 0.2 at the lowest end, to 2.5 updates per month (i.e., on average once every 12 days). These numbers suggest that a diffing approach might be useful for reverse engineers.

6.2 Self-Matches

Because unobfuscated versions of the apps in our dataset are not available, it is difficult to evaluate the algorithm for accuracy. However, we can still create an upper bound on the accuracy of ApkDiff by having matching an app version *with itself*, as the ground truth is known in this case: it is the identity mapping, where every class has to match with itself. This already allows us to evaluate different aspects. First, we can consider false positives, i.e., when a class is incorrectly matched to a different class. Furthermore, we can consider the amount of true positives, i.e., how many classes are correctly matched with itself.

To evaluate ApkDiff in this setting, we executed it on 229 different versions across our data set of 20 different banking apps, for a total of 3,466,656 classes. There were no false positives. This is not unsurprising, as we only consider matches when, according to the features ApkDiff uses, there is no possible ambiguity. As for the true positive rate, ApkDiff correctly matched 92.3% of the classes per application.

We do not achieve a true positive rate of 100% as ApkDiff uses its heuristic to detect obfuscated identifiers. In that case, the obfuscated identifier is discarded and the entire matching process then only depends on the class structure itself. We observe that the unmatched classes did not find a unique match because there were too many potential matches with the same class structures. For all of these classes, there were other classes that had the same class structure. An example is empty marker interfaces. Sometimes interfaces are left empty and are used to mark a class to have a certain feature, e.g. Serializable in Java. If there are multiple of these interfaces and their names are removed, they all look exactly the same. There is no way to distinguish between them by looking at their structures alone.

6.3 Matching Different App Versions

Of course, our goal is not to match a version with itself. As our use case is a reverse engineer wanting to port information from one version to another, it is more interesting to match different versions and see how many classes can be matched. We start by matching subsequent pairs of app versions, the results of which are shown in Table 3. On the left are the results for each version that matches with itself and on the right are the results for subsequent pairs of versions. We see that here on average 73.7% of the classes matched per application. Even though in this case we have no ground truth, note that ApkDiff only considers a pair of classes as a match when there is no ambiguity given the used features, which would give a reverse engineer a reasonable degree of confidence in the correctness of these matched classes.

While percentages are easier to compare between different applications, the absolute number of classes that matched gives a more accurate view of the application. The absolute number of classes, the self-matches, and the matches between the last and second to last version are shown in Figure 6. Here we see that the absolute number of matches differs significantly between apps and that a higher percentage does not necessarily mean that more classes were matched. For example, a higher matching percentage is observed in appI but more work can be saved for a reverse engineer in appK because more classes were matched.

Table 3: Percentage of classes that matched with itself and between subsequent versions per app (averaged over all available versions or pairs).

App	Self	Pair	App	Self	Pair
appA	87.5%	77.1%	appK	91.0%	88.2%
appB	93.0%	82.9%	appL	90.0%	87.8%
appC	97.6%	74.6%	appM	100.0%	97.6%
appD	98.0%	98.0%	appN	98.5%	62.6%
appE	94.0%	67.0%	appO	83.3%	58.4%
appF	88.8%	79.9%	appP	99.1%	79.0%
appG	92.0%	79.3%	appQ	99.4%	78.8%
appH	92.0%	91.4%	appR	84.0%	76.6%
appI	96.0%	93.3%	appS	96.0%	51.7%
appJ	89.2%	78.3%	appT	84.0%	76.8%
Arithmetic Mean				92.7%	78.8%
Harmonic Mean				92.3%	73.7%

Table 4: Percentage of unobfuscated and obfuscated classes that matched between subsequent versions per app (averaged over all available pairs).

App	Unobf	Obf	App	Unobf	Obf
appA	90.6%	72.9%	appK	97.3%	85.6%
appB	98.0%	66.0%	appL	97.5%	82.2%
appC	77.0%	77.4%	appM	97.6%	97.5%
appD	99.0%	84.3%	appN	64.5%	62.5%
appE	72.2%	61.6%	appO	93.3%	51.4%
appF	95.4%	71.4%	appP	86.7%	68.5%
appG	85.3%	72.7%	appQ	79.5%	70.8%
appH	98.8%	85.3%	appR	79.8%	75.6%
appI	97.0%	86.5%	appS	71.7%	23.3%
appJ	92.5%	61.0%	appT	80.4%	75.6%
Arithmetic Mean				87.7%	71.6%
Harmonic Mean				83.8%	56.2%

However, when interpreting these results, two aspects are important to consider: the amount of native code, and whether or not code has been obfuscated. As for the amount of native code, Figure 7 shows the relation between the percentage of native code in an app versus the amount of matched classes. This shows a trend where apps with a higher matching percentage generally have more native code compared to bytecode. As for the code obfuscations, we used our heuristic as described in Section 3. We are interested in the performance on obfuscated classes versus the obfuscated classes, the idea being that the obfuscated classes might tend to have more important functionality, and are harder to manually match based on their names. We stress, however, that is impossible to know with certainty whether or not a class has been obfuscated with these methods, but at least this leads to an estimate. Table 4 splits the percentages of matched classes between the classes that are marked as unobfuscated or obfuscated, based on their identifier.

For some apps we can almost match 100% of the classes, while other applications have a matching percentage as low as 23%. In most applications however, we achieve a relatively high matching percentage. On average, around 56.2% of the obfuscated classes have a match, meaning that the majority of the classes can be linked between versions, potentially saving a substantial amount

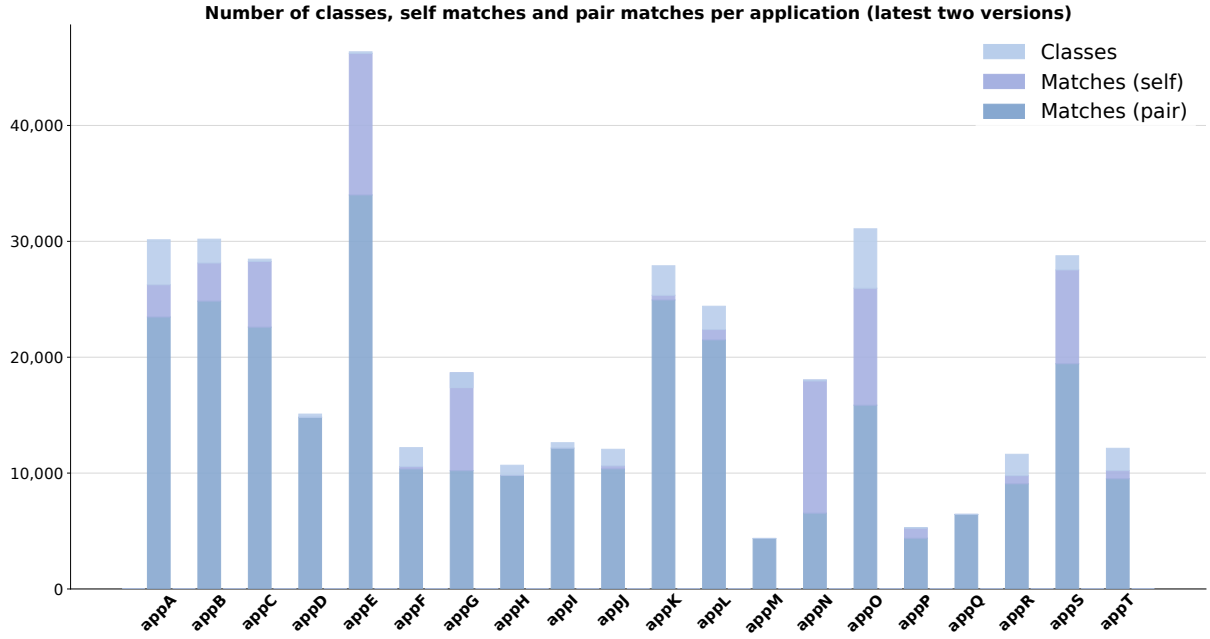

Figure 6: Number of classes, self matches and pair matches per application (latest two versions).

of reverse-engineering work for an attacker. Note that the harmonic mean of 56.2% is significantly lower than the arithmetic mean because of the outliers with low percentages. Most apps achieve a better matching percentage for the obfuscated classes.

Finally, it is possible that different versions of an app are completely different because of the obfuscations which have been applied to them. With ApkDiff, we can do a basic analysis to verify this. For every match that we found, we can check if the class names are identical and thus did not change across versions. The class name is either not changed or it is renamed to the same obfuscated identifier. We calculated the number of matches that had identical names and averaged this over all subsequent pairs of versions. The results are shown in Table 5. Here we see varying results: apps such as appA or appR have a low percentage. Only around 40% of the matches had the same class name. Conversely, we have apps such appD and appH that reach 98% or more. In the latter case, we have to assume that the obfuscations are applied in the same manner for each new version. If this is the case, we can simply match classes based on their class name alone and reach the same results. This is not possible for all applications, as on average only 68% of the matches have the same class names. Here we cannot simply match on the class names as we would lose too many matches.

App	Pairs	Id. Name	App	Pairs	Id. Name
appA	7	38.1%	appK	18	95.8%
appB	10	71.9%	appL	4	91.4%
appC	9	71.0%	appM	41	99.5%
appD	6	99.9%	appN	12	54.7%
appE	5	50.7%	appO	9	71.2%
appF	46	70.0%	appP	6	91.6%
appG	3	73.3%	appQ	4	97.7%
appH	9	98.9%	appR	5	39.2%
appI	4	99.6%	appS	3	83.8%
appJ	4	94.1%	appT	5	40.8%
Arithmetic Mean					76.7%
Harmonic Mean					68.0%

Table 5: Percentage of matches with an identical class name per app (averaged over all pairs).

6.4 ApkDiff for Deobfuscation

ApkDiff is designed to match classes between different versions to aid reverse engineers in porting information between potentially obfuscated versions. Given that we also tracked a history of different app versions, an interesting question is whether or not obfuscations are *introduced* throughout the lifetime of an app, and, if so, whether or not ApkDiff can help a reverse engineer link the 'original' unobfuscated class to later obfuscated versions of that same class. We can answer both questions affirmatively. By analysing the results of ApkDiff on our dataset, we sometimes see an obfuscated class match with an unobfuscated one, which is indicative of the developers suddenly enabling obfuscations for this class. We investigated two different such classes of appC, and manually verified the correctness of our results by comparing the bytecode. We observed both renaming and repackaging of the classes. Interestingly, we observed that the identifier renaming scheme used changed across versions. The initial versions used unobfuscated class names and contained the full package hierarchy details and class name. Suddenly the class was moved into a container package with a short identifier, and the class name itself was changed into a clearly meaningless sequence of Unicode characters. The next version, while still being located in the container package, the class name was changed into a meaningful name with clear

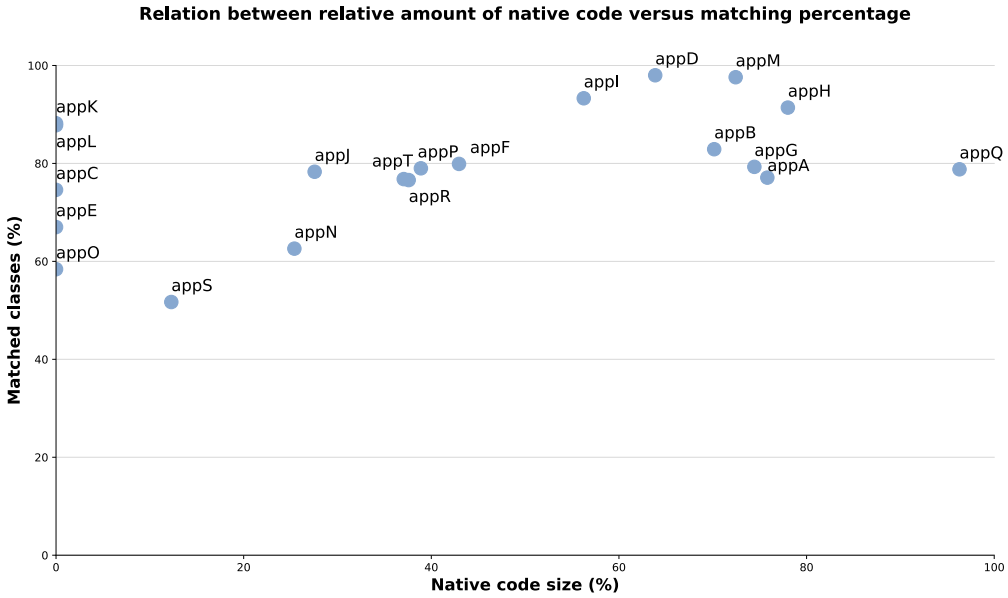

Figure 7: Relation between relative amount of native code versus matching percentage.

human-interpretable semantics which were completely unrelated to the original class name. Across subsequent versions, the name of this class kept being changed into different human-interpretable-but-unrelated names, until it eventually settled on a particular such name that remained unchanged afterwards. This shows that ApkDiff can also allow reverse engineers to link functionality across differently-obfuscated app versions to unobfuscated app versions.

7 RELATED WORKS

In this section, we will discuss the techniques and related works that are relevant for ApkDiff.

7.1 Binary Diffing

The most closely-related topic is that of *binary* diffing, which tries to solve a similar problem, but for native/binary code. In that case, different representations associated with binary code are matched with one another, ranging from whole functions, through basic blocks (BBLs), to individual instructions. The best-known example of such a tool is BinDiff[1]. BinDiff starts by matching functions to one another, then matches BBLs in matched functions, and finally matches instructions in matched BBLs. BinDiff builds on the work of Flake [24] and Dullien et al. [22]. Flake matches programs based by partitioning functions according to a 3-tuple containing the number of BBLs, the number of edges in their control flow graphs, and the number of edges in the call graph originating from that function. If a group contains exactly one function of each program version, these functions are considered a match. The set of matched functions is then used as a base for iteratively improving the matched function set. In subsequent iterations, functions are only considered when they have call edges originating from already matched functions. Dullien et al. extend and generalize the work of Flake in two significant ways. Firstly, they apply the same techniques to BBLs and instructions in these BBLs. Secondly, their algorithm generalizes the partitioning of functions with 3-tuples by considering generic *selectors* and *properties* for matching graphs. A selector maps a single node from graph A to the unique most similar node of graph B, if it exists. The 3-tuples from Flake are an example of such a selector. Properties are functions that, when given a graph, return a subset of that graph's nodes. They can thus be used to reduce the size of the graphs fed to the selectors as input. For example, a property for call graphs could return only the functions of the input call graph that contain a recursive function call. The goal of reducing the graph is to reduce the number of identical 3-tuples for a selector, improving the chances of finding a unique match. Other tools such as Diaphora[2] have implemented similar techniques. ApkDiff's approach is inspired by BinDiff and the work on which BinDiff is based, but focuses on matching *classes*, which allows us to take advantage of the much richer set of metadata associated with them when compared to binary code. Other binary diffing approaches exist, such as techniques based on symbolic execution and theorem proving [28], dynamic similarity testing [23], system call slicing [33], locality-sensitive hashing [31, 34], etc.

7.2 Clone Detection

Android apps can be downloaded, extracted, modified and repackaged because they are accessible. This makes these apps susceptible to malware distribution and clones. One way to find such clones is by fingerprinting the Abstract Syntax Tree (AST) [20]. Another method of finding reused code is used by SUIDroid [30],

[1]https://www.zynamics.com/bindiff.html

[2]https://github.com/joxeankoret/diaphora

who uses UI-based birthmarks (or fingerprints) to detect similarities between Android apps. Soh et al. [38] also opt for a UI-based approach to clone detection. Instead of fingerprinting the AST, it is also possible to make fingerprints at the (approximated) source code level.

The above methods do not actually use the extra metadata that an Android application carries. Android apps follow a known structure. The entry points of the application are defined in the Android manifest file. This is something that Byoungchul et al. address in their clone detection tool RomaDroid [29]. RomaDroid follows the entry points throughout the software and compares the paths using a longest common sequence approach.

7.3 Identifying Third-Party Libraries

A different use case that utilizes some similar techniques is identifying third-party libraries in Android apps. Here we try to identify which libraries an app uses. There has been quite a bit of research in this domain. Most recently Zhang et al. proposed a tool named LibID [40]. LibID starts by constructing the control flow graph and splitting the code into BBLs. These BBLs form a class signature which is used to match classes from a library. The study also introduces the candidate (or potential) match, where two classes could possibly be matched later. The importance is also shown in its results: LibID identified a vulnerable version of the OkHttp library, an HTTP client for Android, in nearly 10% of popular Google Play apps. Another recent tool employing some similar ideas is LibScout [16, 21]. The main difference is the usage of a fixed-depth merkle tree, together with a fuzzy hashing technique to account for uncertainties.

7.4 Deobfuscation

Deobfuscation tries to undo some obfuscations so that attacker can understand the semantics of an application faster. Comparing different application versions could becomes easier as well. However, deobfuscation is no easy task: Many attempts have been made with various techniques, but no single solution has been found. Obfuscation is a broad topic, so most research tries to focus more on specific cases. For example, Baumann et al. [18] try to find known pieces of code (e.g. from a library) inside an APK by comparing method fingerprints in at the intermediate language level. Bichsel et al. [19] use a similar technique. Method fingerprinting might be a useful extension for ApkDiff.

8 CONCLUSION

We designed and implemented ApkDiff, an algorithm to match classes in different Android application versions. For every class in an application, our algorithm tries to find the equivalent class in the new version. It does so based finding unique matches, an then propagating information from these unique matches to extend the set of matched classes.

To evaluate ApkDiff, we popular financial apps from Belgium and the Netherlands for eight months to create a dataset, allowing us to extensively evaluate the attack strategy.

On average, our algorithm could match 92.7% of the classes in the latest version of all apps in the dataset when matching that version with itself. This provides an upper limit for the matching performance per app. Across all successive pairs of all application versions, on average 78.8% of the classes could be matched. If we split the classes into obfuscated and unobfuscated classes, as determined by our heuristic, we find that on average 71.6% of the obfuscated classes and 87.7% of the unobfuscated classes were matched. All of these statistics suggest that current popular financial applications are not resilient to this style of matching.

Besides matching classes between subsequent versions, we also demonstrated other use cases for this algorithm, such as finding the unobfuscated version of a class by matching a much older version where this class had no obfuscations.

ACKNOWLEDGMENTS

This research was partly funded by the Cybersecurity Initiative Flanders (CIF) from the Flemish Government and by the Fund for Scientific Research - Flanders (FWO) [Project No. 3G0E2318].

REFERENCES

[1] Accessed: 2021-01-05. Android's R8 documentation. https://developer.android.com/studio/build/shrink-code.
[2] Accessed 2022-08-26. *Chinese Python.* http://chinesepython.org/.
[3] Accessed 2022-08-26. DashO. https://www.preemptive.com/products/dasho/features.
[4] Accessed 2022-08-26. DexGuard. https://www.guardsquare.com/en/products/dexguard.
[5] Accessed 2022-08-26. *Farsinet, a .NET based farsi programming language.* https://web.archive.org/web/20220709205424/https://code.google.com/archive/p/farsinet/.
[6] Accessed 2022-08-26. Google Java Style Guide. https://google.github.io/styleguide/javaguide.html#s5.1-identifier-names.
[7] Accessed 2022-08-26. GPlayCli GitHub. https://github.com/matlink/gplaycli. Accessed: 2020-12-16.
[8] Accessed: 2022-08-26. GUJTO – Ghent University Java Type Obfuscator. http://gujto.elis.ugent.be/.
[9] Accessed 2022-08-26. *Hindawi Indic Programming System.* https://sourceforge.net/projects/hindawi/.
[10] Accessed 2022-08-26. JFuscator. https://secureteam.net/jfuscator-features. Accessed: 2020-12-16.
[11] Accessed 2022-08-26. KlassMaster. https://www.zelix.com/klassmaster/features.html. Accessed: 2020-12-16.
[12] Accessed 2022-08-26. ProGuard. https://www.guardsquare.com/proguard.
[13] Accessed 2022-08-26. SecureIt. http://www.allatori.com/features.html. Accessed: 2020-12-16.
[14] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. 2020. Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX* 11 (2020), 100403.
[15] Jeff Atwood. 2009. The Ugly American Programmer. https://blog.codinghorror.com/the-ugly-american-programmer/.
[16] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *ACM CCS*. 356–367.
[17] barthen. 2009-03-19. [WoW] [3.0.9] Symbolic info. https://www.ownedcore.com/forums/world-of-warcraft/world-of-warcraft-bots-programs/wow-memory-editing/219320-wow-3-0-9-symbolic-info.html.
[18] Richard Baumann, Mykolai Protsenko, and Tilo Müller. 2017. Anti-ProGuard: Towards Automated Deobfuscation of Android Apps. In *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems (SHCIS '17)*. ACM, 7–12.
[19] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 343–355.
[20] Michel Chilowicz and Gilles Roussel. 2009. Syntax tree fingerprinting for source code similarity detection. 243–247.
[21] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *ACM CCS*.
[22] Thomas Dullien and Rolf Rolles. 2005. Graph-Based Comparison of Executable Objects. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications*. 1–3.
[23] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *USENIX Security*. 303–317.
[24] Halvar Flake. 2004. Structural Comparison of Executable Objects. In *DIMVA*.

[25] Christophe Foket. 2015. *Global obfuscation of bytecode applications.* Ph. D. Dissertation. Ghent University.
[26] Christophe Foket, Bjorn De Sutter, and Koen De Bosschere. 2014. Pushing java type obfuscation to the limit. *IEEE Transactions on Dependable and Secure Computing* 11, 6 (2014), 553–567.
[27] Christophe Foket, Bjorn De Sutter, Bart Coppens, and Koen De Bosschere. 2012. A novel obfuscation: class hierarchy flattening. In *International Symposium on Foundations and Practice of Security*. Springer, 194–210.
[28] Debin Gao, Michael K Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*. Springer, 238–255.
[29] B. Kim, K. Lim, S. Cho, and M. Park. 2019. RomaDroid: A Robust and Efficient Technique for Detecting Android App Clones Using a Tree Structure and Components of Each App's Manifest File. *IEEE Access* 7 (2019), 72182–72196.
[30] F. Lyu, Y. Lin, J. Yang, and J. Zhou. 2016. SUIDroid: An Efficient Hardening-Resilient Approach to Android App Clone Detection. In *2016 IEEE Trustcom/BigDataSE/ISPA*. 511–518.
[31] Elie Mengin. 2019-09-09. Weisfeiler-Lehman Graph Kernel for Binary Function Analysis. https://blog.quarkslab.com/weisfeiler-lehman-graph-kernel-for-binary-function-analysis.html.
[32] Charlie Miller, Dion Blazakis, Dino DaiZovi, Stefan Esser, Vincenzo Iozzo, and Ralf-Philip Weinmann. 2012. *iOS Hacker's Handbook*. John Wiley & Sons.
[33] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *USENIX Security*.
[34] Rylan O'Connell and Ryan Speers. 2019-12-02. Hashashin: Using Binary Hashing to Port Annotations. https://www.riverloopsecurity.com/blog/2019/12/binary-hashing-hashashin/.
[35] Oracle. Accessed: 2021-3-11. Code Conventions for the Java TM Programming Language. https://www.oracle.com/java/technologies/javase/codeconventions-contents.html.
[36] Oxford. Accessed 2021-2-27. *Are There Any Words With The Same Letter Three Times In A Row?* https://www.lexico.com/explore/words-with-same-letter-three-times-in-a-row.
[37] ProGuard. Accessed 2022-08-29. ProGuard manual. https://www.guardsquare.com/manual/configuration/usage.
[38] C. Soh, H. B. Kuan Tan, Y. L. Arnatovich, and L. Wang. 2015. Detecting Clones in Android Applications through Analyzing User Interfaces. In *2015 IEEE 23rd International Conference on Program Comprehension*. 163–173.
[39] Y. Wang and A. Rountev. 2017. Who Changed You? Obfuscator Identification for Android. In *MOBILESoft*. 154–164.
[40] Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. 2019. LibID: Reliable Identification of Obfuscated Third-Party Android Libraries. In *ISSTA*. ACM, New York, NY, USA, 55–65.