PEDroid: Automatically Extracting Patches from Android App Updates

Hehao Li
Shanghai Jiao Tong University, China

Yizhuo Wang
Shanghai Jiao Tong University, China

Yiwei Zhang
Shanghai Jiao Tong University, China

Juanru Li
Shanghai Jiao Tong University, China

Dawu Gu
Shanghai Jiao Tong University, China

Abstract

Identifying and analyzing code patches is a common practice to not only understand existing bugs but also help find and fix similar bugs in new projects. Most patch analysis techniques aim at open-source projects, in which the differentials of source code are easily identified, and some extra information such as code commit logs could be leveraged to help find and locate patches. The task, however, becomes challenging when source code as well as development logs are lacking. A typical scenario is to discover patches in an updated Android app, which requires bytecode-level analysis. In this paper, we propose an approach to automatically identify and extract patches from updated Android apps by comparing the updated versions and their predecessors. Given two Android apps (original and updated versions), our approach first identifies identical and modified methods by similarity comparison through code features and app structures. Then, it compares these modified methods with their original implementations in the original app, and detects whether a patch is applied to the modified method by analyzing the difference in internal semantics. We implemented PEDroid, a prototype patch extraction tool against Android apps, and evaluated it with a set of popular open-source apps and a set of real-world apps from different Android vendors. PEDroid identifies 28 of the 36 known patches in the former, and successfully analyzes 568 real-world app updates in the latter, among which 94.37% of updates could be completed within 20 minutes.

2012 ACM Subject Classification Software and its engineering → Software evolution

Keywords and phrases Diffing, Patch Identification, Android App Analysis, App Evolution

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.21

1 Introduction

Android apps nowadays are published at an unprecedented rate and many developers frequently update their apps for a variety of reasons such as helping maintain the robustness or introducing more competitive features. An update usually leads to multiple modifications of the app, some of which are used to improve the functionality or performance, while a significant type of modifications is to fix bugs in apps. This type of modifications, also known as *patches*, reflect how the developers fix the bug. Researchers not only learn the causes of bugs but also discover and fix similar bugs [19, 23, 22] in other apps through analyzing the information carried by patches. However, it is often unclear for analysts how Android app developers repair existing defects for lack of detailed commit logs, especially for security participants who do not have access to the source code. Thus, the gap between the updated apps and patches hinders the analysis of patches.

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 21; pp. 21:1–21:31
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

To the best of our knowledge, few approaches effectively identify patches against Android *updates* (i.e., the original and updated versions of an app). A common and simple way to retrieve existing patches is crawling from bug-tracking systems of open-source projects, such as GitHub Issue Tracker [16], where the detailed commit messages or bug reports are available to determine whether the modified methods contain patches. This approach does not work on closed-source apps that have less information to explain the reasons for updates. The descriptions about the updates of closed-source apps often only claim what feature has been added or some bugs have been repaired, but do not further explain the type, cause, and repair information of the bugs. On the other hand, compared with the open-source project, the closed-source app has a much larger amount and accounts for the majority of Android apps. As for binary-level analysis, SPAIN [45] focuses on patches in C binaries, but the huge difference between procedure-oriented and object-oriented program languages makes it unable to apply on Android apps.

Another problem to identify patches at bytecode level is how to locate modified methods in updates. Previous works [45, 38] of patch analysis on C binary utilize BinDiff [7] to achieve the goal. However, there exist few accurate diffing tools on bytecode of Android apps, due to the popularity of code obfuscation (e.g., using ProGuard [30] to protect bytecode). Most works only implement coarse-grained similarity comparison [6, 49, 39, 47] cross apps, which cannot locate the modified methods between two versions of an app, while other works [20, 43, 33] link the original methods with their updated versions by method names which cannot resist obfuscation techniques.

To address the above problems, in this paper, we propose a bytecode-level patch extraction approach, named PEDroid, to automatically locate the patches in updates of Android apps. The workflow of PEDroid consists of two phases: 1) locating the modified methods in two versions of an app, and 2) identifying patches among the modified methods. In phase 1, given the original and updated versions of an Android app, PEDroid first calculates the method-level *matching relations* based on features extracted from bytecode and the structure of the app. The method-level matching relation refers to the two versions of the same method, including identical and modified methods. With the matching relations, it filters out the identical methods whose features are identical and focuses on the modified methods. To identify patches in phase 2, we propose an effective approach to determine the patches from two aspects: 1) the call sites of the modified methods, and 2) the difference in internal semantics. In particular, PEDroid analyzes the call sites of the modified methods using a static taint analysis to check whether the methods use *external values* (i.e., external inputs or results from other methods). Then, it compares the internal semantics of the two versions of the modified methods through aligning the same operations of external values within the two methods and analyzing the modification related to these operations. Finally, PEDroid identifies the patches whose modification is used to fix the processing logic before these operations or handle the errors generated by them.

We evaluated PEDroid on two datasets of Android apps: the first set contains 13 updates of popular open-source apps, and the second one contains 568 real-world updates. We first tested PEDroid on the open-source dataset to evaluate its effectiveness. PEDroid achieves a recall of 92% in differential analysis, and successfully identifies 28 of 36 patches in patch identification. The results show that our approach effectively locates the modified methods and identifies patches. Then, PEDroid ran on the second dataset and successfully extracted 98,591 patches. Through a further manual analysis, we confirmed several types of patches including security check addition, date usage correcting, error handling, etc. For the time cost, 63.91% of the updates were analyzed within 5 minutes, 83.98% were completed within 10 minutes, and 94.37% were completed within 20 minutes. It shows that PEDroid is capable of discovering rich types of patches in real-world apps.

In summary, our work includes the following contributions:

- We propose a novel approach to extract patches from the neighboring versions of Android apps, and implement PEDroid based on the approach, which labels the identical and modified methods in given APK files, and then identifies patches among all modified methods. To the best of our knowledge, PEDroid is the first work that extracts patches from updates of close-sourced Android apps.
- Due to the lack of a standard benchmark to evaluate the accuracy of differential analysis and patch identification, we collected a dataset with 13 updates of 6 popular open-source apps, which contains 36 patches and 47 non-bugfix updates. The dataset can be used as a benchmark for future works to evaluate the performance of patch extraction.
- We also evaluate the applicability of PEDroid on 568 real-world app updates. 98,591 patches are discovered by PEDroid, including various types (e.g. adding security checks, correcting data usage). All updates are successfully analyzed and 94.37% can be completed within 20 minutes.

2 Related Works

2.1 Diffing in Android

Diffing is a common technique to compare the difference between two programs. There are numerous works to diff two versions of a program at the source code level. Git-diff tool [11] defaults input is sequential and cannot handle the changes in text order, for example, the different order of methods in a class between compilation. Furthermore, it cannot resist the broadly-used renaming obfuscation (e.g., ProGuard[30]) for sensitiveness to all characters in the text. GumTree [9] diffs two versions of abstract syntax tree (AST) of a single Java source code file and considers the different order. However, it provides only a fine-grained diffing between two class files but no method-level matching relations on apps. To retrieve matching relations, some works [32, 33, 43] link two versions of a method by defined patterns, and involves method names in patterns or similarity comparison. But it cannot either handle changes that do not follow these patterns or deal with bytecode with little symbolic information. Schäfer *et al.* [31] propose an approach to extract matching relations of methods in framework by their usage (e.g. calling and extension) in apps, which builds on the framework or test cases provided by developers. But for all methods in apps, a large proportion will be ignored by the approach. Therefore, these existing diffing tools cannot meet our requirements to locate the modified methods on bytecode.

Apart from these diffing tools, there are many bytecode-level approaches to detect similarity between two Android apps. Many previous works only extract coarse-grained features from code to resist obfuscation. For example, only method signatures are extracted as code features in several works [6, 49, 39, 47], which makes them unable to discover the modification within a method. To achieve the goal of comparing the similarity at the method level, SimiDroid [20] defaults the two methods with the same signatures (i.e., class name, method name, parameter and return types) as matched methods. Hence, the approach cannot resist renaming obfuscation. Another similarity comparison technique [8] only focusing on single methods also obtains inaccurate results. For example, method `a` and `b` of class `A` in the updated version are matched with method `b` of class `B` and method `c` of class `C` in the original version. Therefore, a more precise approach to matching at the method level is necessary.

2.2 Patch Identification

Most existing works on patch analysis focus on open-source projects. The keyword-based approach is the most common way to identify patches, and they collect patches directly from open-source project repositories by parsing reports with predefined keywords (e.g., bug, error and fault) in their issue tracking systems [26, 24, 37, 21, 17, 40]. Different from open source projects that provide formatted and exact code update information, released apps usually do not provide detailed descriptions about changed methods. Instead, they just give some brief comments about update information[1] or even nothing [29]. Hence, it is hard to locate relevant code snippets just by these text descriptions. In addition, Xinda Wang *et al.* [38] adopt a matching learning-based technique to identify security patches in open-source C projects. They conclude basic, syntactic, and semantic features of changes and train models by open-source patch datasets. However, due to the commercial competition between apps and the prevention of attackers carrying out attacks, few developers open security issues to promote research and analysis. Therefore, the lack of datasets makes it difficult to implement effectively on closed-source Android apps.

As for previous efforts at binary level, Xu *et al.* [44] generate function signatures for known patches to match, which is unlikely to discover unknown patches. SPAIN [45] identifies patches based on the heuristic that patches are less likely to introduce new semantics than other modifications, and they use the difference of registers, flags, and memory between before and after code snippets to represent the semantics. However, since the object-oriented program language (e.g., Java) is used, most registers in Android apps point to object references, and operations are usually implemented by API or method invocation instead of calculation. Therefore, the semantics of Android bytecode cannot be represented by numerical differences and such an approach is inapplicable in Android apps. To our best knowledge, there is no effective way to identify patches on Android apps.

3 Overview

The goal of our work is to understand patches and the corresponding bugs, and automatically extract patches from Android app updates. While there are a variety of ways to do so, we seek to design an applicable, automated and systematic approach. In this section, we first discuss various challenges we need to solve (Section 3.1), then give corresponding solutions against these challenges (Section 3.2), and finally describe the overview of our tool (Section 3.3).

3.1 Challenges

There will be a number of challenges in order to achieve our goal and these include:

Challenge 1. How to obtain code features. In order to retrieve matching relations, we first calculate code feature similarity. One of the most used code features between two version apps is the sequences of instructions, which describes the project updates by comparing the text line by line [11]. Another common code feature is method signature [20, 43, 33]. However, both the two features could not be applied to represent Android bytecode due to the compilers, obfuscators and even developer customization. Hence, only code order or the method signatures is not feasible in our work. Therefore, we have to first determine how to retrieve the code features.

[1] App developers usually describe the app update briefly (e.g., 'Fixed some bugs') in the `WHAT'S NEW` section of a mobile app homepage.

Challenge 2. How to retrieve the matching relations. Having the method features, the next step is to retrieve the matching relations to locate the methods that are of our interest. Since the patches are usually used to update apps, we focus on the modified methods. Unfortunately, existing studies could not retrieve matching relations at the method level concretely. Some works only detect re-used components (e.g., third-party library) by coarse-fine similarity comparison [6, 49, 39, 47] or retrieve specific matched methods by patterns and method name [20, 43, 33]. Hence, a more precise approach to matching at the method level is necessary.

Challenge 3. How to identify patches in modified methods. Having obtained the modified methods, we still need to further identify the patches. Since the lack of commit logs and open-source databases, the existing works [26, 24, 37, 21, 17, 40] cannot be applied to Android updates. And other approaches are also inapplicable because of the huge difference between procedure-oriented language and object-oriented program languages [45] or the aim to discover specific patches against our purpose [44]. Hence, how to identify the patches from modified methods is another challenge.

3.2 Solutions

As previously mentioned, if we intend to perform patch identification in Android apps, we have to face lots of challenges. Fortunately, we have obtained the following insights to address the above challenges.

Solution 1. Extracting features after removing noisy changes. Instead of calculating similarity directly on bytecode through code instruction sequences and method signatures, we combine multiple strategies to extract stable code features which eliminate the noisy changes caused by obfuscation and compilation. Specifically, two steps are involved. First, we replace volatile identifiers with specific labels to resist renaming obfuscation. Second, we divide bytecode into different code units and sort order-independent units, including basic blocks[2], fields and methods, to normalize the order.

Solution 2. Matching guided by positional relationships. We observed that *most of the code is identical between app updates, especially for the updates with small version upgrades.* Thus, to pinpoint the matching relations and further locate the modified methods, our key insight is to utilize the positional relationships in the program structure to assist in matching the modified code. Specifically, we first locate packages containing identical code features in different versions as matched packages. And then we utilize the package hierarchy [3] of the matched packages and similarity comparison to determine the matching relations of other packages. All matched packages are used to further determine the matching relations of classes and methods. Finally, those matched methods with different features are considered as modified methods.

Solution 3. Identifying patches by pinpointing buggy operation. Most unexpected behaviors of the methods are caused by the incorrect handle of the input, and the corresponding patches in the updated version are used to fix incorrect usage or handle the errors. Especially, the input comes from not only external inputs (e.g., network I/O and user interaction) but also unexpected results returned from other methods. We call them *external values.*

[2] a straight-line code sequence with no branches in except to the entry and no branches out except at the exit

[3] a tree of packages and their subpackages. It is like directory structures.

Our insight to identifying the patch is that *a patch usually fixes the processing logic before the buggy operation or handles the errors generated by the buggy operation, while the target of operation tends to involve external values.* Thus, we try to locate the buggy operation to identify patches. To achieve it, we first analyze the usage of the modified methods to check whether they use the external values, then align the original operations of external values within the two methods, and finally determine the patch by specific semantic changes. Such changes are indicated by the original operations which have different dependencies between two versions or result in extra error handling (i.e., exit or exception capture) of the method, and the operation is pinpointed as a buggy operation.

Example. To better illustrate the insight used in Solution 3, we give the motivating examples in Figure 1. The example in Figure 1a fixes the processing logic for the input by adding checks. In this case, the parameter `path` is the input of the method, and it usually accepts an *external value* when invoked, so Line 4 which indirectly depends on `path` is an operation of external values. Since the dependencies of Line 4 are modified, the operation is pinpointed as a buggy operation as our insight. Similarly, another example in Figure 1b is identified for its handling the exception generated by the deleting operation in the patch code, which is different from the original version.

```
1     private void patch1(String path) {
2         File file = new File(path);
3 +       if(file.exists()) {
4             file.delete();
5 +       }else{
6 +           Log.e("Tag", "Cannot find target file.");
7 +       }
8     }
```

(a) Fix processing logic before a buggy operation.

```
1     private void patch2(String path) {
2         File file = new File(path);
3 +       try {
4             file.delete();
5 +       } catch (Exception e){
6 +           Log.e("Tag", "Cannot delete target file.");
7 +       }
8     }
```

(b) Handle errors generated by a buggy operation

Figure 1 Examples of two types of patches. Statements with green background are added snippets in updated version.

3.3 Framework Overview

Based on the solutions to the three challenges, we design PEDroid, the first patch extraction tool on Android updates. Figure 2 depicts the workflow of PEDroid, which consists of two phases:

1. **Differential analysis.** PEDroid first establishes the structure of apps and extracts features of disassembly code (in Section 4.1). Then, it uses the package as the unit to match between the two versions of the app (in Section 4.2), and finally extracts the matching relations at the method level (in Section 4.3).
2. **Patch identification.** PEDroid extracts the modified methods in the results of differential analysis, and checks whether it is affected by external values at each call site (in Section 5.1). It then locates the operation of the external values within the method and analyzes the modification related to the operations. PEDroid reports the patch if the modification is used to fix the processing logic or handle the errors (in Section 5.2).

4 Differential Analysis

In this section, we present the design principles of differential analysis, as well as the adopted techniques. PEDroid retrieves method-level matching relations between APK updates through three steps: structure construction and feature extraction, package-level matching, and matching relation extraction.

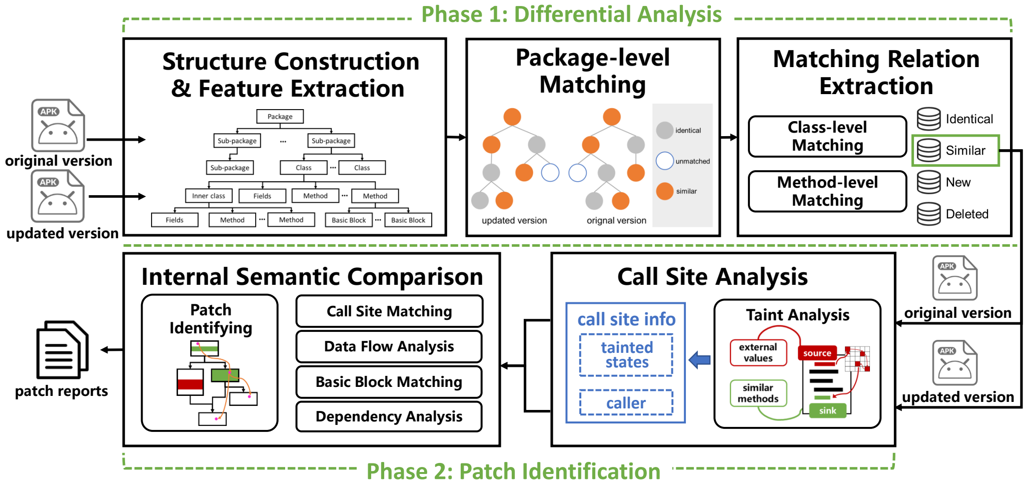


Figure 2 The workflow of PEDROID

4.1 Structure Construction & Feature Extraction

The first step of differential analysis is to disassemble the Android app and establish the app structure, including package hierarchy, classes, and code elements in classes (e.g., methods). First, PEDROID builds the relations among packages and classes by the directory structures of the disassembled app, where directories correspond to packages and files correspond to classes. Then, it parses the file content and extracts details of each class, such as fields and methods. Especially, since many nested classes (e.g., inner classes, local classes, anonymous classes, and lambda expressions) contain less information, matching them respectively will lead to false positives. To eliminate it, PEDROID recovers the nested relations and treats them as subunits of the classes they belong to. In detail, PEDROID retrieves it through system annotations from the decompiled class files, i.e., `Ldalvik/annotation/MemberClasses`, `Ldalvik/annotation/EnclosingClass`, `Ldalvik/annotation/EnclosingMethod`.

After app structure construction, PEDROID builds code features from the bottom up according to the structure. Specifically, we adopt two strategies to make the feature stable.

1. **Replacing volatile identifiers.**
   To remove the volatile parts in code, we use the specific labels to fuzz types and the instructions. First, because types contain volatile identifiers, PEDROID only retains the primitive types and framework types, and replaces others by label X to remove the noise brought by the identifiers, when extracting types involving some code elements such as fields. In this way, PEDROID converts them into the *fuzzy type*. For example, List 1 gives an example of fuzz types in a method signature. For instructions, PEDROID replaces the different types of the operand with the different labels, as shown in Table 1. Each processed instruction is called *fuzzy instruction*.
   In detail, PEDROID extracts the following feature elements for different code units:
   - **Basic Block.** The feature of a basic block consists of all the fuzzy instructions in the

Listing 1 Example for fuzzy type. `Landroid/content/Context` is a framework-type and `V` (i.e., `void`) is a primitive type. `Lcom/text/example` is replaced by `X`.

```
Original:  <init>(Landroid/content/Context;Lcom/test/example;)V
Fuzzy   :  <init>(Landroid/content/Context;X)V
```

Table 1 Rules for fuzzy instruction

Type	Label	Original instruction	Fuzzy instruction
Register	R	mov v0, v1	mov R, R
Label	L	if-eqz :const_0	if-eqz :L
Resouce ID	N	const v0, 0x7f112222	const R, N
Method/Class (except Android API)	X	invoke-virtual p0, Lcom/test/example;->call()V	invoke-virtual R, X

basic block.

- **Method.** The feature of a method includes method access flags, fuzzy types of all parameters, and the features of all basic blocks in the method.
- **Field.** The feature of a field is a string consisting of access flags, fuzzy type, and the non-default initialization value. The default initialization values (i.e. null, ‘’, 0, etc.) and names of fields are ignored.
- **Class.** The feature of a class includes the fuzzy types of superclass and interfaces, the features of fields, methods, and nested classes.

2. **Normalizing orders.**
The order-independent features such as the features of basic blocks and methods are sorted to normalize the order. It is because the extracted features without normalizing will be different because of the different orders between the two versions. Since these changes are caused by compilation rather than developers, we eliminate them. To normalize the order of fuzzy instructions with a basic block, PEDroid analyzes the dependencies of registers and sorts the order of sequential instructions without dependencies on each other. For independent units (including basic blocks, methods, fields, and classes), PEDroid directly sorts the features of the same types of the included units. For example, the features of basic blocks are sorted and then become a part of the method feature.

After extracting features and normalizing the order, PEDroid calculates the overall feature of each unit by hashing all the orderly features to represent the unit. Hence, the overall feature of a unit is calculated based on the overall hash of the included units, rather than all the feature elements of each included unit. And PEDroid records the overall features and feature elements of all units and the inclusion relations between the units.

4.2 Package-level Matching

With the app structure and the features of code elements, PEDroid calculates the matching relations between packages based on the package hierarchy, which is the sub-graph of the app structure. Specifically, PEDroid extracts identical classes, which are the two classes with identical features. And then it locates *identical packages* having at least one identical class. Among the rest packages, PEDroid utilizes their positional relations with the identical packages on the two package hierarchy to search for matching candidates, and treats the packages with the greatest similarity as *similar packages*. In summary, it includes two steps: *identical package matching* and *similar package matching*.

Identical Package Matching

PEDroid builds an identical package collection PKG_{iden}, which stores the identical package pairs. To achieve it, PEDroid first finds out the identical classes. Especially, only when the overall feature of the class in the updated version is unique and the same as the unique feature in the original version, the two version classes are regarded as identical classes. Packages with one or multiple identical classes are considered identical, and the two packages are added to PKG_{iden} as a pair. According to these rules, PEDROID obtains the matching pair collection PKG_{iden} of the identical packages, which maps an updated package to all the original packages considered to be identical. That means a package may have multiple identical classes to different packages of another version.

Similar Package Matching

Based on the identical package collection PKG_{iden} and package hierarchy, PEDROID matches similar packages by different positional relationships. Algorithm 1 represents our approach to determine similar packages from candidates. In detail, PEDROID first discovers the candidates by the positions of matched packages (which are initially identical packages) on package hierarchy and then selects the packages with the greatest similarity among candidates as similar packages.

Algorithm 1 Searching similar packages in all candidates

```
Input: Candidates set Candidate_sim
Output: Similar packages PKG_simi
PKG_simi ← ∅
map_1 : mapping new version packages to all candidates packages in old version
map_2 : mapping old version packages to all candidates packages in new version
for ⟨p_1, p_2⟩ in Candidate_sim do
    map_1[p1].add(p2)
    map_2[p2].add(p1)
end
for ⟨p_1, candidates_1⟩ in map_1 do
    p_2 ← get most similar package in candidates_1 of p_1
    candidates_2 ← map_2[p_2]
    p_1' ← get most similar package in candidates_2 of p_2
    if p_1 == p_1' then
        PKG_simi.add(⟨p_1, p_2⟩)
    end
end
return PKG_simi
```

Similarity Calculation. PEDROID quantifies similarity based on the similarity between features. Since the feature is extracted from the bottom up, the similarity between the upper units involves their bottom units. That means, before calculating the similarity of the units, the matching relations between their included units should be obtained. For example, the similarity of classes is calculated based on the matching relations between the methods in the target classes. The matched units are called *peer units*. Besides the included units, other feature elements of the same type in a unit are also regarded as *peer units*, such as the access flags of methods. Furthermore, to reflect the amount of information, we introduce *the length of feature* in similarity calculation, which means the number of basic elements contained in the feature. For example, the length of features of a basic block is the number of extracted instructions. Specifically, we define three types of similarity at different levels as follows:

Method-level Similarity. The proportion of the sum of the lengths of identical features to the total length of features of the method.

Class-level Similarity. The weighted average of the similarity between peer units where the weight is the length of features. If the class has nested classes, the similarity is added with the sum of the similarities of all nested classes.

Package-level Similarity. The sum of the similarity of peer units between two packages.

To support similarity calculation of packages, we propose the matching algorithm to retrieve the matching relations between classes in two packages and methods in two classes in Algorithm 2. PEDroid calculates the similarity between each two of the target units (i.e., classes or methods). It sorts the similarity scores from high to low and selects the matching pairs in turn. If the similarity of a pair is greater than THRESHOLD, the two units in the pair are considered similar. Considering the trade-off between false positives and false negatives, we set THRESHOLD as 0.15.

Algorithm 2 Matching relation construction at the class/method level

```
Input: Members set S1, S2 in matching targets T1, T2, similarity threshold THRESHOLD
Output: Matching relationship set R
L ← ∅
for m1 in S1 do
    for m2 in S2 do
        s ← similarity between m1 and m2
        L.put(s, ⟨m1, m2⟩)
    end
end
sort L by similarity from highest to lowest
R ← ∅
for s, ⟨m1, m2⟩ in L do
    if (s > THRESHOLD) and (R have no pair containing m1 or m2) then
        R.add(⟨m1, m2⟩)
    end
end
return R
```

Positional Relationships. A package acts as the namespace, and it usually includes a collection of classes or sub-packages with similar functions. Therefore, the positional relationships between nodes in the package hierarchy indicate the relations on function. Moreover, if a subtree, consisting of a package and all its sub-packages, represents a third-party library, which is relatively independent, changes in structure generally happen within the library. Hence, two nodes with identical child nodes (or descendants) may be similar or belong to the same library.

PEDroid first retrieves candidates by three close positional relationships, i.e., the packages that have identical parent, child, or sibling packages. The nodes, which have closer relations to others, are first considered to be potentially similar. PEDroid builds the candidate collection $Candidate_{sim}$ according to the three positional relationships to identical packages in PKG_{iden}, and then selects the most similar pairs to build the matching collection PKG_{simi}.

For the nodes which cannot be matched through the close positional relationships, PEDroid obtains the similar collection $PKG^{'}_{simi}$ through the more general positional relationships in the package hierarchy, i.e., the ancestors and descendants. Algorithm 3 gives the approach to find the ancestors with matched descendants and then locate candidates by the distance to the matched ancestors. In detail, the process of matching has a loop to search for candidates and find the most similar ones. Before the loop starts, PEDroid retrieves a set $PKG_{ancient}$ by the matched packages. It collects the node pairs having at least one matched pair in the descendant nodes. For the i^{th} subround, PEDroid considers the nodes, whose ancestor nodes with distance i are a pair in $PKG_{ancient}$, to be candidates and adds them into $Candidate^{'}_{sim}$. And then it obtains similar packages from $Candidate^{'}_{sim}$ by Algorithm 1, and adds the pairs into $PKG^{'}_{simi}$. Until all similar packages are found or the number of rounds exceeds the depth of the package hierarchy, the matching process is stopped.

Algorithm 3 Matching by the ancestors and descendants

```
Input: Unmatched packages in new and old version P1,P2, two versions of hierarchy H1, H2,
       matched packages set PKG_matched
Output: Similar packages PKG'_simi
PKG_ancient ← ∅
for ⟨p1, p2⟩ in PKG_matched do
    for k = 0 .. min(level(H1, p1), level(H2, p2)) do
        a1 ← k^th ancestor of p1 in H1
        a2 ← k^th ancestor of p2 in H2
        PKG_ancient.add(⟨a1, a2⟩)
    end
end
R1, R2 ← P1, P2
PKG'_simi ← ∅
for i = 0 .. min(height(H1), height(H2)) do
    Candidate'_sim ← ∅
    for p1 in R1 do
        for p2 in R2 do
            if i > min(level(H1, p1), level(H2, p2)) then
                continue
            end
            a1 ← i^th ancestor of p1 in H1
            a2 ← i^th ancestor of p2 in H2
            if ⟨a1, a2⟩ in PKG_ancient then
                Candidate'_sim.add( ⟨p1, p2⟩)
            end
        end
    end
    matched ← get matched packages from candidate collection Candidate'_sim
    PKG'_simi.union(matched)
    for ⟨p1, p2⟩ in matched do
        R1.remove(p1)
        R2.remove(p2)
    end
end
return PKG'_simi
```

4.3 Matching Relation Extraction

With the results of package matching, PEDROID obtains matching relations (i.e. *Identical* and *Similar*) at class and method level in matched packages. The identical classes are obtained by the identical overall features of classes, while the similar classes in identical packages collected in PKG_{iden} are matched by similarity as Algorithm 2. For the similar packages in PKG_{simi} and $PKG^{'}_{simi}$, the matching relations between classes have been calculated and cached during the matching process, and can be extracted directly.

Except for the matching relations, the unmatched classes/methods in the updated version of the app are classified as *New*, and those in the original version are classified as *Deleted*. Therefore, by calculating the similarity, the classes and their methods in the two packages are finally divided into four categories: *Identical*, *Similar*, *New* and *Deleted*.

5 Patch Identification

In this section, we introduce how PEDROID distinguishes whether a modified method contains a patch after locating the modified methods. Since the insight is that *a patch usually fixes the processing logic before the buggy operation or handles the errors generated by the* *buggy operation, while the target of operation tends to involve external values*, PEDroid analyzes the two version methods from two aspects: 1) the call sites of the methods and 2) the difference of internal semantics. Through the analysis of the call sites, PEDroid could check whether the method uses external values. Through internal semantic analysis, it locates the variables carrying external values and the original operations of these variables in the modified methods to discover potential buggy operations, and then identifies the two types of modification.

5.1 Call Site Analysis

In order to find the modified methods using external values, PEDroid employs static intra-procedural taint analysis to analyze the call sites of all modified methods. Compared with inter-procedural analysis which is more accurate but brings unacceptable overhead, the intra-procedural analysis is more suitable for us to analyze the real-world apps. And to alleviate the limitation that intra-procedural analysis cannot find external values explicitly or implicitly passed between functions, PEDroid takes the parameters and member variable as taint sources.

Since static taint analysis has been studied well, we omit its technical details for brevity here. In the following, we only describe the strategies how PEDroid selects sources and sinks and then propagates the taint.

Taint Sources. PEDroid marks the variables that may carry external values as taint sources, including parameters, member variables, and return values of method invocation statements. As a part of external values, return values of other methods are marked as sources, and external input could also be obtained by return values of Android API. Especially, the return value of the constructor method (i.e., <init>, <clinit>) without other sources is excluded for its purpose is initialization. Both the parameters and member variables could introduce external values from other methods, so PEDroid treats them as sources to avoid missing reports.

Taint Sinks. The modified methods are sinks of our taint analysis to find out whether the modified methods use external values at the call sites. PEDroid directly retrieves the methods classified as *Similar* in Section 4.3 and marks them as sinks.

Taint propagation. PEDroid mainly focuses on two types of statements, i.e., assignment and invocation, to propagate the taint.

- Assignment. If the right-hand side expression is tainted, the left-hand side value is also tainted.
- Invocation. Due to the limitation of intra-procedural analysis, it is unknown how the taint values propagate in the callee. PEDroid specifies that if a parameter is tainted, the return value and instance (if any) are also tainted, but PEDroid does not consider the possibility of taint propagation between method parameters to reduce false positives.

```
void CallerA(int arg){
    int a = this.A;
    int b = 0;
    sink(arg, a, b);
}
```

```
void CallerB(){
    int a = 10, b = 1;
    int c = d();
    sink(a, b, c);
}
```

Figure 3 Example for result extraction in call site analysis.

After taint propagation, PEDroid extracts the tainted states of the modified methods. For the tainted call sites, PEDroid records the indexes of all the tainted parameters and the caller. And the taint states of different call sites of a method will not be merged to reduce false positives. Figure 3 gives an example where method `sink` has two call sites in method `CallerA` and `CallerB`. In this case, PEDroid separately records that the first and second parameters of `sink` are tainted in `CallerA` and the third parameter is tainted in `CallerB`, rather than regards that all the parameters are tainted. This is because `sink` may only trigger a bug at the call site of `CallerA` and the invocation by `CallerB` has nothing to do with the bug. So, the operations of the third parameter in method `sink` can be ignored. On the other hand, `CallerB` may be a new method or the call site in `CallerB` may be newly introduced for feature enhancement. The operations of the third parameter within `sink` method are modified so that it can adapt to new features. Therefore, merging them will bring false positives.

In addition, Android callback techniques would bring false negatives to the approach, because callback methods are invoked in Android frameworks. They are driven by Android lifecycle events (e.g., `onCreate`), user interactions (e.g., `onClick`) and so on. To alleviate this problem, we collect the names of all Android callback methods in advance, and PEDroid treats the overriding callback methods as having identical call sites whose parameters are used to pass external values.

5.2 Internal Semantic Comparison

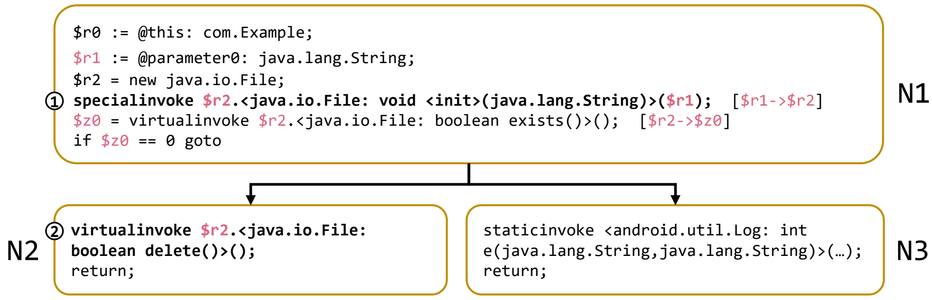
Based on the analysis of the call sites of modified methods, PEDroid identifies the patches through internal semantic comparison. Specifically, our aim is to find out whether the modification is used for correcting the processing logic or handling the errors. The former is indicated by the different dependencies of original operations, so PEDroid extracts the control and data dependencies and then compares the dependencies between two versions. As for the latter, PEDroid takes two cases into consideration. The first case is adding an exception capture operation to catch the exception generated by original operations. The second is adding checks of the return value of the original operation, while a branch of the check is a *aborting block* which aborts execution of the method when an error occurs. To identify the case, PEDroid searches for the aborting blocks by exits of methods:

1. a basic block ends with exception throwing;
2. a basic block contains only a `return` statement or logging and `return` where logging is often used to record the errors.

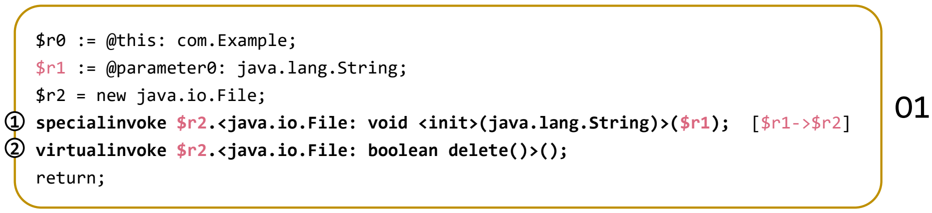
We implement it on the top of Soot [34]. And for illustration purpose, we take the patch in Figure 1a as example and give their Control flow graphs (CFG) in Figure 4. In detail, PEDroid compares the internal semantics through the following steps:

Step 1. Call site matching. With the modified methods and their usage, PEDroid matches the call sites between two versions to obtain all similar usage of the method in the app. Specifically, it matches the call sites whose callers have been identified as *Identical* or *Similar* in Section 4.3. According to the matching results, PEDroid analyzes each pair of the call sites respectively in the following steps. It is because the matched call sites represent the identical usage of the methods and different usage should be separately analyzed as discussed in Section 5.1.

Step 2. Data flow analysis. To find usage of the tainted parameters within the method, PEDroid performs forward data flow analysis in the modified method to locate all statements which use the variables directly or indirectly dependent on these parameters. It retrieves data flows through assignment and invocation statements, where the rules are



(a) Fixed version of example code.



(b) Buggy version of example code.

Figure 4 CFGs of the two versions of methods in Figure 1a. The example code is displayed in Soot intermediate representation. Registers in pink font indicate they depend on affected parameters, and the data flows are labeled after the statement as well. The bold statements are candidates of buggy operations.

similar to propagation discussed in Section 5.1. We call the located statements *affected statements*. In Figure 4, the statements with pink registers are affected statements.

Step 3. Basic block matching. To improve the accuracy of dependency comparison, PEDroid aligns the basic blocks between the two versions of methods, instead of matching at the statement level. Alignment is based on the statements in basic blocks and the structure of CFG whose nodes are basic blocks. Due to the complexity of solving the graph matching problem, we adopt a simplified strategy that utilizes the breadth-first traversal orders of CFG to flatten the graph and aligns the blocks by LCS (longest common subsequence). The identical basic blocks are the blocks with identical representative statements including `return`, `if`, exception, method invocation, and array operations and constant values in statements.

After alignment, the blocks between two matched blocks (or entry/exit) are also regarded as matched blocks that may have many-to-many matching relations. In the example, there are three-to-one matching relationships between basic blocks which map from the basic blocks N1, N2 and N3 to O1.

And with the matching relations between basic blocks, PEDroid collects the aborting blocks which have no identical basic block. Therefore, the basic block N3 is located when analyzing the example.

Step 4. Dependency analysis. With the matching relations between basic blocks, PEDroid obtains the matched statements and then filters the subset marked in Step 2. The subset of matched statements are the original operations of the external values in the methods and includes the buggy operations we focus on. We bold these statements in the examples in Figure 4. To pinpoint which operations among the candidates (i.e., matched statements in the subset) are modified satisfying our insight, PEDroid analyzes the dependency of two types of statements.

1. To distinguish the changes to fix processing logic, PEDroid extracts control and data dependencies of each candidate in original and updated versions, which will be compared in the next step.
2. To distinguish the changes to handle errors, PEDroid analyzes the data dependency of `if` statements. Specifically, if the predecessors of the aborting blocks located in Step 3 end with a `if` statement, PEDroid searches for sources of registers compared in the statement, where the sources are the assignment statements defining these registers. If a candidate is found, PEDroid will record it as having an *error value check*. In the example, although N3 is an aborting block, the register compared is irrelevant to any candidates, so it is filtered out in this step.

Step 5. Patch identifying. Finally, PEDroid determines patches by checking two types of specific changes:

1. To check the changes for fixing the processing logic, PEDroid compares the dependencies between the original and updated methods. In particular, it compares the control and data dependencies of each candidate. A patch is reported if a difference in dependencies is found.
   In Figure 4, the candidate ① has the identical control and data dependencies between the original and updated versions, so it is not a buggy operation. But the dependencies of the candidate ② are modified where the file existence check is added in the updated version. Hence, PEDroid identifies it.
2. To check the changes for handling errors, PEDroid respectively identifies two cases. First, if an exception capture is added and its predecessors contain a candidate, it is identified as a patch. And the second case is identified by the candidate that has an error value check in the updated version but no such check in the original version.

6 Evaluation

6.1 Dataset

In the experiment, we collected two datasets, the manually selected open-source Android projects from GitHub [12] named *dBench*, and APK files of pre-installed apps extracted from Android phones. The former is used to measure the accuracy and effectiveness of PEDroid, and the latter is used to evaluate the applicability to real-world apps and check whether PEDroid can discover patches on real-world apps.

dBench: we selected apps and their updates by manually reading the commit message of the projects on GitHub, and then downloaded the release version APK files for testing, to achieve the effect on the real-world apps as far as possible. The policy for selecting updates is as follows:

1. For modification of each method in an update, detailed commits can be found so that we can determine whether a commit is used to fix a bug by the title, description, or related issue;
2. This version update has at least one patch and one non-bugfix update (e.g., code refactoring and feature enhancement). Especially, PEDroid focuses on the patches which lead to the method change and filters out other commits (e.g., configure files).

Finally, *dBench* includes 6 projects with a total of 13 updates, as shown in Table 7 and Table 8. In the tables, we also list the filtered commit IDs and whether they are marked as patches. It includes a total of 83 commits, of which 36 are marked as patches. Table 2 shows the size of APK files in each update, where the size is represented by the number of classes and methods in updated versions.

Table 2 The number of classes and methods of applications in *dBench*. ProjectName_un is corresponding to each update in Table 7 and 8 for short.

Update	Classes	Methods
markor_u1	4,339	31,561
markor_u2	4,443	32,202
gpstest_u1	2,103	15,510
gpstest_u2	3,165	22,527
gpstest_u3	3,165	22,527
MaterialFiles_u1	5,822	29,637
MaterialFiles_u2	5,824	29,632
MaterialFiles_u3	7,624	42,316
andotp_u1	3,011	22,424
andotp_u2	3,996	29,155
gnucash_u1	6,688	47,398
gnucash_u2	6,690	47,414
anki_u1	14,332	135,646

Pre-installed apps: we collected pre-installed apps as a real-world app dataset. Because of the privilege permissions of pre-installed apps, the defect will lead to more serious problems. Moreover, these apps cover various categories (except games), so comprehensive types of apps can be analyzed. In detail, we collected mobile phones from six mainstream Android mobile device manufacturers, including Huawei, Motorola, Oneplus, Samsung, Vivo, and Xiaomi. In the first step, we regularly monitored app updates, and used the tool *ADB* [1] to pull the APK files from phones to the computer. For the preliminarily collected APK files, we removed duplicate files with the same hash value. Then, we used the tool *keytool* [18] to analyze the certificates of APK files, and then filtered out apps that are not signed by the vendor. Finally, the number of unique apps in our real-world dataset is 187. We regard the different APK files of an app with the minimum version gap as an update, and a total of 568 app updates are collected. The detailed amount and distribution of updated versions are shown in Table 3.

Table 3 The collected updates of pre-installed applications.

	Huawei	Motorola	Oneplus	Samsung	Vivo	Xiaomi	Total
App	42	5	25	8	28	79	187
Update	105	6	28	10	75	342	568
Major upgrade	30	1	9	0	3	34	77
Minor upgrade	16	3	6	0	19	127	171
Small update	59	2	13	10	53	181	320

6.2 Setup

Differential analysis is implemented in Python, and we disassemble the Dex bytecode of APK files by the tool *baksmali*. For patch identification, our taint analysis is based on the taint engine provided by Find Security Bugs [10], and the analysis of internal semantics is implemented in Java on top of Soot [34], a framework for analyzing and transforming Java and Android apps. In addition, PEDroid would not identify whether modified methods in the standard libraries (e.g., Android Support Library) are patches because the changes in these methods are to provide compatibility between different versions.

The experiments were performed on a server running Ubuntu 18.04 x64 with two Intel Xeon Gold 5122 Processors (each has eight logical cores at 3.60 GHz) and 128GB RAM.

6.3 Effectiveness

To measure the effectiveness of differential analysis and patch identification, we conducted a controlled experiment on *dBench*.

6.3.1 Results

In total, PEDroid found 429 modified methods which are classified as *Similar* after differential analysis and then reported 60 out of them are patches. Based on the related commits and manual analysis, the accuracy of the results will be further evaluated in Section 6.3.3 and 6.3.4. In this section, we will discuss the intermediate results and effectiveness of each phase of PEDroid.

Matching relations. 2,706 identical packages are found after identical package matching. During similar package matching, 36 packages were matched using parent-child and sibling-sibling relationships and one package was matched by ancestors and descendants. Although only one package was matched by ancestors and descendants on *dBench*, its parent package has no class to determine the similarity resulting in having no matched package, while it has no child or sibling package, so the close relationships cannot indicate the candidates for matching. Hence, matching based on ancestors and descendants is necessary for our design. In these small updates, most packages can be matched by the identical classes, and both two approaches based on positional relationships work in the process.

By class-level matching, 36,811 classes were classified as *Identical*, 251 classes were classified as *Similar*, 69 classes were classified as *New*, and 23 classes are classified as *Deleted*. Among *Similar* classes used to locate the modified methods, we found one pair of classes had the wrong matching relation. Between the two classes in the pair, a class is derived from another class in the updated version, which leads to a similar implementation and confuses matching. Unfortunately, it finally caused wrong matching relations between methods.

Modified method usage. In the call site analysis, we found a total of 1,071 call sites of *Similar* methods in updated versions, but only 893 call sites in original versions. It indicates that new call sites are introduced in the updated version of the app. Our consideration of filtering call sites in Section 5.2 is necessary.

PEDroid discovered 251 unique methods using external values by taint analysis, and 54 additional methods through the name of callback methods. We conducted a manual analysis on the filtered methods to identify false negatives. We found that most of them were filtered out because they used no external values or had no call sites (e.g., changes in the updated third-party libraries). As for false negatives, call sites of 12 methods were missing in the taint analysis. Among them, four were overriding methods because PEDroid failed to find the correct callee at the call site, and the rest came from the lack of accuracy in the implementation of taint analysis. On the other hand, due to the limitations of callback method identification, 22 callback methods could not be found, of which three methods are customized methods by developers, and 19 methods are unrecognized due to obfuscation. In short, due to the limitations of implementation, the usage of some modified methods can not be found in analysis, most of which are caused by callbacks.

6.3.2 Performance

The time cost of each update is shown in Figure 5. PEDroid completed every analysis in 6 minutes, where taking up to 336 seconds to analyze the update `anki_u1`. According to the data in Table 2 and Figure 5, it is obvious that the time cost is greatly affected by the size of APK files. Most of the time was spent on analyzing the call sites, up to

80.7% (`MaterialFiles_u1`). It is because that PEDroid checks every method in the app for searching the usage of the modified methods.

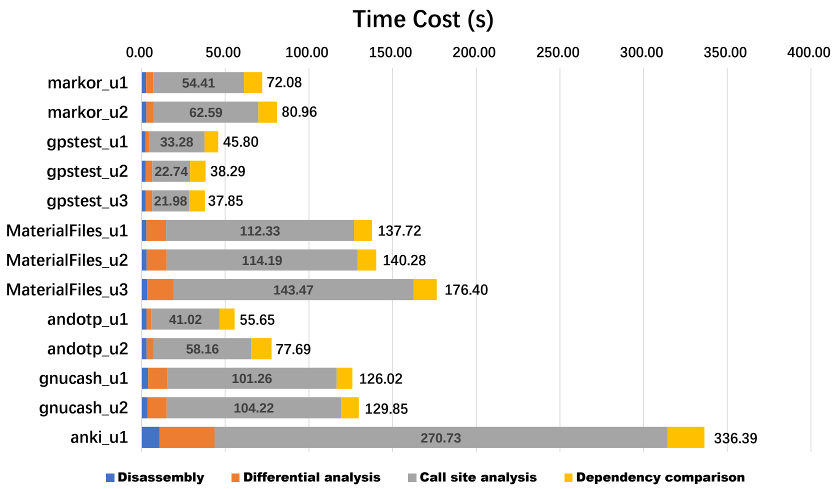


Figure 5 Time cost of each step on *dBench*.

6.3.3 Differential analysis

To evaluate the accuracy of differential analysis, we use the commits as the ground truth to check whether the modified methods are found by PEDroid. Especially, among the commits, we focus on the modifications that cause semantic changes. It means that some modifications such as renaming identifiers and merging two statements into one in commits will be ignored. In total, 238 methods have been modified by developers in *dBench*.

6.3.3.1 Accuracy

Table 4 reports the detailed results of our accuracy evaluation on *dBench*, PEDroid classified 429 methods into *Similar* category, where 234 methods belong to the project and 195 methods change with the upgrade of third-party libraries. Among the 238 modified methods, PEDroid successfully identified 221 of them, where 17 modified methods were missing. On the other hand, PEDroid mistakenly classified 13 pairs of methods as *Similar*.

It is obvious that the wrong matching relations will lead to both false negatives and false positives. For example, if two pairs $(A,\ A^{'})$ and $(B,\ B^{'})$ are modified methods, the wrong relation $(A,\ B^{'})$ brings a false positive and two false negatives to the results. Before illustrating the false negatives and the false positives, we conducted a manual analysis of the incorrect results and summarized the causes for wrong matching relations between methods.

Method inlining or extraction. Method inlining would merge multiple methods into one method, and extraction splits a method into multiple methods. In this case, PEDroid matches one of the methods with the highest similarity, which may wrongly match the new (or deleted) method and the long method of the other version.

Similar implementation. The implementation of some methods is very similar for their similar functions. It leads to similar extracted features, which confuse similarity calculation. When matching methods with similar implementation, the results may be crossed.

Large changes. The proportion of method body changes is large, especially for the methods with few features (e.g., only one or two basic blocks in the method body), the little change of code can lead to large changes in the extracted features. It leads to the correct matching relation can not be calculated, and the modified method is matched with irrelevant methods with partially the same features.

In the reported *Similar* methods, 13 pairs have wrong matching relations. Among them, five pairs are caused by the first reason, six pairs are caused by the second reason, and two are caused by the third reason.

The false negative refers to missing reports of modified methods. Among 17 false negatives, 13 of them are caused by wrong matching relations, which have been discussed before. Two false negatives were classified as *New* and *Deleted* by mistake due to large changes. The rest two were classified as *Identical* because the extracted features could not reflect the changes.

As for false positives, it indicates *New*/*Deleted*/*Identical* methods which are incorrectly classified as *Similar* methods, and *Similar* pairs with wrong matching relations. Especially, numbers in parentheses in Table 4 are the number of pairs with wrong matching relations. It shows that all the false positives came from the wrong matching relations.

6.3.3.2 Obfuscation-resistant

To address renaming obfuscation techniques is very important for our design. For example, the method `example()` in class `Example` was renamed with `A.a()` in the original version but `B.b()` in the updated version, which are different. Even if some of APK files in *dBench* do not enable the obfuscator, the third-party libraries it depends on are generally obfuscated. To evaluate how renaming obfuscation techniques influence apps, we counted the different method signatures (i.e., class name, method identifier, parameters, and return value of a method) between the original and updated version methods. Only in the *Similar* results, 135 of 429 *Similar* methods (31.5%) have different signatures. Moreover, based on manual analysis, only one signature is renamed by developers, and all the others are caused by compilation and obfuscation. It shows that the renaming obfuscation is commonly applied in apps, and PEDROID can resist it to a certain extent.

6.3.3.3 Comparison with previous works

We compared our approach with the previous works, including Androdiff [8], components of Androguard [3], and SimiDroid [20]. They can also provide method-level diffing between two versions of apps, and divide the results into four categories: *Identical*, *Similar*, *New* and *Deleted*. We used the same dataset *dBench* for experiment. The results are shown in Table 4. It is obvious that PEDROID identified much more modified methods as well retrieved less wrong matching relations, with the highest recall of 92.86%. Especially, the other two tools incorrectly regarded a large number of *Identical* methods as modified methods. Although it does not mislead patch identification, the overhead would be greatly increased. So, PEDROID is much better than the other tools.

Androdiff adopts the normalized compression distance algorithm to calculate the similarity of the two methods and extracts the instruction sequence of the basic block as the feature of the method. However, it can not resist the subtle changes caused by compilation, and most of the false positives come from the changes in the resource ID influenced by compilations. In addition, the tool does not consider the overall feature of a class and only performs similarity matching from the instructions at the method level.

SimiDroid also provides code-level similarity comparison, but it assumes that methods with identical signatures have matching relations between two versions. So, renaming ob-

Table 4 Comparison with Androguard and SimiDroid. The *Total* in the table indicates the number of reported methods, and the *TPL* and the *Project* indicate the reported similar methods in project source code and third-party library, respectively. The TP_P, FN_P, FP_P and $Recall_P$ indicate the accuracy in project code.

Tool	Total	TPL	Project	TP_P	FN_P	FP_P	$Recall_P$
Androdiff	816	525	291	105	133	186(16)	44.12%
SimiDroid	2111	1550	561	138	100	423(18)	57.98%
PEDroid	429	195	234	221	17	13(13)	92.86%

fuscation techniques have a great impact on this approach. It is the reason why SimiDroid reports much more modified methods than the other two tools, where it treats two unrelated methods as matched and detects the changes between them.

6.3.4 Patch identification

PEDroid discovered 60 patches, where 50 of them belong to the projects and 10 methods are in third-party libraries. Similar to the evaluation of differential analysis, we only evaluated the accuracy of code changes in the projects without the ground truth of third-party libraries.

6.3.4.1 Accuracy

To evaluate the accuracy of PEDroid in identifying patches, we manually identified all the patches and non-bugfix updates of all the 13 updates by analyzing their commits on GitHub. As shown in Table 7 and Table 8, among all the 83 commits in *dBench*, a total of 36 commits are identified as patch, where 47 commits are non-bugfix updates, including 35 feature updates and 12 code refactorings.

Among 36 commits containing patches, PEDroid successfully identified 28 patches during patch identification and missed eight, while it incorrectly identified seven of the 47 non-bug updates as patches. In particular, a commit could be associated with multiple modified methods. As for the amount at the method level, 41 methods were correctly identified as patches, and nine were false positives.

False negatives. The false negatives could be generally divided into three categories:

1. **Deficiency in implementation.** Four of eight false negatives come from the false negatives of call site analysis described in Section 6.3.1. It is caused by the obfuscated name of callbacks and overriding methods.
2. **Code refactoring.** We found that some patches are also accompanied by code refactoring, where the modified dependencies are encapsulated in a new method. So, PEDroid could not discover it by intra-procedural analysis, which brings two false negatives.
3. **Limitation of insight.** There are two false negatives that do not meet our insight. One is to modify the constant value in a static constructor. Another one is to add text on UI which only involves a method invocation addition without modifying any dependency.

False positives. Seven non-bugfix updates are incorrectly classified. Similarly, we also divide them into three categories:

1. **Deficiency in implementation.** One false negative comes from incorrectly matching between basic blocks. It results in different extracted dependencies at different usage of an external value.

2. **Code refactoring.** The code refactoring also leads to dependency modification, which brings two false positives to the results.
3. **Irrelevant dependency modification.** Four of the false positives are due to dependency modification irrelevant to patches. Three of them are caused by the added control dependencies, where two are to check and adapt different Android versions and one is to add a branch to enhance the feature. And the other one is introduced by the added number of parameters of the callee, which leads to the addition of data dependencies.

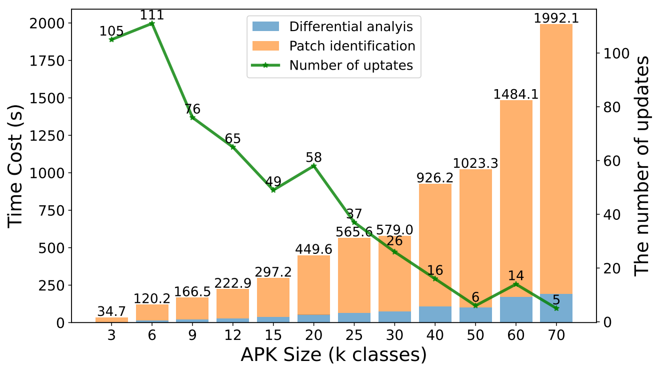
6.3.4.2 Comparison with other works

Since there is no previous work to distinguish patches from other code changes in Android apps, we evaluated whether the tool using pre-defined patterns could detect the related bugs to find out these patches. Spotbugs [35] is a state-of-the-art tool that can detect more than 400 types of bugs. Find security bugs [10] is a plugin of Spotbugs, which can detect 141 different vulnerabilities on Java and Android apps.

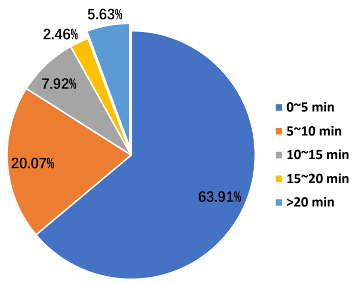
First, we applied *dBench* on the tool SpotBugs with its component Find Security Bugs, and detected the original and updated versions of the app updates respectively. Then we found out the difference of the bug reports between two versions with the method-level matching relations generated by differential analysis. Finally, only two different bug reports were found, and they belonged to one commit. It is because detecting bugs according to manually defined patterns has limitations which cannot discover the unknown bugs.

6.4 Applicability

6.4.1 Performance



(a) Time cost of analyzing updates with different sizes.



(b) Distribution of time cost of analyzing updates.

Figure 6 Performance on real world dataset.

PEDROID extracted a total number of 98,591 patches from the dataset. In detail, 45,805 patches were identified in 320 small updates, 31,549 patches were identified in 171 minor upgrades and 21,237 patches were identified in 77 major upgrades. The time cost is shown in Figure 6a, where the updates are grouped by the size of APK files (e.g., the first group consists of updates with the number of classes less than 3000, and so on). It shows that size of apps has a great impact on the overhead of PEDROID, especially for patch identification. Since the number of updates in each group is different, Figure 6a also gives the number. Furthermore, the time cost distribution of updates is given in Figure 6b. It is concluded

(a) white list

```
   @JavascriptInterface
   public void caseWhiteList(String arg){
+        if(!DomainUtils.checkUrl(WebActivity->mUrl,
                                  WebActivity.WHITE_LIST_JS)){
+            return;
+        }
       ...
   }
```

(b) black list

```
   public class Case{
       private boolean caseBlackList(DiskShowItem arg8, List arg9) {
           if(!arg8.mIsDir) {
+              if(FileUtils.checkSpecialChars(arg8.mFileName) >= 0) {
+                  return true;
+              }
               ...
           }
       }
   }

   public final class FileUtils {
       static{
           FileUtils.specialChars = new char[]{'\\', '/', ':', '*', '?',
                                               '\"', '<', '>', '|', '\n'};
           ...
       }
       public static int checkSpecialChars(String arg5) {
           if(!TextUtils.isEmpty(arg5)) {
               char[] v0 = FileUtils.specialChars;
               for(int v2 = 0; v2 < v0.length; ++v2) {
                   char v3 = v0[v2];
                   if(arg5.indexOf(v3) >= 0) {
                       return arg5.indexOf(v3);
                   }
               }
           }
           return -1;
       }
   }
```

(c) data processing

```
   public void dataProcessing() {
       ...
       String[] dirs = getDirectoryPathList();
       for(int i=0; i<dirs.length; i++){
-          File file = new File(dirs[i], "lib.so");
+          File file = new File(dirs[i].trim(), "lib.so");
           ...
       }
       ...
   }
```

(d) field addition to record status

```
   protected void onDestroy() {
       super.onDestroy();
+      if(this.mHasBinded) {
           this.unbindService(this.mServiceConnection);
+          this.mHasBinded = false;
+      }
   }
```

Figure 7 Case Study for common patches

that 63.91% of updates could be analyzed within 5 minutes, 83.98% of apps could be analyzed within 10 minutes, and 94.37% could be analyzed within 20 minutes.

6.4.2 Analysis of Extracted Patches

In order to illustrate that PEDroid can help the analysis based on patches, we made a further analysis to understand the patches extracted from updates of the pre-installed apps.

6.4.2.1 Discovered Patches

To demonstrate that PEDroid can extract effective patches from the real-world apps, we first randomly selected several reports on pre-installed apps for manual analysis. We discovered many typical cases of patches, and the security check addition appears most among them, which confirms the conclusion of the previous work [41]. Another common repair case is adding an exception-capture operation to prevent the app from crashing. In this section, we discuss the typical cases and how they improve the security and stability of apps.

Security check. Adding security checks is a common way to fix bugs. This type of patch can be detected because a new control dependency is always added. Due to complex scenarios such as network communication, local data access, and user interaction, the added security check also has various purposes, where two of the most common cases are checking whether the referenced object is null to avoid `NullPointerException`, and calling `TextUtils.isEmpty` to prevent empty strings. In addition, we show two typical cases of adding black and white list checks to discuss the security improvement by checking addition.

Figure 7(a) gives a patch with a white list check. The method has `@JavascriptInterface` annotation, which means that it can be invoked by web pages in WebView. In the fixed version of the method, the domain name of the web page which invokes this method is checked, and only the domain names in the white list are allowed to use this method, which increases the security.

The function of the method in Figure 7(b) is to download files. The security check at line 3 is added to resolve a vulnerability. The method `checkSpecialChars` checks whether

```
  public class CaseClass {
      static {
-         CaseClass.CRPYT_IV_BYTE = new byte[]{34, 0x20, 33, … , 35, 0x20, 0x20};
-         CaseClass.CRPYT_KEY_BYTE = new byte[]{33, 34, 35, … , 35, 34, 33};
          ...
      }
      public CaseClass(Context arg2) {
          ...
          this.mCryptoUtil.init(this.mContext);
+         this.CRPYT_IV_BYTE = this.mCryptoUtil.initIV();
+         this.CRPYT_KEY_BYTE = this.mCryptoUtil.initKey();
          this.loadData();
      }
      protected void loadData() {
          ...
-         String iv = Cryptor.xorKey(Case3Class.CRPYT_IV_BYTE);
-         String key = Cryptor.xorKey(Case3Class.CRPYT_KEY_BYTE);
+         String iv = Cryptor.xorKey(this.CRPYT_IV_BYTE);
+         String key = Cryptor.xorKey(this.CRPYT_KEY_BYTE);
          String data = new String(Cryptor.decrypt(iv, key, Base64.decode(cipher, 0)), "utf-8");
          ...
      }
  }
```

Figure 8 Case Study for hard-coded key removal

there are special characters in the file name. The existence of these special characters could lead to path traversal vulnerability. Once these special characters are detected, this method returns directly and does not continue downloading the target file.

Data processing. Figure 7(c) gives an example of modification of data dependencies to correct data processing. In the buggy version, the blank characters are not trimmed after obtaining the path of the directory. As a result, the corresponding library cannot be found and the function is unavailable. This patch will be reported through modification of data dependencies extracted from the invocation of the constructor of `File`.

Field addition for status recording. This patch is applied to check before resource access or release and sets the field to the corresponding value when resources are required and released. The case is found through the inconsistency of control dependencies. The case is shown in Figure 7(d).

Hard-coded key removal. A security patch of discarding the usage of hard-coded keys is given in Figure 8. The decryption key and IV used in the original version are hard-coded and defined in the static constructor (`<clinit>`). The updated version is generated in the constructor (`<init>`). PEDROID identified the patch by comparing dependencies between the two versions of the method `loadData`. In the buggy version, the hard-coded key and IV are static member variables of the class, and its acquisition has nothing to do with the affected parameter `this`. But in the fixed version, the decryption key and IV are generated at runtime, which are bound to the object instance, and have a data dependency on the parameter `this` which uses external values.

In addition to the examples of modifying the processing logic listed above, handling the errors is also commonly encountered in our manual analysis, including the error value check to end wrong execution and exception capture to prevent crashes. Since these cases are easy to understand, we would not list them here. Especially, exception capture will be further discussed later.

6.4.2.2 Application of Patches

Based on the typical patches, we further identified similar patches to find out what patches are frequently applied to fix bugs and whether the developers make the mistakes commonly. Specifically, we selected the five simple patch cases found in the manual analysis and used the buggy and fixed versions of the method and the potential buggy operations in reports to determine whether the patch is the same type as the cases. For security checks, we collected two common types, i.e., the addition of `null` and `TextUtils.isEmpty` check before the buggy operation. And we located the added invocation of `trim` which was used to correct the data processing of a buggy operation. Similarly, when a check of a `boolean` field is added and the state of the field is modified around the buggy operation, the check would be marked as field addition for status recording. For exception capture, we focused not only on the addition of exception capture but also on the types of exceptions.

Table 5 shows the usage of different types of common patches in all the extracted patches. It is reported that the check of null reference is added most commonly, similar to the results of our manual analysis. Even if we only searched a simple case of correcting data processing (i.e., string trimming), we still found that several developers at different vendors, made the same mistake and repaired it. It shows that it is a feasible means to summarize the problems that have been repaired to find similar problems in other apps.

Table 5 Usage of common patches in updates.

Type	Total
Null Reference	7682
Empty String	1409
Status Record	269
String Trimming	23
Exception	6289

Table 6 Top 10 most common types of added exception catching

Type	Total
Ljava/lang/Exception	3838
Ljava/lang/Throwable	1353
Ljava/io/IOException	1212
Ljava/lang/IllegalArgumentException	663
Lorg/json/JSONException	633
Ljava/lang/RuntimeException	457
Ljava/lang/NumberFormatException	284
Ljava/lang/IllegalStateException	234
Ljava/lang/IllegalAccessException	225
Ljava/lang/SecurityException	223

In addition, we analyzed exception-capture patches and found the types of exceptions that are easily ignored during development. Table 6 gives the top 10 most common types among our extracted patches and the number of exception-capture patches corresponding to each type. Especially, a patch could add the capture of multiple types of exceptions at the same time, so the exception-capture patches counted in Table 5 may be counted multiple times in Table 6. It shows that developers often simply use the basic type `Exception` to catch all types of exceptions, as well `Throwable` which can catch both exceptions and errors. As for other types of exceptions, the capture of `IOException` is patched most frequently in the extracted patches because it can be thrown by unexpected behaviors in a variety of scenarios including network and file I/O. The exceptions are easy to be accidentally missed by developers.

7 Discussion

7.1 Limitation and Future works

In the following, we discuss limitations and future works to improve the accuracy of the analysis performed by PEDroid.

First, PEDroid is designed to resist the renaming obfuscation because it has been broadly used by many Android applications. However, to be sensitive to code changes and efficiently retrieve matching relations, PEDroid chooses to retain features of instructions in the method body and utilizes package trees to assist the matching process. Given our current design, some advanced obfuscations can impede PEDroid to a certain degree. For example, some obfuscation tools can move a sub-package from one package to another, so as to modify the package hierarchy. Considering commonly-used obfuscators such as ProGuard do not totally break package structures, and our approach does not require the package structures to be exactly identical, we believe the selected strategies are acceptable in practice.

Second, PEDroid is mainly designed based on static intra-procedural analysis considering applicability to real-world apps. However, only analyzing the data dependencies and original operations within a single method could bring both false positives and false negatives, especially when meeting code refactoring. Meanwhile, the more precise usage of external values is more likely obtained through the inter-procedural taint analysis. We believe the inter-procedural feature could be implemented by considering method invocation, which is an interesting future work.

Third, PEDroid tries to find out patches and the corresponding bugs without manually defined patterns [19] or generated signatures of known patches or bugs [44]. Although the approach could not cover patches of all types of bugs (e.g., the two false negatives beyond the insight), it could make up for the gap in this research field to a certain degree. And we have evaluated the effectiveness by running our approach on *dBench*, and identified most patches. The results on the real-world dataset also show that rich types of bugs can be discovered through this approach.

7.2 Usage of Extracted Patches

In the paper, we discovered some typical cases of bugs and patches in Android apps and summarized the rules by manually analyzing the patches to distinguish them. Similarly, several APR (Automated Program Repair) techniques adopt manually defined code transformation schema to automatically repair bugs in Android apps [48, 25, 42, 5, 36]. Therefore, it is feasible to summarize new schemas through the analysis of the extracted patches and then apply them to APR. In addition, lots of efforts focus on learning from the existing patches which require no manually defined templates and empirical knowledge [17, 40, 26, 24, 37, 21]. However, these works are all designed for repairing source code rather than bytecode. We believe that our work can make up for the lack of learning data sets to promote the proposal of the technique on bytecode.

The extracted patches can also be used to detect similar bugs. Some binary-level similarity detection and code reuse detection techniques [15, 46] can take the buggy version of patched methods as the comparison target and detect whether there are similar problems in other apps.

8 Conclusion

We propose an approach to extract bytecode-level patches from Android apps, which includes two phases: obtaining the modified methods from the neighboring versions of Android apps and identifying patches among them. To achieve the first step and resist name-based obfuscation, we employ similarity comparison at the method level based on code features and the structure of the app. We design an approach to detect patches by analyzing the usage and internal semantics of the original and updated versions of methods. We applied the approach to extract patches from 13 updates of open-source projects and identified 28/36 patches. To evaluate the applicability to real-world apps, we further performed an experiment on the real-world dataset, which is proved that this approach can find various types of patches within a reasonable amount of time.

Acknowledgment

We are grateful to our reviewers for their valuable support and suggestions. This work was supported by the National Key Research and Development Program of China (No.2020AAA0107803).

References

1 Android debug bridge (adb), accessed: Nov. 2021. URL: `https://developer.android.com/studio/command-line/adb`.

2 Open source two-factor authentication for android, accessed: Nov. 2021. URL: `https://github.com/andOTP/andOTP`.

3 androguard, accessed: Nov. 2021. URL: `https://code.google.com/archive/p/androguard/`.

4 Ankidroid: Anki flashcards on android. your secret trick to achieve superhuman information retention, accessed: Nov. 2021. URL: `https://github.com/ankidroid/Anki-Android`.

5 Tanzirul Azim, Iulian Neamtiu, and Lisa M. Marvel. Towards self-healing smartphone software via automated patching. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 623–628. ACM, 2014.

6 Michael Backes, Sven Bugiel, and Erik 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*, pages 356–367. ACM, 2016.

7 Bindiff, accessed: Nov. 2021. URL: `https://www.zynamics.com/bindiff.html`.

8 Anthony Desnos. Android: Static analysis using similarity distance. In *45th Hawaii International International Conference on Systems Science (HICSS-45 2012), Proceedings, 4-7 January 2012, Grand Wailea, Maui, HI, USA*, pages 5394–5403. IEEE Computer Society, 2012.

9 Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

10 Find security bugs, accessed: Nov. 2021. URL: `https://find-sec-bugs.github.io/`.

11 git-difftool documentation, accessed: Nov. 2021. URL: `https://git-scm.com/docs/git-difftool`.

12 Github: Where the world builds software, accessed: Nov. 2021. URL: `https://github.com/`.

13 Gnucash for android mobile companion application., accessed: Nov. 2021. URL: `https://github.com/codinguser/gnucash-android`.

14 open-source android gnss/gps test program, accessed: Nov. 2021. URL: `https://github.com/barbeau/gpstest`.

15 Steve Hanna, Ling Huang, Edward XueJun Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers*, volume 7591 of *Lecture Notes in Computer Science*, pages 62–81. Springer, 2012.

16 Project planning for developers, accessed: Nov. 2021. URL: `https://github.com/features/issues`.

17 Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 298–309. ACM, 2018.

18 keytool, accessed: Nov. 2021. URL: `https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html`.

19 Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 802–811. IEEE Computer Society, 2013.

20 Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Simidroid: Identifying and explaining similarities in android apps. In *2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017*, pages 136–143. IEEE Computer Society, 2017.

21 Yi Li, Shaohua Wang, and Tien N. Nguyen. Dlfix: context-based code transformation learning for automated program repair. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 602–614. ACM, 2020.

22 Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.*, 3(OOPSLA):162:1–162:30, 2019.

23 Xuliang Liu and Hao Zhong. Mining stackoverflow for program repair. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 118–129. IEEE Computer Society, 2018.

24 Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 727–739. ACM, 2017.

25 Siqi Ma, David Lo, Teng Li, and Robert H. Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 711–722. ACM, 2016.

26 Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H. Deng. Vurle: Automatic vulnerability detection and repair by learning from examples. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*, volume 10493 of *Lecture Notes in Computer Science*, pages 229–246. Springer, 2017.

27 Text editor - notes & todo (for android), accessed: Nov. 2021. URL: `https://github.com/gsantner/markor`.

28 Material design file manager for android, accessed: Nov. 2021. URL: `https://github.com/zhanghai/MaterialFiles`.

29 Stuart McIlroy, Nasir Ali, and Ahmed E. Hassan. Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. *Empir. Softw. Eng.*, 21(3):1346–1370, 2016.

30 Shrink your java and android code, accessed: Nov. 2021. URL: `https://www.guardsquare.com/proguard`.

31 Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *International Conference on Software Engineering (ICSE)*, pages 471–480, New York, NY, USA, 2008. ACM.

32 Danilo Silva, João Paulo da Silva, Gustavo Jansen de Souza Santos, Ricardo Terra, and Marco Tulio Valente. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Trans. Software Eng.*, 47(12):2786–2802, 2021.

33 Danilo Silva and Marco Tulio Valente. Refdiff: Detecting refactorings in version histories. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, page 269279. IEEE Press, 2017.

34 Soot - a java optimization framework, accessed: Nov. 2021. URL: `https://github.com/soot-oss/soot`.

35 Spotbugs, accessed: Nov. 2021. URL: `https://spotbugs.github.io/`.

36 Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. Repairing crashes in android apps. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 187–198. ACM, 2018.

37 Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29, 2019.

38 Xinda Wang, Kun Sun, Archer L. Batcheller, and Sushil Jajodia. Detecting "0-day" vulnerability: An empirical study of secret security patch in OSS. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 485–492. IEEE, 2019.

39 Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. ORLIS: obfuscation-resilient library detection for android. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2018, Gothenburg, Sweden, May 27 - 28, 2018*, pages 13–23. ACM, 2018.

40 Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 479–490. IEEE, 2019.

41 Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

42 Jiayun Xie, Xiao Fu, Xiaojiang Du, Bin Luo, and Mohsen Guizani. Autopatchdroid: A framework for patching inter-app vulnerabilities in android application. In *IEEE International Conference on Communications, ICC 2017, Paris, France, May 21-25, 2017*, pages 1–6. IEEE, 2017.

43 Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 54–65. ACM, 2005.

44 Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. Patch based vulnerability matching for binary programs. In *Proc. 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Virtual Event, USA, 2020. ACM.

45 Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. SPAIN: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 462–472. IEEE / ACM, 2017.

46 Dongjin Yu, Jie Wang, Qing Wu, Jiazha Yang, Jiaojiao Wang, Wei Yang, and Wei Yan. Detecting java code clones with multi-granularities based on bytecode. In *41st IEEE Annual Computer Software and Applications Conference, COMPSAC 2017, Turin, Italy, July 4-8, 2017. Volume 1*, pages 317–326. IEEE Computer Society, 2017.

47 Jiexin Zhang, Alastair R. Beresford, and Stephan 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*, pages 55–65. ACM, 2019.

48 Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

49 Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao 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*, pages 141–152. IEEE Computer Society, 2018.

A Dataset

A.1 *dBench*

dBench includes six popular open source Android apps on GitHub shown as Table 7 and 8. Except for *markor* with 900+ stars, other projects have 1k-4.4k stars.

Table 7 Updates in *dBench* and all commits - part.1

Project	Old Version	New Version	Commit id	Bug fix
markor[27]	2.2.3	2.2.5	5b53574c88888ecbcc4b5c712d26a4c0e4f89650	✗
			464579b59047bbacb2f9fb7edb9fb9563a9dfe2c	✗
			35e25bff0de3521a41c4574561b958a8068fafa1	✗
			d0a5103223430e7af925a48f49affa0ae64ef83b	✗
			37a9c135e7a2502f8ce1b6b463614a7c10168816	✓
			9dd83708e49f45d85e2c4f3ef9cc21a3019d327d	✗
			cbd37234b587222c974b29a196f54c8f20f08b77	✗
			14cd95d37d0c12bacb2bd290bdee07d4a949ea24	✓
			47cff19dd5030d2c3ce470ce525fb2ab20f19727	✗
			22b7681cb52eb4f820c1bd036683b102be144b82	✗
			57745bb82ef225223e6780f65bc0d5dabf81cead	✓
	2.3.1	2.3.3	11895e5554c59033927a7fb5e8139797165a703d	✓
			e182dcc64057cd5f1bd8ac63492de4fa6f2f6658	✓
			51e8febed782e824ae4953bc266777828afc076e	✓
			2f5352c59e8e1edc15ad7825d3b50d0980ec70b1	✗
			46d9165b0a6f3a6a6e243fb2e8c4417c9bab0666	✓
			c9a9cc7736084355cc422b3822a8da61d58b9569	✓
			d24f2cb29d76422d5e01f69d9b01b1ff78c8c8db	✗
			63808c166aef82aaee2ed5ca67dd8a10eb2fa054	✗
			df02630b66914176f28d07a32ccde9478d20742c	✗
			6e2b07c7c1b61718904096245f9106fd14b1447e	✗
			f725a85011fc9342d37f55c58ba35926a94b6d0a	✗
			76184b2aa73a215d7e5c66a3dfee6db8f8cfad1d	✗
			27a0e8506abcdcaf2d7801493712eafb4e6ffbd7	✗
gpstest[14]	3.7.4	3.8.0	0b47fca1a9f06017b6d319269764ac6cec9b1f7b	✓
			8ed5b31c8e356b79cfe8b8bba49a10156101f758	✗
	3.9.5	3.9.6	c14a1025d6026aebef5747fb53eb28e891b02501	✓
			944733d36f44451096823200242f0ebdd5ef02c6	✓
			396c52a796e924cc5507bb087b4eadd684806fda	✗
	3.9.6	3.9.7	70d8ec5197117660e6251945e804829e5221dbbe	✓
			5625b632c4a60767950f61651629d09c8cb9fbe2	✗
MaterialFiles[28]	1.0.0-beta.11	1.0.0-rc.1	b864874d87450591f20562f1e240ff228393c554	✗
			cfcfce564e42db79a7668dbedab978a35dd01e1e	✓
			e0f488a7950402ac6464dae451b7a462898af316	✗
			8480642ddf39521eff7f30a79c5d1feec5a7d4cb	✓
			2c379913b0cf6272e1b60da265a3f7ab32cfdaaf	✗
			0d98dc34fc1cee5908514aa8eb8679f82c3d36dc	✓
	1.0.0	1.0.1	fdd9940d98974b8291496922ddb98714162b0ccc	✗
			041d384eed4cdc85d16ef063dd966a300b3b4769	✗
			428fab2cb24512e90d6d94e781134e85de29c104	✓
			fbc862d8a80bca16365dd8cfc42f0f846b0b2935	✓
			c81f380f4ec11071f139f3993987b15d3cb4a77c	✓
			4b14cefb59d746822e1f31a92ecf46e15c2d88ff	✗
	1.2.0	1.2.1	a5c07bc764c0678d423594ff454349ab63def5aa	✗
			fc22c3ada63c8392b1dcced1c96d818404ba140b	✓
			b78c799aa0f356d551c12904f07e2c9dfd3aba8e	✗
			0f0d306e5db2e2afea257449c050936c5a60a5c0	✓
			d4918e0c5a3e11d0f7e49033aa3625c5b5138da9	✗
			618806bafcf6cc424b84471d485744f96dba4b4b	✓
			ac8ca9988f761b5e8cdf7d0ecbd47d215540d145	✓

Table 8 Updates in *dBench* and all commits - part.2

Project	Old Version	New Version	Commit id	Bug fix
andOTP[2]	0.2.6	0.2.7	77655b610897eb59e6ff7fcc4f13454f34b4a86d	✗
			f0518a265c858414b74ef84e2e8bd945a96ad59a	✗
			dd97ac87f059f8c1498d17d7c99ac6dc70068ea5	✗
			f41eb620aadb3dd203f923d934ce1f6da713c901	✗
			cbdced2df1d5ab5fd35d17c7230b60a89d3d4012	✗
			247f4e938ed6def7668e3259c81a6fc9e1dd5db0	✗
			842d49b68f86412d246c9ab9a8d59dcbc11c4f8c	✗
			ce696861c7497a67c72be0a315fc9d1e5cbd0489	✓
	0.7.1	0.7.1.1	73f8c14ec389a2ad8c2a61edef2bcfd4b4894b70	✓
			cdc54028b3395401fa65665bc5e01e6a279071d3	✗
			c1d6c6b2b8c01fbfb3a0ab7ba5b3c247bf80cd3f	✗
			5215308a1afcf774499850967450725201dbb1c9	✗
gnucash-android[13]	2.1.1	2.1.2	57241e8c064302a215aa74501e0dc1ba31e6a096	✓
			1794882757a37c108c4b4cf40f6876aa7a51c87d	✓
			dae1caf7078bdd3e425e25cbfd5a37eb2309e0e6	✓
			f81ad6067a4136b34ccfc277cd21913682a3ce31	✓
			a363eebaff01f7fdadbda5edc661aa35133a450a	✗
			404759620a5a33cecf0bf836fe5802401eacf4d6	✓
	2.1.2	2.1.3	ff894a5ce5901bafc8626279d09278efc229ef23	✗
			6048bd8d0604370a38189dad9ba451aa121fc7bb	✓
			a6aa211734accf94664da91316cf6e26bed0de92	✓
			b2e9bf7f38a287985656e48ec6b13979a070dcd0	✗
			d790b805ec17fd22ab4566ae1d24cefe72486e36	✓
			724a686177798685112a02fcc3873873fb7a9595	✓
			952cb2b697b9bd946437e19db4597d23b3446f55	✓
Anki-Android[4]	2.16alpha24	2.16alpha25	a38503e08c0a8f0445adb527a015aa3a82cd4404	✗
			672c44eb664284339b697bff27ec8b37925c3c31	✓
			5135b06f4ca61cb15f75973362e2d25340925524	✗
			09430ad55c4186f5d9e5284800596527036030 8d	✗
			81d1d134863b8ab2c0560f9f11148b6a91996c0d	✓
			99ea713f780a428332990d3e5b7033d714a3ffad	✗
			b7d283f96fd3922806beb5eeb499e475f034d5a8	✗
			0f7b0bebed9539c6ee46608539be23c2e5db4780	✗