Accurate and Efficient Code Matching Across Android Application Versions against Obfuscation

Runhan Feng[†], Zhuohao Zhang[§], Yetong Zhou[†], Ziyang Yan[†] and Yuanyuan Zhang[†*]
[†]Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
[§]School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China
[†]{fengrunhan, andrew_yan, yzy123, yyjess}@sjtu.edu.cn, [§]2020090922028@std.uestc.edu.cn

***Abstract*—In an effort to enhance the attractiveness of apps, developers consistently and frequently release updates to introduce new features and address known issues. Although frequent updates are beneficial for improving user experience, they also increase the workload for reverse engineers since existing analysis results may become obsolete after the release of a new version. Matching code across app versions can help reverse engineers quickly migrate existing analysis results to new versions, verifying whether their prior findings still hold in the new version. This allows them to focus more on the modified portions of the code, thus increasing reverse engineering efficiency. Nevertheless, existing techniques cannot effectively match the code of apps protected by obfuscation techniques, which are pervasively adopted in practice. To address the challenges introduced by code obfuscation, this study presents MatchScope, a novel automated approach designed to match code at the method level across versions of Android app binaries. MatchScope effectively leveraging different levels of fine-grained code features, including class structures and method opcodes, etc., for similarity comparison, thus achieving high accuracy. To further enhance the matching efficiency, we design an index-aware matching algorithm, significantly reducing the scope and number of pairwise comparisons required compared with existing work. The critical insight of our algorithm lies in that the obfuscation tools usually rely on an incrementing index to generate obfuscated names for classes in a deterministic way. Our evaluation on 20 open-source and 60 real-world apps demonstrates the effectiveness of MatchScope. The precision and recall of MatchScope on the ground truth achieve 97.49% and 92.34%, respectively, which are 19.50% and 30.74% higher than the state-of-the-art tool.**

I. Introduction

Recent years have witnessed exponentially rapid development of the Android ecosystem. As of 2022, there are over 3 billion active Android devices worldwide [1], and over 2.6 million apps are hosted on the Google Play Store, the primary marketplace for Android apps [2]. To introduce new competitive functionalities and address issues such as security vulnerabilities and UI bugs in existing code, app vendors consistently and frequently release updates. This practice not only improves the apps' quality but also boosts their popularity [3] [4].

While frequent updates enhance user experience, they also bring additional burdens to app reverse engineers. Due to the closed-source nature of most commercial apps, security professionals rely on reverse engineering for security and privacy-related tasks, including vulnerability discovery and malicious behavior identification [5] [6] [7] [8]. Reverse engineering is a time-consuming process that requires considerable expertise and tedious manual efforts [9] [10]. Previous research indicates that reverse engineers need 39 minutes on average to comprehend small decompiled code snippets with less than 150 lines [11]. Owing to the complexity of reverse engineering, the limited number of qualified reverse engineers is insufficient to cope with the continuous growth in the number of apps and their subsequent versions [12]. To close the gap, there has been considerable effort to simplify the reverse engineering process [13]. A substantial amount of literature is focused on detecting third-party libraries used in apps [14] [15] [16] [17] [18] and inferring identifier names as well as data structures [19] [20] [21] in the decompiled code. While tools proposed in these works can enhance reverse analysis efficiency to some extent, they cannot migrate existing analysis results to new versions of the apps, which is particularly crucial in the context of frequent app updates. This motivates the research of matching code across app versions [22] [23] [24]. Specifically, assuming reverse engineers have already completed the analysis of a specific app version, when a new version is released, if the code from the original and new versions can be matched, reverse engineers can quickly verify whether existing analysis results remain valid in the new version. Otherwise, reverse engineers have to start from scratch to locate the relevant code in the new version, leading to an extra workload. Automatically matching code across app versions allows them to focus more on the modified or newly added code, which is more likely to introduce new security issues [3] [25].

Ideally, the changed code can be easily identified by comparing the binary of app versions. However, considering protecting intellectual property rights and reducing app size, etc., many developers have adopted obfuscation techniques, which is a common practice in the development of Android apps [26] [27]. In particular, Dominik et al. show that roughly 50% of the most popular apps with over 10 million downloads are obfuscated before being released to Google Play Store [28]. Code obfuscation presents significant challenges for this task because the code features between obfuscated and original code can be exceptionally different. For example, considering the identifier renaming strategy used by almost all obfuscation tools, identifiers, including method, class, and package names, are converted into short and meaningless strings, which makes it difficult to recover the matching relationships across versions. Although existing work shows some resilience to obfuscation [22] [23] [24], we have found that they cannot achieve

satisfactory performance in practical analysis. On one hand, existing tools have not taken into account advanced package hierarchy obfuscation techniques (e.g., class repackaging and package hierarchy flattening). Specifically, although existing work considers the impact of obfuscation and designs solutions that are obfuscation-resilient, they often consider identifier obfuscation only and utilize package hierarchy features which are not available when advanced package hierarchy obfuscation techniques are used to aid in matching code. However, based on our analysis of popular apps, we have observed that, besides identifier obfuscation, package hierarchy obfuscation is currently widely adopted in numerous apps, significantly impacting the effectiveness of existing approaches [23]. On the other hand, existing work struggles to strike a balance between accuracy and efficiency. In particular, to achieve high efficiency, code features are used for direct hash queries, resulting in insufficient accuracy due to the low entropy of the features and the bias introduced by obfuscation [29]. To enhance matching accuracy, more granular features are used to perform pairwise similarity comparisons, which is time-consuming. As the app evolves, the amount of code it contains also increases significantly (e.g., tens of thousands of classes and hundreds of thousands of methods). Although some approaches utilize the package hierarchy structure to limit the scope of pairwise similarity comparisons to improve efficiency, as mentioned earlier, the package hierarchy is often obfuscated in many cases (e.g., most classes in an app are packaged into one package). Consequently, in the worst case, this could result in the degradation of existing approaches to performing pairwise similarity comparisons across all classes in an app, affecting both the accuracy and efficiency [22].

In this paper, we propose MatchScope, a comprehensive solution that enables accurate and efficient code matching across app versions. To achieve high accuracy, we extract fine-grained code features, including class structures, method descriptors, and method opcodes, for code similarity comparison. We make a practical assumption about the obfuscation applied to the apps, namely there usually exist non-obfuscated classes in apps. Based on this, we adopt adaptive matching strategies according to whether the class names are obfuscated or not. Regarding improving matching efficiency, MatchScope fully leverages the index information implicitly embedded in the obfuscated class names of the apps to perform index-aware matching so that the pairwise comparison can be guided, significantly reducing the number of comparisons required. The insight of such optimization lies in that the obfuscation tools usually rely on an incrementing index to generate obfuscated names for classes in a deterministic way. Therefore, the obfuscated names of the same class are stable unless developers make changes like adding or removing classes, resulting in alterations to the final indexes of the classes during obfuscation. Given two versions of an app, MatchScope outputs the matched (**identical** or **updated**) methods in both versions, **new** methods in the newer version, and **deleted** methods in the original version. This information can effectively assist reverse engineers in dealing with frequent app updates.

We implement MatchScope and evaluate it on 20 open-source and 60 real-world apps. The latest five versions of the 20 open-source apps constitute the ground truth. The evaluation results on the ground truth show that MatchScope can accurately match classes and methods across versions. The precision and recall for class matching are 97.49% and 92.34%, respectively, which are 19.50% and 30.74% higher than APKDiff, the state-of-the-art class matching tool. Regarding method matching, MatchScope achieves precision and recall of 99.75% and 95.23%, respectively. With respect to efficiency, for popular apps, MatchScope takes an average of 95.87 seconds to analyze consecutive versions, significantly faster than APKDiff, which takes 328.15 seconds on average.

In summary, we make the following major contributions in this work:

- We design a novel technique to match code across Android app versions, which can effectively handle common obfuscation techniques, including identifier renaming and package hierarchy obfuscation. Our approach extracts fine-grained code features, including class structures, method descriptors, and method opcodes, for code similarity comparison and leverages the implied index information from obfuscated class names to improve the matching efficiency further.
- We implement our technique in a tool called MatchScope and evaluate its performance with open-source and real-world apps. Compared with the existing tool, MatchScope can match classes and methods across app versions more accurately and efficiently. The source of MatchScope is publicly available at https://github.com/Aoa0/MatchScope.

The rest of the paper is organized as follows. We first introduce the background and research scope in Section II. We then describe the design of MatchScope in Section III. The evaluation results are presented in Section IV, followed by a discussion in Section V. We introduce related work in Section VI and conclude our paper in Section VII.

II. Background and Scope

This section briefly introduces the obfuscation techniques adopted by Android apps, which is the main challenge in matching code across versions. Also, we define the scope of our research in this work.

A. Android Obfuscation

Code obfuscation is a technique that makes program code less understandable by applying functionality-preserving transformations [30]. It is widely adopted to defend against reverse engineering and protect intellectual property [31]. In the Android ecosystem, source code written in different languages will ultimately be compiled into binaries with different file formats. For instance, Java/Kotlin code is compiled into Android bytecode, while C/C++ code is compiled into native binaries. Generally, different formats of binaries adopt different obfuscation techniques. In this work, we focus on the Java binaries, and the subsequent parts of this section will introduce two

obfuscation techniques (i.e., identifier renaming and package hierarchy obfuscation) used on Java bytecode, which is highly related to our work.

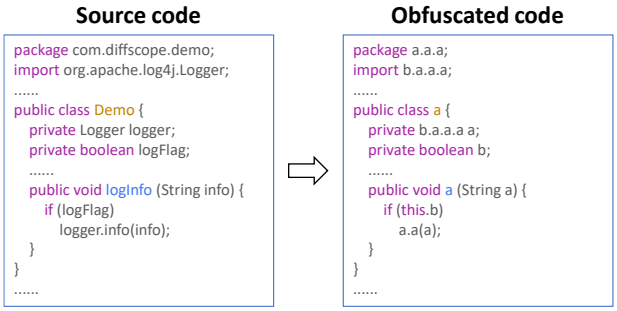


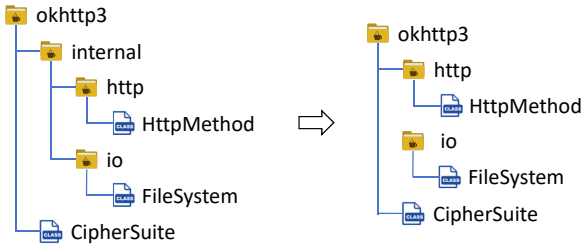
Fig. 1: Code Example of Identifier Renaming

Identifier Renaming. Identifier renaming is the most commonly used obfuscation technique in Android apps. In software development, developers use meaningful names for program elements (e.g., variables, methods, classes, and packages) to improve code readability and maintainability [26]. However, if such meaningful identifiers are compiled into the released app, additional semantic information is also provided for reverse engineering. Therefore, developers often use obfuscation tools like ProGuard [32], R8 (from Google) [33], and Redex (from Facebook) [34] to rename identifiers in the source code during the compilation phase into meaningless names, which typically consist of permutations of letters and numbers, such as `aabb` or `X01` et al [28]. Figure 1 shows an example of identifier renaming obfuscation. As shown in the figure, most meaningful identifiers, including names of fields, methods, classes, etc., have been replaced by short and meaningless names, except for the Java keywords, primitive types, and classes defined in the system libraries.

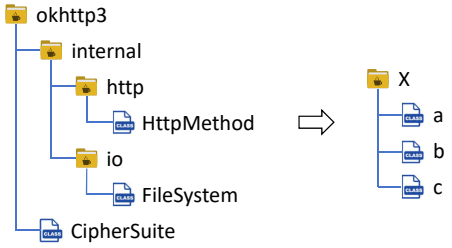
Package Hierarchy Obfuscation. In recent years, more and more apps have adopted package hierarchy obfuscation schemes to provide more profound protection to their code. Packages are usually used to group related files in the development, thereby organizing the source code for improved maintainability. However, such grouping relationships are no longer necessary in the compiled apps. Moreover, preserving this kind of grouping relationship can reduce the effectiveness of other obfuscation schemes because many anti-obfuscation techniques consider the package hierarchy as an essential feature. Therefore, currently, mainstream obfuscation tools like ProGuard and R8 all support obfuscation of the package hierarchy. Taking ProGuard as an example, ProGuard has two obfuscation mechanisms that support modifying the package hierarchy: package flattening and class repackaging. As shown in Figure 2a, with the `-flattenpackagehierarchy` option, ProGuard can move the specified package into the given parent package. Figure 2b gives an example of class repackaging. With the `-repackageclasses` option, ProGuard will move all classes from the target package to the given new package. These two obfuscation mechanisms do not have essential differences as they both lead to altering the original package hierarchy.

B. Scope of Our Work

Obfuscation and deobfuscation techniques have been in an ongoing battle, giving rise to the emergence of new techniques. Therefore, when designing obfuscation-resilient code-matching schemes, supporting all potential obfuscation techniques is impractical. Therefore, before elaborating on the technical details of MatchScope, we define the scope of obfuscation techniques that MatchScope can support. In this work, we focus on identifier renaming and package hierarchy obfuscation on Java code of Android apps, which are the most commonly used obfuscation techniques in practice and are supported by ProGuard and R8, the most popular obfuscation tools [35]. Current obfuscation tools also support more obfuscation techniques, such as string encryption and control-flow randomization. However, due to program performance, robustness, and cost considerations (e.g., control-flow randomization is only supported by commercial obfuscation tools), they are seldom used in Java code, especially in popular apps [26]. These less commonly used techniques are not within the scope of this work, and we will discuss potential mitigation against some of them in Section V.



(a) Example of Package Flattening



(b) Example of Class Repackaging

Fig. 2: Examples of Package Hierarchy Obfuscation

III. Design

Matching code across app versions is by no means trivial due to the obfuscation adopted during the app compilation phase. In this section, we present the design of MatchScope, which effectively tackles the challenge of obfuscation and matches code across app versions with high accuracy and efficiency. The design principle of MatchScope is to minimize

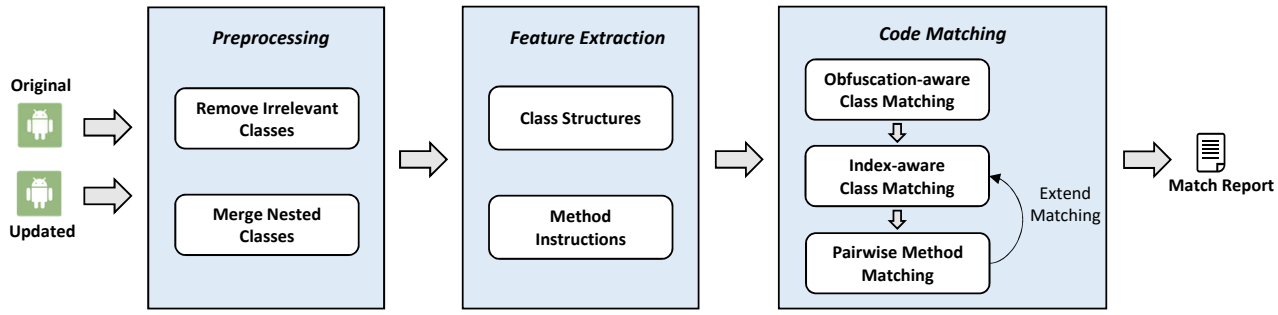


Fig. 3: The workflow of MatchScope

the scope and number of pairwise similarity comparisons as much as possible.

A. Overview

We model code matching across app versions as a search problem. Given two versions of an app, MatchScope searches the newer version for the corresponding code in the older version to establish matching relationships between program elements (i.e., classes and methods) in both versions. In this work, we target method-level matching. Specifically, as for all the methods in two versions, MatchScope eventually outputs the matched methods (which can be **identical** or **updated**) in both versions. Additionally, it identifies **new** methods exclusive to the newer version and the **deleted** methods exclusive to the original version. In contrast with the class-level matching in previous work [22], matching at the method level provides a more granular perspective, thereby enhancing the efficiency of reverse engineers in practical tasks such as bug-fixing analysis.

The workflow of MatchScope is illustrated in Figure 3. It consists of three major phases. Both input apps undergo the same preprocessing and feature extraction procedures. In the preprocessing phases, we adopt two strategies (i.e., removing irrelevant classes and merging nested classes) to reduce the code search space, which will be detailed in Section III-B. In the feature extraction phase, we extract features required for code matching from both class and method levels (Section III-C). Although our ultimate goal is to match methods across versions, we begin with class matching since classes have higher entropy and are relatively more straightforward to match. It is worth mentioning that during the class matching phase, we establish correspondences between classes but do not just find identical classes across versions. In other words, the designed class-matching approach must tolerate a certain degree of code modification. During the class matching phase, MatchScope effectively leverages the information retained in the binaries and adaptively adopts different strategies based on whether the class names have been obfuscated. Particularly, as for the classes with non-obfuscated names, the matching relationships can be established directly based on the class name (Section III-D). As for the remaining classes, MatchScope further inspects the obfuscation-resilient code features to perform matching. A somewhat naive approach would be directly performing pairwise comparisons between all remaining classes from two versions using the extracted code features, but it incurs significant performance overhead. Therefore, we propose index-aware matching, which guides MatchScope in conducting more targeted code comparisons, achieving class matching accurately and efficiently (Section III-E). After completing the class matching, MatchScope further matches the methods within the corresponding classes and eventually outputs the match report (Section III-F).

B. Preprocessing

In the preprocessing phase, two strategies are used to reduce the search space to make subsequent matching more accurate and efficient. On one hand, we remove program elements unrelated to the code-matching task in apps. Three types of program elements are considered irrelevant.

(i) Classes that do not contain actual code, such as interface classes and classes that contain only empty methods.
(ii) Classes and methods with the modifier `synthetic`, which are generated by the compiler.
(iii) Classes in the system libraries (e.g., Android Support Library) since they mainly provide language features or compatibility between different Android versions.

On the other hand, we merge nested classes, e.g., anonymous and inner classes, with the corresponding outer classes, a strategy adopted in previous work [23]. The benefits of the merger operation primarily come from two aspects. Firstly, it aligns well with MatchScope's design principles, further reducing the search space. Secondly, compared to outer classes, nested classes usually contain less information. The matching accuracy could be affected if they are treated as separate units for subsequent class matching. In contrast, merging them with the outer classes can increase the entropy of the outer classes, which further improves the probability of successful matching.

C. Feature Extraction

The key to establishing code-matching relationships across app versions lies in making full use of the features preserved within the binaries for comparison. As for feature extraction, we construct class and method signatures from the bottom up.

Method-level Features: For each method, we extract features from the method descriptor (to construct class-level features) and the method body (to perform method matching), respectively. With respect to the method descriptor, to counter the impact of obfuscation, we normalize all non-system library classes within the method descriptor, replacing them with the capital letter "X." Then, we sort the normalized parameter types and finally get the method descriptor signature similar to existing approaches [16] [36]. Concerning the method body, we traverse all the instructions in the method body, extracting the opcodes, the system APIs called, and constant information to construct the method body signature.

Class-level Features: For each class, we first extract its class structure information, including the superclass it inherits from and the interface classes it implements. Secondly, we extract the type information of all fields in the class. We perform the same normalization operation on the extracted class information, replacing non-system classes with the capital letter "X." Last but not least, we collect method descriptor signatures of all methods defined in the class. To ensure determinism, we sort the interface classes, field types, and method descriptor signatures separately and then concatenate all the features to form the final class structure signature.

In addition to the obfuscation-resilient features mentioned above, for each class, we also record the class and package names to determine whether the class name has been obfuscated, which is necessary to implement adaptive matching strategies. To sum up, we obtain class names, package names, class structure signatures, and method body signatures for subsequent matching. For each class structure signature, we calculate its hash (C_h) and fuzzy hash (C_f). The former is used for quick matching, while the latter is used for fine-grained similarity comparison. For each method body signature, we also calculated its hash (M_h) and fuzzy hash (M_f) for method-level matching.

D. Obfuscation-aware Class Matching

Inspired by existing work [22] and our investigation on real-world apps, we make a practical assumption about the obfuscation used in apps: there usually exist classes whose names are not obfuscated. There are two possible scenarios in this situation. One is that some apps themselves do not use obfuscation techniques. However, due to the inclusion of third-party libraries, obfuscated classes in the apps could still exist. The more common scenario is that the apps do utilize obfuscation techniques. But typically, obfuscation tools do not obfuscate all the class names in the apps to avoid affecting the normal functionality of the app after obfuscation. For example, the name of the classes corresponding to the parameters of the method `Class.forName()` and the classes referenced in the `AndroidManifest.xml` file are usually not obfuscated. Therefore, in real-world apps, it is common to have a portion of the classes whose name is not obfuscated. The matching relationship across versions can be directly constructed without further inspection if the unobfuscated class name exists in both versions, reducing the search space for obfuscated classes. However, this requires our tool to be aware of whether the class names are obfuscated or not.

Therefore, in this work, we conduct an obfuscation analysis on the class name first. Besides class names, we also consider whether the names of the packages where a class is located are obfuscated. In other words, only when both the class name and each level of package name are not obfuscated can this class be directly used for building matching relationships. Nevertheless, deciding whether a symbol name is obfuscated is a non-trivial task [37]. We have formulated some heuristic rules and combined them with a wordlist to analyze whether the identifiers (i.e., package names and class names) have been obfuscated. Specifically, given an identifier, we determine whether it is obfuscated based on the following rules.

(i) If the length of the identifier is less than or equal to 2, and it is not present in the allowlist, which contains common short identifiers such as `os`, `io`, and `ui`, etc., we consider it to be obfuscated.
(ii) If the length of the identifier is greater than or equal to 3 and less than or equal to 5, and it is not present in the wordlist, we consider it to be obfuscated.
(iii) If the length of the identifier is greater than 5, we consider it to be not obfuscated.

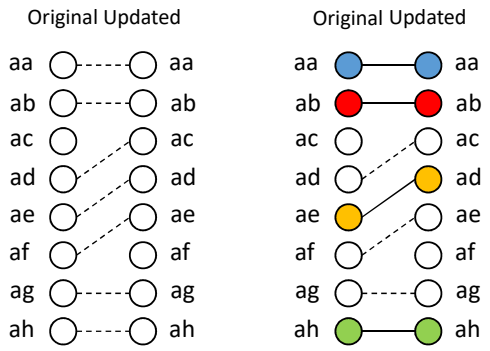
Regarding the wordlist collection, we crawl all libraries from the Maven Central Repository, which contains over 300 thousand Java third-party libraries. We then extract identifiers from their groupId and artifactId to form the wordlist. In addition, we add common words used in the computer science field to the wordlist. The wordlist contains most identifiers that may be used in Android projects since the developers follow similar naming conventions. In Rule (ii), we directly compare the identifiers in the apps with those in the wordlist without tokenizing them based on CamelCase or snake_case naming conventions, as unobfuscated identifiers requiring tokenization are typically longer and can be effectively filtered out by Rule (iii). Our obfuscation analysis strategy has achieved excellent results in practical analysis, successfully identifying almost all classes with unobfuscated names. In the cases that unobfuscated names are mistakenly identified as obfuscated, the overall result will not be affected since such classes will be further inspected for matching in subsequent analysis.

E. Index-aware Class Matching

As for the remaining classes with obfuscated names, we utilize their code features to achieve matching and leverage the index information implicitly encoded in the obfuscated class names to enhance matching efficiency. In this section, we first introduce our observations of indexes used by obfuscation tools to generate obfuscated names for classes through a motivating example. Then, we describe in detail how we use this characteristic to design an efficient class matching approach.

Motivating Example: Assuming we have a mini app's original and updated version, both containing eight classes and obfuscated already. Figure 4a shows the class-level matching relationship between these two versions. Each node represents

a class, and their obfuscated class names, such as `aa`, `ah`, etc., are marked next to them. In this example, the two nodes connected by dashed lines correspond to the matched class in two versions. In the updated version, the class renamed to `ac` during obfuscation in the original version is deleted, and a new class that is renamed to `af` during obfuscation is introduced in the updated version. These two classes do not have matching classes in the corresponding versions. From the figure, we can see that the obfuscation is performed in a deterministic way. Specifically, the classes corresponding to `aa` and `ab` in the original version are still obfuscated as `aa` and `ab` in the updated version. There are mainly two reasons for this. On one hand, obfuscation tools use an incremental index to generate class names deterministically. On the other hand, obfuscation tools process the classes in a fixed order. We analyzed the implementations of mainstream obfuscation tools, namely ProGuard and R8, and confirmed that they exhibit similar designs. This design aims to ensure the determinism and reproducibility of obfuscation results. If obfuscation tools introduce bugs, developers can quickly identify issues and change the scope of obfuscation accordingly. Therefore, ideally, supposing that the class structure of an app remains unchanged while only method-level changes are introduced in updates, the class-matching relationship can be easily constructed since the obfuscated class names remain unchanged across versions. However, in real-world apps, class structures are commonly changed because of adding new classes or deleting existing classes, etc. Concerning the scenario where class structure changes occur, as shown in Figure 4a, the removal of the class corresponding to `ac` does not have a significant impact on the overall order of classes processed by the obfuscation tool. The only effect is that the classes following will have their indexes reduced by one, and this change is reflected in the obfuscated name in the new version (e.g., `ad` changed to `ac` and `ae` changed to `ad`). The opposite situation occurs when new classes are added.



(a) Class Matching Relationships (b) Anchor Matches

Fig. 4: Motivating Examples for Class-level Matching

Our Approach for Matching Classes in a Nutshell: Although this is an extremely simplified example, many popular apps exhibit the same characteristic, probably because of the adoption of mainstream obfuscation tools. Effectively leveraging this characteristic to guide code matching is non-trivial. We design the index-aware class matching approach based on the assumption that most of the code (before obfuscation) remains unchanged before and after the app update. Therefore, we can more easily uniquely match a portion of classes with higher entropy. We refer to these matches as anchor matches since they act as anchors to the indexes between original and updated versions. As is shown in Figure 4b, the nodes that are colored represent classes that are matched as anchor classes. Based on these anchor classes, all classes are divided into smaller slices. The slice between two consecutive anchor matches is referred to as a matched slice. For example, in Figure 4b, `[ac,ad]` and `[ac]` are matched slices, so are `[af,ag]` and `[ae,af,ag]`. The classes in the matched slices are more likely to be successfully matched. Therefore, we perform pairwise similarity comparisons within the matched slices to find more matching relationships efficiently. In the following section, we provide a more detailed explanation of each step in the approach.

Identifying anchor matches: Anchor matches refer to classes that remain unchanged between app versions and are relatively easy to be uniquely matched due to their higher entropy. They serve as anchors during the subsequent matching process. Therefore, the correctness of anchor matches needs to be ensured. To achieve this, we adopt a rigorous approach to determine anchor matches. Specifically, for two app versions, we first obtain the C_h of all classes in each version. Then, we remove classes with duplicate hash values from respective lists, thus obtaining the unique classes in both app versions. After that, we compare C_h between the two versions one by one and finally choose classes with the same hash as anchor matches. In this step, we employ hashing for queries, guaranteeing a quicker identification of anchor matches.

Anchor-guided Matching: To perform subsequent matching, we sort the anchor matches and get the matched slices between consecutive anchor matches, which enables us to perform pairwise comparisons within smaller scopes, effectively enhancing both matching accuracy and efficiency. In some situations, we may encounter crossed anchor matches. For example, if `ab` matches `ah` while `ae` matches `ab` in Figure 4b, then the crossing happens. This could be due to classes renamed by developers or false positives when identifying the anchor matches. To address this issue, we remove the anchor matches that result in crossing. In most apps, a part of the package structures still exists despite the use of package hierarchy obfuscation techniques. If anchor matches appear within these packages, then matching relationships can be established between these packages. In this case, we divide the classes within corresponding packages into matched slices. In processing the matched slices, we adopt a pairwise similarity comparison approach. For classes in the slice of the original version, we search for the most similar class in the corresponding slice in the updated version to establish the class-level matching relationship. In the comparison, we use C_f to measure the class similarity. We only consider

classes to be matched if the similarity of their class exceeds a pre-defined threshold. We denote the threshold for class similarity as T_{class}. After completing pairwise comparisons within the matched slice, the unmatched classes are recorded for further global pairwise comparison. In the global pairwise comparison, if the similarity of C_f between two classes exceeds T_{class}, we add them to the matching relationships. If there are still classes that have not been matched in the global pairwise comparison, we consider them newly added or deleted. After completing all the outer class matches, we further compare the nested classes. Specifically, we compare C_f of the nested classes and match those with a similarity greater than T_{class}.

F. Pairwise Method Matching

In the method-matching phase, we use the method body signatures to identify if methods have been modified. Specifically, for the methods in the matched classes, we perform pairwise comparisons to match methods in classes and further determine whether the matched methods are identical or have been updated. If M_h of two methods are the same, we take them as identical. Otherwise, we further compare the M_f of two methods and take them as updated if the similarity exceeds the pre-defined threshold, which we denote as T_{method}. Ultimately, for the matched classes, we can output whether the methods they contain are identical, updated, newly added, or deleted.

Extend Matching: As for the identical methods, we further inspect the external dependencies to extend class matches. Specifically, if two methods are identical, then the external classes that these two methods depend on should be the same, even if they might be obfuscated with different names. The similarity of the corresponding classes is further inspected to check whether they are actually matched. Method-level matching and class-level matching are performed recursively until no new class matches are found.

IV. Evaluation

We evaluate MatchScope on different settings and address the following three research questions:

- RQ1: How effective is MatchScope for matching classes and methods on the ground truth?
- RQ2: How effective is MatchScope on real-world apps?
- RQ3: What is the runtime performance of MatchScope?

A. Evaluation Setup

Implementation: We build MatchScope based on Soot [38], using Soot to extract code features for matching. MatchScope comprises 2K+ lines of Java code. The evaluation environment is an 80-core server with 256GB RAM, running on Ubuntu OS 20.04 LTS with a Linux 5.4.0 kernel. Two thresholds used by MatchScope, i.e., T_{class} and T_{method}, are set as 0.6 and 0.2, respectively, for the balance of false positives and false negatives.

Dataset: For the evaluation of MatchScope, we construct three datasets. First, we create a ground truth dataset DS_1, which consists of 20 open-source projects from F-Droid [39]. We randomly select these 20 projects from the default app recommendation page of F-Droid [40], covering most categories. We collect each project's latest five released versions from its source code repository for cross-version matching. It is important to note that not all apps in this dataset enable obfuscation by default. For those apps that do not use obfuscation (9/20), we enabled the obfuscation with R8 (the default compiler since Android Studio 3.4) in the compilation to ensure that all apps in the dataset utilize obfuscation mechanisms. We adopted the default obfuscation configuration, which is used by most apps [35]. In addition to the open-source projects, we collect 30 most popular apps [41] on the Google Play Store to investigate the performance of our tool on real-world apps. In order to cover apps in their early lifecycle stages (considering that popular apps have often undergone continuous iterations for a long time), we also randomly gather 30 newly uploaded apps to the Google Play Store in 2023. These newly uploaded apps have received updates within the last two months, indicating they are still actively maintained. With respect to the 60 apps, we collect their latest 10 versions and create the popular app dataset DS_2 and the new app dataset DS_3, respectively. For new apps with a total number of updates below 10, we collect all their updated versions and included them in the dataset. Each app's historical versions are downloaded through APKPure [42]. Table I illustrates the detailed information of the collected apps. In total, we collect 200 versions for the DS_1, 300 versions for the DS_2, and 214 versions for the DS_3. Popular apps undoubtedly have significantly higher download counts. We also collect the release dates of the versions in the dataset to analyze the update cycles of the apps. The table shows that popular apps tend to have shorter update cycles, with an average of 15.30 days. In contrast, apps in DS_1 have the longest update release cycle, averaging 46.33 days. The rapid update frequency of real-world apps in both DS_1 and DS_2 signifies the imperative need for automated app cross-version code matching.

TABLE I: Statistics on the Download Counts and Update Cycle of Apps in the Datasets

Dataset	Average Downloads	Update Cycle (Days)
DS_1	353.51 K	46.33
DS_2	3.51 B	15.30
DS_3	16.44 M	19.26

Table II provides statistics on the number of classes and methods of apps in these three datasets. Since MatchScope performs preprocessing operations on these apps, removing irrelevant classes from matching, we separately collected statistical information before and after preprocessing. We can observe that popular apps generally contain more classes and methods. Before preprocessing, on average, the class quantity in the DS_2 is 16,707, nearly 13 times that of the DS_1 and

TABLE II: Statistics on the Number of Classes and Methods of Apps in the Datasets

Dataset	Before Preprocessing		After Preprocessing	
	# Classes	# Methods	# Classes	# Methods
DS_1	1,297	12,202	1,251	11,409
DS_2	16,707	125,202	16,522	115,371
DS_3	5,313	54,272	5,181	50,407

3 times that of the DS_3. Similar differences in quantity are also reflected in the number of methods. Therefore, achieving good performance on these popular apps requires designing an efficient matching approach. Besides, the proportion of classes obfuscated after preprocessing in the DS_2 is 84.02%. This is a relatively high ratio and also imposes high requirements on the anti-obfuscation capabilities of our designed approach.

B. RQ1: Effectiveness on the Ground Truth

For the 200 apps in DS_1, we manually compiled them to obtain the mapping file generated by the obfuscation tool. This mapping file contains the relationship between the original class names in the source code and the meaningless obfuscated identifiers generated by the obfuscation tool. Using this mapping, we can construct mappings of obfuscated class and method names across different versions, which serve as our ground truth. We choose APKDiff [22], the state-of-the-art class-level matching tool for comparison. Since the source code of APKDiff is currently not publicly available, we followed the description provided in the research paper to reproduce its implementation. APKDiff mainly extracts app class structure features (similar to the class-level features we have extracted) for cross-version class matching and does not support method-level matching. Besides, since APKDiff does not adopt similar preprocessing operations as MatchScope to remove synthetic classes and empty classes, the overall matching scope of APKDiff is slightly larger than MatchScope. We use MatchScope and APKDiff to match consecutive versions of each app in DS_1, and analyze the output of these two tools with the cross-version obfuscated identifier mapping extracted from the mapping file (i.e., the ground truth). Specifically, both tools output the class-to-class mapping relationship from the original to the updated app. For each class in the original app, if the tool identified the correct corresponding class in the updated version, we take it as a true positive. On the contrary, if the tool incorrectly matches it to another class, we take it as a false positive. For those unmatched classes, if they exist in both versions, we take them as false negatives.

As is shown in Table III. MatchScope achieves a precision of 97.49% and a recall of 92.34%, 19.50% and 30.74% higher, respectively, than APKDiff. In terms of false positives of MatchScope, since we use a similarity-based approach to determine whether the classes in the matched slices are matched, if the similarity between different classes is relatively high (e.g., two classes are different implementations of the same interface and share similar functionalities), it may unavoidably lead to incorrect matching. The false negatives of MatchScope mainly stem from two reasons. Firstly, some classes in these apps undergo significant changes (e.g., apps update the third-party libraries used), resulting in the similarity falling below our set threshold. Secondly, in some cases, the apps in DS_1 update their compilation configurations, leading to substantial differences between consecutive versions. However, once the compilation configurations stabilize, the matching performance will return to the normal level. Actually, for real-world apps, considering compatibility issues, the compilation configurations rarely change. Therefore, the performance of our tool in practical use is not affected.

We further evaluate the effectiveness of the method-level matching of MatchScope. The overall evaluation approach is similar to the class-level evaluation. For the methods in the matched classes of the original app, we examine whether they are correctly matched to the methods in the updated version. The methods in the classes taken as false negatives in the class-level matching are also taken as false negatives in the method-level evaluation. As illustrated in Table III, in the method matching task, with the aid of accurate class-level matching, the precision of method matching is exceptionally high, reaching 99.75%. At the same time, due to the effect of the class-level matching recall, the recall at the method level is relatively low but still reaches 95.23%.

TABLE III: Comparison with APKDiff on the Ground Truth (MS=MatchScope, AD=APKDiff, PR=Precision, RC=Recall)

Tool	Class-level			Method-level		
	PR	RC	F1	PR	RC	F1
MS	97.49%	92.34%	94.85%	99.75%	95.23%	97.44%
AD	81.58%	70.63%	75.71%	/	/	/

C. RQ2: Effectiveness on Real-world Apps

To further validate the effectiveness of MatchScope, we evaluate MatchScope on real-world apps, i.e., the apps in DS_2 and DS_3. We use MatchScope to analyze consecutive versions of these apps to perform the matching. We set the timeout for analysis on a pair of consecutive versions to 20 minutes. For all the 540 (60×9) comparisons, MatchScope failed in 6 comparisons due to errors from Soot. Table IV shows the detailed results of these two datasets. As we can see, MatchScope successfully matched on average 93.26% and 93.28% of all classes across app versions in DS_2 and DS_3, respectively. With regard to the method-level matching, MatchScope successfully matched 94.85% and 93.83% methods in the two datasets. Besides, we find that, on average, 47.67% classes in DS_2 will have different obfuscated names after each update.

Since almost all are closed-source, we could not obtain the ground truth of detailed updated information on class and method levels for these real-world apps. Therefore, we manually confirm the accuracy of the matching relationships constructed by MatchScope. Specifically, for each of these 60 apps, we randomly selected one successful matching comparison and extracted at most 50 class mapping relationships

TABLE IV: Matching Results on Real-world Apps

Dataset	# Matched Classes	% Matched Classes	# Added Classes	% Added Classes	# Matched Methods	% Matched Methods	# Added Methods	% Added Methods
DS_2	15,408	93.26%	908	5.50%	109,429	94.85%	3,306	2.87%
DS_3	4,833	93.28%	601	11.74%	47,299	93.83%	2,297	4.56%

where the class names were different in the two versions. Finally, we obtained 2,115 matches. For these matches, we used disassembly tools to observe whether their bytecode matched to determine the correctness of the tool's output. We established an inspection team comprising four researchers with substantial experience in reverse engineering Android apps. The team is divided into two groups, where each pair of members within a group validates the same set of matching relationships and performs cross-validation to avoid errors. For matches with conflicting opinions, four team members discuss together and finally determine the correctness. Eventually, we confirmed that on the real-world apps, MatchScope achieved a detection accuracy of 99.01% (2,094/2,115), which is consistent with our evaluation on the ground truth. In terms of code changes, we find that in each update, the apps in DS_2 on average add more new code than apps in DS_3. This implies that although these popular apps have evolved for several years, they are still undergoing active updates.

TABLE V: Average Execution Time (seconds) on Three Datasets (MS=MatchScope, AD=APKDiff, CM=Class Matching, MM=Method Matching)

Phase	DS_1		DS_2		DS_3	
	MS	AD	MS	AD	MS	AD
Profile	16.36	11.59	68.10	52.42	26.12	17.03
CM	0.52	1.33	26.68	275.73	2.68	9.09
MM	0.11	/	1.09	/	0.44	/
Sum	16.99	12.92	95.87	328.15	29.24	26.12

D. RQ3: Efficiency

We measure the running time of MatchScope and APKDiff on the three datasets. We divide the operation of MatchScope into different phases and record the time separately. Specifically, the running of MatchScope is divided into three phases: profiling, class-level matching, and method-level matching. The profiling phase revolves running Soot packs, extracting the required signatures, and calculating the corresponding hashes. Similarly, the running time of APKDiff is also divided into two parts: profiling and class-level matching. Table V shows the execution time of both tools on the three datasets. MatchScope is overall efficient, and even on DS_2, the dataset with the highest complexity, it can complete the matching process for two apps in an average of 95.87 seconds. In contrast, APKDiff takes three times longer than MatchScope. MatchScope achieves better efficiency because it can conduct more targeted pairwise comparisons compared to APKDiff. This also allows it to use more granular features for similarity comparison, thereby achieving higher accuracy. MatchScope is slightly slower than APKDiff on DS_1. The underlying rationale is that MatchScope needs to collect more features and calculate fuzzy hash compared with APKDiff. However, given that apps in DS_1 are typically smaller than the average, the time saved in code matching by MatchScope is insignificant compared with the time spent on profile building. Indeed, a significant portion of the time spent by MatchScope is dedicated to executing Soot packs and extracting features. There is room for further efficiency improvement by replacing Soot components, which currently MatchScope depends on for feature extraction.

V. Discussion

In this section, we first discuss the robustness and limitations of our work. Then, we discuss the threats to the validity of our work.

A. Robustness and Limitations

To improve the matching efficiency, MatchScope leverages the index information implicitly embedded in the obfuscated class names of the apps to perform index-aware matching so that the pairwise similarity comparisons can be guided. With the evolution of obfuscation techniques, this characteristic (i.e., the implicitly embedded index information) might undergo further obfuscation. In such cases, the efficiency of our tool may be affected, but the overall accuracy will not be affected. However, we believe this characteristic will continue to exist because determinism and reproducibility are among the design principles of obfuscation tools. Introducing additional randomness to the obfuscation stage will increase the workload for developers when analyzing obfuscation-related bugs. Moreover, we argue that it's impractical to design obfuscation-resilient code-matching schemes that withstand all potential obfuscation mechanisms.

As for the limitation, MatchScope may not handle the optimization techniques such as method inlining and wrapping, which needs a better understanding of program semantics by incorporating detection for these optimization techniques. It is still an open question that requires further in-depth research. For the obfuscation technique of string encryption, MatchScope currently does not support it because we extract constant strings as features. We could opt to forgo the constant string feature to support this obfuscation technique. However, we have chosen to use this feature since, in popular real-world apps, Java code rarely employs string encryption as an obfuscation mechanism [26]. Lastly, MatchScope currently supports matching Java code only and does not consider changes in native code. In fact, for some apps, certain functionalities are implemented using native code. The analysis of native code

requires quite different tools and designs, and we leave this for future work.

B. Threats to Validity

In this section, we discuss two potential threats to the validity of MatchScope and provide our solutions to alleviate these threats.

One possible threat to validity is the representativeness of the chosen apps for evaluation. It is essential to ensure that the chosen apps are a diverse and comprehensive representation of the entire Android app ecosystem. To mitigate this threat, we constructed three datasets consisting of open-source apps, popular apps, and new apps, respectively. Among them, popular apps have the broadest impact and can also reflect the characteristics of current mainstream apps. Introducing new apps reflects the early evolution of apps, as popular apps have usually undergone several years of evolution. We also paid attention to the uniformity of categories to which the apps belong, ensuring that biases are minimized.

Another threat to validity is the possible bias and subjectiveness caused by human involvement in the evaluation process. As for the evaluation on real-world apps, we randomly selected matching results and manually validated them. To address the threat caused by human involvement, the authors were divided into two groups in the manual inspection phase. Each pair of members within a group inspected the same set of matching relationships and cross-validated the results. Conflicting opinions were resolved through discussions, ensuring a more objective and comprehensive assessment of the tool's performance.

VI. Related Work

This section briefly reviews prior research on app evolution and code deobfuscation.

A. App Evolution

Since the release of Android, the evolution of Android apps has been a subject of interest among researchers, especially in the context of app security and privacy [43] [44]. Ren et al. studied the privacy evolution of 512 popular apps over eight years [45]. By intercepting and analyzing the network traffic of these apps, they discovered an increasing trend in collecting personally identifiable information within these Android apps. Taylor et al. focus on how permission usage and vulnerability evolve across app versions over two years. They found that in many cases, app updates introduced new security vulnerabilities, leading to more intrinsic vulnerabilities in the new versions of the apps [25]. Similarly, Gao et al. also leverage existing vulnerability-finding tools to investigate how vulnerabilities evolve in terms of vulnerability types and how they were introduced [3]. In addition to the focus on security and privacy aspects, there is a considerable amount of work analyzing the evolution of apps from other various perspectives [46] [47] [48] [49]. For example, McIlroy et al. confirmed that users tend to give higher ratings for apps with higher update frequencies rather than disliking them [4]. Our work can serve as the basis for subsequent studies, enabling research into more granular aspects of app evolution.

B. Code Deobfuscation

Code deobfuscation is a technique to restore the program code safeguarded with obfuscation to the state before obfuscation as much as possible [50] [51] by identifying, simplifying, and removing the obfuscated code, thereby making the program more comprehensible. This is a highly relevant area because, in this work, we focus on circumventing code obfuscation protection to match code across versions. Machine learning is a commonly applied technique for deobfuscation with the help of a large amount of unobfuscated code. DeGuard [19] obtains knowledge of naming methods in Android apps from thousands of unobfuscated app codes and applies the probabilistic model to restore unseen obfuscated apps. Baumann et al. [21] develop a tool for deobfuscating code protected by ProGuard [32], where software similarity algorithms, including SimHash and n-gram, are applied to match similar code segments between the obfuscated and unobfuscated code. Using a deobfuscation tool first to deobfuscate the app code and then perform code matching is also an option for the task in this paper. However, this approach is constrained by the accuracy and efficiency of the deobfuscation tools and can not achieve satisfactory code matching currently.

VII. Conclusion

In this paper, to address the challenge of code obfuscation in cross-version code matching for Android apps, we propose MatchScope, a new technique that leverages features remaining stable before and after obfuscation to perform class and method matching. MatchScope makes a practical assumption about code obfuscation in apps, adaptively matching classes based on whether the names are obfuscated. As for the obfuscated classes, MatchScope effectively leverages different levels of fine-grained code features, including class structures, method descriptors, and method opcodes for similarity matching, thereby achieving high accuracy. Regarding enhancing matching efficiency, the key insight of MatchScope lies in that the obfuscation tools usually rely on an incrementing index to generate obfuscated names for classes deterministically. Based on this finding, we design and implement the index-aware class matching algorithm, which narrows pairwise comparisons to a small scope, achieving accurate and efficient matching. The evaluation on three datasets consisting of open-sourced and real-world apps demonstrates the effectiveness of MatchScope. Our work can not only improve the efficiency of reverse engineering but also serve as a basis for future research on app evolution.

VIII. Acknowledgement

We thank the anonymous reviewers for their constructive comments. This work is in part supported by the National Science Foundation of China (No. 61872237). Yuanyuan Zhang is the corresponding author.

REFERENCES

[1] "Android & Google Play at Google I/O 2022," https://io.google/2022/products/android, 2023.

[2] "Android and Google Play statistics," https://www.appbrain.com/stats, 2023.

[3] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein, "Understanding the evolution of android app vulnerabilities," *IEEE Trans. Reliab.*, vol. 70, no. 1, pp. 212–230, 2021. [Online]. Available: https://doi.org/10.1109/TR.2019.2956690

[4] S. McIlroy, N. Ali, and A. E. Hassan, "Fresh apps: an empirical study of frequently-updated mobile apps in the google play store," *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1346–1370, 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9388-2

[5] D. Schmidt, "A security and privacy audit of kakaotalk's end-to-end encryption," 2016.

[6] P. Rösler, C. Mainka, and J. Schwenk, "More is less: On the end-to-end security of group chats in signal, whatsapp, and threema," in *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018, pp. 415–429. [Online]. Available: https://doi.org/10.1109/EuroSP.2018.00036

[7] K. G. Paterson, M. Scarlata, and K. T. Truong, "Three lessons from threema: Analysis of a secure messenger," in *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, J. A. Calandrino and C. Troncoso, Eds. USENIX Association, 2023, pp. 1289–1306. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/paterson

[8] Q. Zhao, C. Zuo, B. Dolan-Gavitt, G. Pellegrino, and Z. Lin, "Automatic uncovering of hidden behaviors from input validation in mobile apps," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1106–1120. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00072

[9] A. Mantovani, S. Aonzo, Y. Fratantonio, and D. Balzarotti, "Remind: a first look inside the mind of a reverse engineer," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 2727–2745. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/mantovani

[10] D. Votipka, S. M. Rabin, K. K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse engineers' processes," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 1875–1892. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-observational

[11] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 158–177. [Online]. Available: https://doi.org/10.1109/SP.2016.18

[12] "The cybersecurity workforce gap." https://www.csis.org/analysis/cybersecurity-workforce-gap, 2019.

[13] J. Mattei, M. McLaughlin, S. Katcher, and D. Votipka, "A qualitative evaluation of reverse engineering tool usability," in *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*. ACM, 2022, pp. 619–631. [Online]. Available: https://doi.org/10.1145/3564625.3567993

[14] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: scalable and precise third-party library detection in android markets," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 335–346. [Online]. Available: https://doi.org/10.1109/ICSE.2017.38

[15] J. Zhang, A. R. Beresford, and S. A. Kollmann, "Libid: reliable identification of obfuscated third-party android libraries," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 55–65. [Online]. Available: https://doi.org/10.1145/3293882.3330563

[16] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 356–367. [Online]. Available: https://doi.org/10.1145/2976749.2978333

[17] X. Zhan, L. Fan, S. Chen, F. Wu, T. Liu, X. Luo, and Y. Liu, "ATVHUNTER: reliable version detection of third-party libraries for vulnerability identification in android applications," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 1695–1707. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00150

[18] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting third-party libraries in android applications with high precision and recall," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, R. Oliveto, M. D. Penta, and D. C. Shepherd, Eds. IEEE Computer Society, 2018, pp. 141–152. [Online]. Available: https://doi.org/10.1109/SANER.2018.8330204

[19] B. Bichsel, V. Raychev, P. Tsankov, and M. T. Vechev, "Statistical deobfuscation of android applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 343–355. [Online]. Available: https://doi.org/10.1145/2976749.2978422

[20] B. Yadegari, "Automatic deobfuscation and reverse engineering of obfuscated code," Ph.D. dissertation, University of Arizona, Tucson, USA, 2016. [Online]. Available: https://hdl.handle.net/10150/613135

[21] R. Baumann, M. Protsenko, and T. Müller, "Anti-proguard: Towards automated deobfuscation of android apps," in *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems, SHCIS@DAIS 2017, Neuchâtel, Switzerland, June 21 - 22, 2017*. ACM, 2017, pp. 7–12. [Online]. Available: https://doi.org/10.1145/3099012.3099020

[22] R. D. Ghein, B. Abrath, B. D. Sutter, and B. Coppens, "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, Los Angeles, CA, USA, 11 November 2022*. ACM, 2022, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3560831.3564257

[23] H. Li, Y. Wang, Y. Zhang, J. Li, and D. Gu, "Pedroid: Automatically extracting patches from android app updates," in *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, ser. LIPIcs, vol. 222. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 21:1–21:31. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2022.21

[24] L. Li, T. F. Bissyandé, and J. Klein, "Simidroid: Identifying and explaining similarities in android apps," in *2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017*. IEEE Computer Society, 2017, pp. 136–143. [Online]. Available: https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.230

[25] V. F. Taylor and I. Martinovic, "To update or not to update: Insights from a two-year study of android app evolution," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, R. Karri, O. Sinanoglu, A. Sadeghi, and X. Yi, Eds. ACM, 2017, pp. 45–57. [Online]. Available: https://doi.org/10.1145/3052973.3052990

[26] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding android obfuscation techniques: A large-scale investigation in the wild," in *Security and Privacy in Communication Networks - 14th International Conference, SecureComm 2018, Singapore, August 8-10, 2018, Proceedings, Part I*, vol. 254. Springer, 2018, pp. 172–192. [Online]. Available: https://doi.org/10.1007/978-3-030-01701-9_10

[27] "Shrink, obfuscate, and optimize your app." https://developer.android.com/build/shrink-code, 2023.

[28] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A large scale investigation of obfuscation use in google play," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 222–235. [Online]. Available: https://doi.org/10.1145/3274694.3274726

[29] L. Li, T. F. Bissyandé, and J. Klein, "Simidroid: Identifying and explaining similarities in android apps," in *2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017*. IEEE Computer Society, 2017, pp. 136–143. [Online]. Available: https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.230

[30] H. Liu, C. Sun, Z. Su, Y. Jiang, M. Gu, and J. Sun, "Stochastic optimization of program obfuscation," in *Proceedings of the 39th*

International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017. IEEE / ACM, 2017, pp. 221–231. [Online]. Available: https://doi.org/10.1109/ICSE.2017.28

[31] C. S. Collberg and C. D. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *IEEE Trans. Software Eng.*, vol. 28, no. 8, pp. 735–746, 2002. [Online]. Available: https://doi.org/10.1109/TSE.2002.1027797

[32] "Proguard," https://www.guardsquare.com/proguard, 2023.

[33] "D8 dexer and R8 shrinker," https://r8.googlesource.com/r8, 2023.

[34] "A bytecode optimizer for android apps," https://github.com/facebook/redex, 2023.

[35] Y. Wang and A. Rountev, "Who changed you? obfuscator identification for android," in *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*. IEEE, 2017, pp. 154–164. [Online]. Available: https://doi.org/10.1109/MOBILESoft.2017.18

[36] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2169–2185. [Online]. Available: https://doi.org/10.1145/3133956.3134048

[37] P. Wang, Q. Bao, L. Wang, S. Wang, Z. Chen, T. Wei, and D. Wu, "Software protection on the go: a large-scale empirical study on mobile app obfuscation," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 26–36. [Online]. Available: https://doi.org/10.1145/3180155.3180169

[38] "Soot - A Java optimization framework," https://github.com/soot-oss/soot, 2023.

[39] "F-droid - Free and Open Source Android App Repository," https://f-droid.org/, 2023.

[40] "F-droid Packages," https://f-droid.org/packages/, 2023.

[41] "Top apps ranking - Most Popular Apps in United States," https://www.similarweb.com/apps/top/google/app-index/us/all/top-free/, 2023.

[42] "Apkpure," https://apkpure.com/, 2023.

[43] P. Calciati, K. Kuznetsov, X. Bai, and A. Gorla, "What did really change with the new release of the app?" in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. ACM, 2018, pp. 142–152. [Online]. Available: https://doi.org/10.1145/3196398.3196449

[44] A. K. Jha, S. Lee, and W. J. Lee, "An empirical study of configuration changes and adoption in android apps," *J. Syst. Softw.*, vol. 156, pp. 164–180, 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2019.06.095

[45] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. R. Choffnes, and N. Vallina-Rodriguez, "Bug fixes, improvements, ... and privacy leaks - a longitudinal study of PII leaks across android app versions," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_05B-2_Ren_paper.pdf

[46] S. Hassan, W. Shang, and A. E. Hassan, "An empirical study of emergency updates for top android mobile apps," *Empir. Softw. Eng.*, vol. 22, no. 1, pp. 505–546, 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9435-7

[47] D. Domínguez-Álvarez, D. Toniuc, and A. Gorla, "Rechan: An automated analysis of android app release notes to report inconsistencies," in *9th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MobileSoft@ICSE 2022, Pittsburgh, PA, USA, May 17-18, 2022*. IEEE, 2022, pp. 73–83. [Online]. Available: https://doi.org/10.1145/3524613.3527819

[48] J. Gao, L. Li, T. F. Bissyandé, and J. Klein, "On the evolution of mobile app complexity," in *24th International Conference on Engineering of Complex Computer Systems, ICECCS 2019, Guangzhou, China, November 10-13, 2019*, J. Pang and J. Sun, Eds. IEEE, 2019, pp. 200–209. [Online]. Available: https://doi.org/10.1109/ICECCS.2019.00029

[49] H. Wang, H. Li, and Y. Guo, "Understanding the evolution of mobile app ecosystems: A longitudinal measurement study of google play," in *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, L. Liu, R. W. White, A. Mantrach, F. Silvestri, J. J. McAuley, R. Baeza-Yates, and L. Zia, Eds. ACM, 2019, pp. 1988–1999. [Online]. Available: https://doi.org/10.1145/3308558.3313611

[50] R. Guo, Q. Liu, M. Zhang, N. Hu, and H. Lu, "A survey of obfuscation and deobfuscation techniques in android code protection," in *7th IEEE International Conference on Data Science in Cyberspace, DSC 2022, Guilin, China, July 11-13, 2022*. IEEE, 2022, pp. 40–47. [Online]. Available: https://doi.org/10.1109/DSC55868.2022.00013

[51] Y. Zhao, Z. Tang, G. Ye, X. Gong, and D. Fang, "Input-output example-guided data deobfuscation on binary," *Security and Communication Networks*, vol. 2021, pp. 1–16, 2021.