Li WJ, Ma J, Jiang YY *et al.* Understanding and detecting inefficient image displaying issues in Android apps. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 39(2): 434-459 Mar. 2024. DOI: 10.1007/s11390-022-1670-3

Understanding and Detecting Inefficient Image Displaying Issues in Android Apps

Wen-Jie Li (李文杰), *Member, CCF*, Jun Ma* (马　骏), *Member, CCF, ACM, IEEE*
Yan-Yan Jiang (蒋炎岩), *Member, CCF, ACM*
Chang Xu (许　畅), *Distinguished Member, ACM, Senior Member, CCF, IEEE*
and Xiao-Xing Ma (马晓星), *Senior Member, CCF, Member, ACM*

State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing 210023, China
Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China

E-mail: dg1633009@smail.nju.edu.cn; majun@nju.edu.cn; jyy@nju.edu.cn; changxu@nju.edu.cn; xxm@nju.edu.cn

Received June 1, 2021; accepted June 5, 2022.

Abstract Mobile applications (apps for short) often need to display images. However, inefficient image displaying (IID) issues are pervasive in mobile apps, and can severely impact app performance and user experience. This paper first establishes a descriptive framework for the image displaying procedures of IID issues. Based on the descriptive framework, we conduct an empirical study of 216 real-world IID issues collected from 243 popular open-source Android apps to validate the presence and severity of IID issues, and then shed light on these issues' characteristics to support research on effective issue detection. With the findings of this study, we propose a static IID issue detection tool TAPIR and evaluate it with 243 real-world Android apps. Encouragingly, 49 and 64 previously-unknown IID issues in two different versions of 16 apps reported by TAPIR are manually confirmed as true positives, respectively, and 16 previously-unknown IID issues reported by TAPIR have been confirmed by developers and 13 have been fixed. Then, we further evaluate the performance impact of these detected IID issues and the performance improvement if they are fixed. The results demonstrate that the IID issues detected by TAPIR indeed cause significant performance degradation, which further show the effectiveness and efficiency of TAPIR.

Keywords Android application (app), inefficient image displaying (IID), performance, empirical study, static analysis

1 Introduction

Mobile devices such as smartphones and tablets have become an indispensable part of people's daily lives. Over the past few years, we have witnessed a tremendous growth in the variety and complexity of mobile applications (apps for short). Media-intensive mobile apps must carefully implement their CPU- and memory-demanding image displaying procedures. Otherwise, user experience can be significantly affected[①]. For example, inefficiently displayed images can lead to app crash, user interface (UI) lagging, memory bloat, or battery drain, and finally make users abandon the apps even if they are functionally perfect[1].

In this paper, we empirically find that mobile apps often suffer from "inefficient image displaying (IID) issues" in which the image displaying code contains non-functional defects that cause performance degradation or even more serious consequences, such as the app crashing or no longer responding. Despite the fact that existing work has considered IID issues

Regular Paper

A preliminary version of the paper was published in the Proceedings of SANER 2019.

This work was supported by the Leading-Edge Technology Program of Jiangsu Natural Science Foundation of China under Grant No. BK20202001, and the National Natural Science Foundation of China under Grant No. 61932021. The authors would like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

*Corresponding Author

①https://developer.android.com/topic/performance/graphics, Mar. 2024.

to some extent (within the scope of general performance bugs[2–6] or partial image displaying performance analysis[7, 8]), there still lacks a thorough and in-depth study of IID issues for mobile apps, particularly for source-code-level insights that can be leveraged in program analysis for automated IID issue detection or even fixing.

To facilitate an in-depth understanding of IID issues, in this paper, we establish a descriptive framework for presenting the image displaying procedures of IID issues intuitively. In this descriptive framework, an IID issue can be represented as a code slice annotated with its triggering conditions, consequences, image processing functional modules, and error code description. Based on the descriptive framework, we conduct an empirical study towards characterizing IID issues in mobile apps. We carefully localize 216 IID issues (in 41 apps) from 2 674 issue reports and pull requests in 243 well-maintained open-source Android apps in F-Droid[②], and extract these IID issues' annotated code slices[③]. Useful findings are as follows.

1) Inappropriate handling of lots of images and large images are the primary causes of IID issues, most of which cause app crash (29.2%) or slowdown (40.7%).

2) The implementation problems that induce IID issues are four-fold: inappropriate code implementation (37.5%), lack of necessary functional modules (28.2%), misconfiguration of third-party libraries (21.8%), and using unsuitable third-party libraries (12.5%).

3) A few types of runtime behavior cover most (82.9%) examined IID issues: non-adaptive image decoding (49.1%), repeated and redundant image decoding (19.9%), UI-blocking image displaying (8.3%), and image leakage (5.6%).

4) Certain anti-patterns (AP for short) can be strongly correlated with IID issues: image decoding without resizing (23.1%), loop-based redundant image decoding (16.7%), image decoding in UI event handlers (8.3%), and unbounded image caching (3.2%).

The empirical study provides key insights on understanding, detection, diagnosing, and fixing of IID issues. Based on these findings, we design and implement a pattern-based static analyzer TAPIR[④] for IID issue detection in Android apps. We experimentally evaluate the effectiveness of TAPIR. Encouragingly, 49 and 64 previously-unknown IID issues in two different versions of 16 apps reported by TAPIR, respectively, are manually confirmed as true positives. We report these issues to respective developers, among which 16 have been confirmed and 13 have been fixed. To evaluate whether the IID issues detected by TAPIR indeed cause significant performance degradation, we conduct an experiment to measure the performance impact of the IID issues detected by TAPIR and the performance improvement after fixing these detected IID issues. The results show that most IID issues can cause tens to hundreds milliseconds unnecessary time consumption that can be avoided by fixing these issues.

In our preliminary conference version[9] of this work, we conducted an empirical study on 162 IID issues and designed TAPIR to detect potential IID issues in Android apps. In this journal version, we extend our previous work from the following perspectives: 1) We establish a well structured descriptive framework that clearly specifies the key parts (i.e., triggering condition, consequence, image processing functional modules, and error description) of an IID issue as well as their logic connections (Section 3). 2) Guided by the descriptive framework, we extend our empirical study dataset to 216 IID issues, including 54 newly collected IID issues in our empirical study subjects. We also build a dataset[③] of annotated IID issues' code slices for understanding IID issues (Section 4), which is much more informative and valuable compared with the dataset in the conference version. 3) Based on the dataset of annotated IID issues' code slices, we re-conduct the empirical study of IID issues and identify totally six new issue-inducing APIs for the anti-pattern rules used by TAPIR for detecting IID issues. Thanks to the newly added issue-inducing APIs, TAPIR reports six more IID issues than it did in the conference paper. At the same time, a comparison with the other two static anti-pattern based tools (i.e., IMGDroid and PerfChecker) is included. 4) Besides, we extend the study with an additional research question "how are IID issues introduced by developers?" to shed light on what common implementation problems Android developers have in the implementation process of image displaying and provide clues to help developers diagnose IID issues

[②] https://f-droid.org/, Mar. 2024.
[③] The datasets of all studied IID issues are publicly available at https://github.com/IID-dataset/IID-issues, Mar. 2024.
[④] https://github.com/StruggleLi/TAPIR, Mar. 2024.

(Subsection 5.2). 5) We conduct a set of experiments to demonstrate both the performance impact of the IID issues detected by TAPIR and the performance improvement if these detected IID issues are fixed (Subsection 6.2). The result confirms that the IID issues detected by TAPIR indeed cause significant performance degradation, which further shows the effectiveness and efficiency of TAPIR.

The rest of this paper is organized as follows. Section 2 introduces the background knowledge of inefficient image displaying in Android apps. Section 3 presents the descriptive framework for the image displaying procedures of IID issues. Section 4 illustrates the methodology of our empirical study for the 216 IID issues in Android apps. Section 5 discusses our empirical findings. Section 6 designs and evaluates our TAPIR tool. Section 7 discusses the potential threats to validity. Section 8 summarizes related work, and Section 9 concludes this paper.

2 IID Issues in Android Apps

In this section, we introduce the image displaying process and IID issues in Android apps.

2.1 Image Displaying in Android Apps

The process of image displaying in Android apps consists of the following four phases, which are all performance-critical and energy-consuming[⑤].

- Image loading for reading the external representation of an image (from an external source, e.g., a URL, file, or input stream) and decoding the image into an Android-recognizable in-memory object (e.g., `Bitmap`, `Drawable`, and `BitmapDrawable`).
- Image transformation for post-decoding image processing, in which a decoded image object is resized, reshaped, or specially processed for fitting in a designated application scenario (e.g., a cropped and enhanced thumbnail).
- Image storage for managing a decoded and/or transformed image object, particularly in a cache, for later rendering. Caching, on the one hand, can save CPU/GPU cycles for decoding and transformation of precious loaded and transformed images; on the other hand, it would incur huge space overhead.
- Image rendering for physically displaying an image object on an Android device's screen. Images are rendered natively by the Android framework[⑥].

2.2 Inefficient Image Displaying (IID)

Displaying a full resolution image on a high-resolution display may cost 1) hundreds of milliseconds of CPU time[4] that can cause an observable lag, and 2) tens of megabytes of memory[⑦] that can drain an app's limited memory. Therefore, the efficiency of image displaying on CPU and memory-constrained mobile devices is of critical importance. Inefficiently displayed images can severely impact app functions or user experience.

For example, decoding images in the UI thread can significantly degrade an app's performance, causing its slow responsiveness or even "app-not-responding" anomalies. Image objects not being freed in time can consume significantly large amounts of memory, leading to `OutOfMemoryError` and unexpected app terminations. Improperly stored (cached) images may cause repeatedly (and unnecessary) processing of the same images, resulting in meaningless performance degradation and energy waste.

We thus define an "inefficient image displaying (IID) issue" as a non-functional defect in an Android app's image displaying implementation (e.g., improper image decoding) that causes performance degradation (e.g., GUI lagging or memory bloat) and even more serious consequences (e.g., app crash).

3 Descriptive Framework for IID Issues

To facilitate in-depth understanding of IID issues, we establish a descriptive framework for depicting IID issues. The framework acts both as a guideline for studying IID issues and as a template for representing IID issues.

3.1 Descriptive Framework

The descriptive framework depicts an IID issue with a four-tuple $p = (t, c, M, E)$.

- $t \in T$ denotes the triggering condition of the IID issue (e.g., displaying a large image or displaying a lot of images), where T denotes the set of all possible triggering conditions of IID issues.

⑤https://developer.android.com/topic/performance/graphics, Mar. 2024.
⑥https://developer.android.com/guide/platform/, Mar. 2024.
⑦https://developer.android.com/topic/performance/graphics/load-bitmap, Mar. 2024.

- $c \in C$ denotes the consequence of the IID issue (e.g., app slowdown or app crash), where C denotes the set of all possible consequences of IID issues.
- $M = (m_1, m_2, \ldots, m_{|M|})$ denotes the sequence of functional modules involved in the IID issue's image displaying procedure. As mentioned in Subsection 2.1, the process of image displaying in Android apps consists of four phases: "image loading (IL)", "image transformation (IT)", "image storage (IS)", and "image rendering (IR)". We refer to each phase as a "type" of functional module, and let $D = \{\text{IL}, \text{IT}, \text{IS}, \text{IR}\}$ be the set of all possible types of functional modules. Then, for a functional module $m = (d, S)$, $d \in D$ denotes the type of m, and $S = (s_0, s_1, \ldots, s_{|S|})$ denotes a code statement sequence, which implements m and is contained in the IID issue's code slice.
- $E = (ES, ei)$ denotes the IID issue's error description. Specifically, $ES = \{es_1, es_2, \ldots, es_{|ES|}\}$ denotes the set of error statements (contained in the code slice) inducing the issue. $ei \in EI$ indicates the implementation problem of these error code statements (e.g., lack of necessary functional module(s), misconfiguration of third-party libraries), where EI denotes the set of all possible categories of implementation problems for IID issues.

Applying the descriptive framework to a given IID issue would produce an "annotated code slice" for the issue.

3.2 An Illustrative Example

To illustrate the details of the description framework for IID issues, Fig.1 gives an example of an annotated code slice of an IID issue in ownCloud Android app. In this IID issue, a camera photo in a folder is decoded and displayed as a thumbnail, which takes about one second and results in the app running slow.

One could understand the issue intuitively and quickly by just reading its annotated code slice without reading through the tedious texts and source codes. As indicated in Fig.1, the triggering condition of the IID issue is handing a large image (i.e., a camera photo), and the issue would result in the consequence of app slowdown. The image displaying procedure of the IID issue consists of two functional modules: one for loading image (lines 1–15) and the other for rendering image (lines 16–22). Finally, as shown in the "Error description" part of Fig.1, the issue is induced by the "inappropriate code implementation" on line 12 of the code slice, which decodes an image via the `decodeStream()` API without down-sampling.

```
Triggering condition: handling a large image
Consequence: app slowdown
Code slice:
 (1) Functional module of image loading
1   public void onStart()
2   if (getFile() != null)
3   mLoadBitmapTask=new LoadBitmapTask(mImageView, mMessageView,
      mProgressWheel);
4   mLoadBitmapTask.execute(getFile().getStoragePath());
5   protected Bitmap doInBackground(String... params)
6   Bitmap result = null;
7   String storagePath = params[0];
8   InputStream is = null;
9   File picture = new File(storagePath);
10  if (picture != null)
11  is = new FlushedInputStream(new BufferedInputStream(new
      FileInputStream(picture)));
12  result = BitmapFactory.decodeStream(new FlushedInputStream(new
      BufferedInputStream(new FileInputStream(picture))));
13  if (result == null)
14  result = BitmapUtils.rotateImage(result, storagePath);
15  return result;
 (2) Functional module of image rendering
16  protected void onPostExecute(Bitmap result)
17  if (result != null)
18  showLoadedImage(result);
19  private void showLoadedImage(Bitmap result)
20  final ImageViewCustom imageView = mImageViewRef.get();
21  if (imageView != null)
22  imageView.setImageBitmap(result);
Error description:line 12, lack code implementation of image decoding
```

Fig.1. Annotated code slice of issue 921 in ownCloud[8].

[8]https://github.com/owncloud/android/issues/921, Mar. 2024.

3.3 Obtaining Annotated Code Slices for IID Issues

An IID issue's code slice is a statement sequence $(s_1, s_2, \ldots, s_n)$ extracted from the issue's corresponding app's execution trace and the statements in the code slice directly or indirectly influence the execution of image displaying. Given an IID issue, we hypothetically execute the associated Android app under the test case that reveals the issue to obtain the corresponding execution trace. Then, we manually check the collected trace to identify the IID issue's code slice. Finally, we annotate the IID issue's code slice with corresponding triggering condition, consequence, image processing functional modules, and error description obtained by analyzing the statements in the code slice as well as corresponding issue report, pull request, and patch.

During the above process for obtaining annotated code slides, we have to read and extract different types of fragmented information scattered among many comment texts and patch codes. Furthermore, we have to combine pieces of extracted information and reason about their relations to finally determine the expected annotations. All the steps require strong intellectual processing and logical reasoning. Therefore, we currently carry out them manually. In the following parts, we will describe each step in detail.

Collecting an IID Issue's Execution Trace by Hypothetical Android App Execution. For a given IID issue, we firstly infer a test input that can reveal the IID issue by analyzing the corresponding issue report and/or pull request, as well as the IID issue's patch code (i.e., statements that are added, deleted, or modified by developers for fixing the issue in the patch), which provides information of how the IID issue is triggered and what code must be executed. Then, we use the inferred test input to hypothetically execute the buggy revision (associated with the IID issue) of the Android app and collect the hypothetical execution trace.

Extracting an IID Issue's Code Slice. We extract an IID issue's code slice by identifying the statements related to image displaying from the collected execution trace. First, we identify the statements containing image displaying API invocations[⑨] in the IID issue's execution trace. We believe that these identified statements are particularly related to image displaying. Then, we identify the image displaying related data values (e.g., image data, image displaying related parameters) in these statements by referring to Android documents[⑩]. Based on the collected statements and identified data values, we further extract all statements influencing the execution of image displaying via performing manually control- and data-dependence analysis. In particular, a statement s_i control-depends on s_j if whether s_i is executed or not depends on the outcome of s_j; a statement s_i data-depends on s_j if the value of a variable defined in s_i is calculated from a value defined in s_j. For example, a data value that determines the size of an image cache can indirectly affect the maximum number of images that can be stored in the cache. By the control- and data-dependence analysis, we can extract a statement sequence influencing the execution of image displaying in an IID issue's execution trace. We treat the statement sequence as the IID issue's code slice[10], which can focus our attention to the statements that shed light on exactly how image data is decoded, transformed, stored, and displayed.

Annotating an IID Issue's Code Slice. To present the image displaying procedure of an IID issue intuitively and help developers or researchers understand IID issues' more easily, we further annotate an IID issue's code slice with the information of triggering conditions, consequences, image processing functional modules, and error description.

An IID issue's triggering condition and consequence can be obtained by inspecting the textual information in the titles, bodies, and comments of the issue report and/or pull request. For instance, the reporter (of issue 921 in ownCloud) complained that "I have several folders with larger amounts of photos and the remote thumbnail feature is extremely slow and unreliable. I often have to explicitly refresh in the client for it to start showing thumbnails and it takes about a second to load one thumbnail." Therefore, we conclude that displaying the thumbnail for a large image would cause an explicit slowdown of the app (as shown in Fig.1).

An image processing functional module is a set of statements for specific functional purpose (such as decoding images, caching images). We identify statements for certain image processing functional mod-

[⑨]For details about the image displaying APIs, please refer to https://developer.android.com/reference/android/graphics/package-summary and https://developer.android.com/reference/android/graphics/drawable/package-summary, Mar. 2024.

[⑩]https://developer.android.com/reference/, Mar. 2024.

ules in an IID issue's code slice by checking the invoked APIs of each statement and referring to the detailed description of the functionalities of these APIs in the Android official APIs document. Specifically, from the `BitmapFactory.decodeStream()` and `BufferedInputStream(FileInputStream)` APIs used by the statement on line 12 of the code slide shown in Fig.1, we conclude that the statement is for loading an image from a given file. Similarly, the statement on line 22 is for rendering an image as it invokes the `ImageViewCustom.setImageBitmap()` API. Then, by inspecting statements forwardly and backwardly with respect to control- and data-dependence in the code slide, we could further find statements closely related to the two statements on line 12 and line 22 accordingly. By putting all closely related statements together, we finally obtain the two modules as shown in Fig.1.

Finally, an IID issue's error description can be identified by checking the issue's patches. For instance, Fig.2 shows the (simplified) patch for fixing issue 921 of ownCloud. The `decodeSampledBitmapFromFile()` method defined in `BitmapUtils` actually fetches the size of the original image (specified by `storagePath`), resizes and decodes the image with respect to the required size (specified by `minWidth` and `minHeight`, indicating the size of screen[11]). Inspecting the patch code, we conclude that the issue is induced by an inappropriate implementation of the image loading module that fails to resize (down-sample) image correctly.

4 Understanding IID Issues in Android Apps: Empirical Study Methodology

We conduct an empirical study for the purpose of better understanding IID issues in Android apps. First, we collect a set of 216 real-world IID issues (i.e., 54 more than that of the conference version[9]) by keyword search and manual inspection from 243 well-maintained open-source Android apps in F-Droid in Subsection 4.1. Then, we extract annotated code slices for the 216 IID issues based on the proposed descriptive framework and further analyze these annotated code slices around the four research questions in Subsection 4.2.

4.1 IID Issue Collection

The process of IID issue collection follows a methodology similar to those adopted in existing work[11, 12] for characterizing real-world Android app bugs, and it consists of three steps: selecting apps, identifying image-related performance issue reports and pull requests, and collecting IID issues and buggy code. Fig.3 illustrates the overall issue collection process.

Selecting Apps. We collect 243 Android apps from 1 093 randomly selected Android apps in F-Droid as our study subjects, meeting the following criteria:

1) *open-source*: hosted on GitHub with an issue tracking system for tracing potential IID issues;

2) *well-maintained*: having over 100 code commits in the corresponding GitHub repository;

3) *of realistic usage*: having over 1 000 downloads on the Google play market.

Identifying Image-Related Performance Issue Reports and Pull Requests. An issue report (IRep for short) usually denotes a manifested app bug from end users. An pull request (PR for short), on the other hand, possibly contains the developer's perspective on a concerned app bug. Therefore, we collect both of them in the empirical study. We first identify IReps and PRs involving images in the GitHub repositories by searching in their issue tracking systems with the following keywords[12]: "image", "bitmap", "decode", "di-

```
1  - result = BitmapFactory.decodeStream(new FlushedInputStream(
         new BufferedInputStream(new FileInputStream(picture))));
2  + Point screenSize = DisplayUtils.getScreenSize(getActivity());
3  + int minWidth = screenSize.x;
4  + int minHeight = screenSize.y;
5  + result = BitmapUtils.decodeSampledBitmapFromFile(storagePath,
             minWidth, minHeight);
```

Fig.2. Simplified patch fixing issue 921 in ownCloud. "+" and "–" denote the code added and deleted to fix this bug, respectively.

[11] Actually, `minWidth` and `minHeight` can be further optimized with respect to the size of the widget for showing the image, but the developer adopted this patch as it is enough to display the thumbnail in a reasonable time.

[12] These keywords are general natural language words related to image displaying. They come from existing research work, e.g., [4, 5] and our empirical study experience.

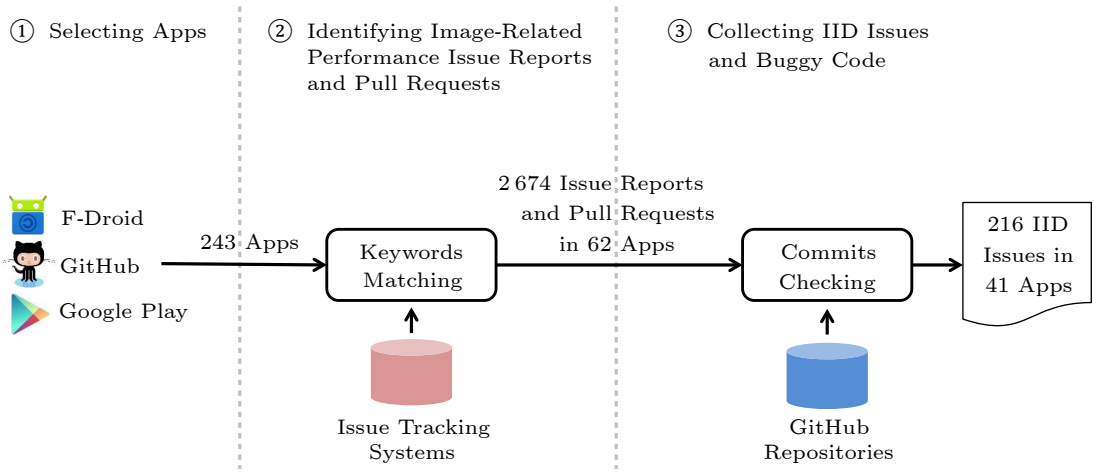


Fig.3. IID issue collection process.

splay", "picture", "photograph", "show", and "thumbnail".

Any IRep or PR that contains any of the above keywords in its title, body, or comments, is included in our initial collection. There are totally 41 334 IReps/PRs reported after May 2011 in the selected 243 Android apps, at the moment (i.e., May. 2021) of data collection[13]. We then manually inspect each IRep/PR to further confirm whether it indeed fixes any performance bug with the following criteria.

1) The IRep's/PR's text complains about the performance degradation or more serious consequences (e.g., app crash) when performing image displaying.

2) There is evidence that a bug is fixed (e.g., the concern issue report is associated with a fixing commit ID or an accepted fixing patch), and the same issue has never been re-reported in the following three months[14].

After the manual inspection, we obtain a total of 2 674 image-related performance IReps/PRs in 62 Android apps.

Collecting IID Issues and Their Buggy Code. Although each of the remaining 2 674 IReps/PRs is image-related and involves a performance degradation or more serious consequences, not all of them are guaranteed related to any IID issue. Therefore, we further inspect the fixing commits associated with these IReps/PRs to decide whether they correspond to IID issues or not. Generally, each fixing commit consists of one or more patches, each of which may patch several places in a file or multiple files. For each code patch fixing a particular image-displaying-related performance bug that is clearly documented in the corresponding IReps/PR, we consider the patch related to a new IID issue. Finally, we collect totally 216 IID issues (distributed in 103 IReps/PRs) in 41/243 (16.9%) studied Android apps, suggesting that IID issues are not rare, but common in practice and worthy of an in-depth study.

Then, for each identified IID issue, we restore the statements modified, added, or deleted in the patch to obtain the buggy code and the buggy revision of the corresponding Android app. As such, we obtain the issue's textual descriptions in the corresponding IRep/PR, its buggy code, its buggy revision of Android app, and a patch for fixing it.

4.2 Research Questions

We extract the 216 IID issues' annotated code slices following the process described in Subsection 3.3. Based on the extracted annotated code slices, we organize the study of IID issues around the following four research questions.

*RQ*1. *What are the consequences and triggering conditions of IID issues?*

Understanding IID issues' triggering conditions and consequences can provide useful implications on how to design efficient test cases and oracles to trigger and identify IID issues, respectively.

*RQ*2. *How are IID issues introduced by developers?*

[13]The issues reported by TAPIR are excluded from the collection to avoid bias.

[14]For those issues that do not contain any explicit link to any patch, we conduct a bisect on their GitHub repositories to find potential fixing patches by following the methodology of existing work[13].

Investigating implementation problems of IID issues can provide practical programming guidance to developers and assist them in diagnosing and fixing IID issues.

*RQ*3. *What is the runtime behavior of IID issues?*

The functional module sequence and error description in each IID issue's annotated code slice can shed light on how image data is decoded, transformed, stored, and rendered. By analyzing them, we can clearly know under what runtime behaviors IID issues are likely to occur and these runtime behaviors can provide useful implications for the root causes of IID issues.

*RQ*4. *Are there common anti-patterns for IID issues?*

By inspecting the error statements indicated in the annotated code slices, we could find whether there are common anti-patterns correlated to IID issues. Specifically, in this paper, we define anti-patterns as recurrent source-code level mistakes which cause IID issues, and we are particularly interested in code patterns which can facilitate lightweight static `lint`-like checkers.

5 Empirical Study: Results

In this section, we present our empirical study results by answering the four research questions described in Subsection 4.2.

5.1 Answering RQ1: What Are the Consequences and Triggering Conditions of IID Issues?

We answer RQ1 by inspecting the consequences and triggering conditions in each IID issue's annotated code slice. The overall results are summarized as the follows.

Finding 1. "Most IID issues cause app crash (29.2%) or slowdown (40.7%). Handling lots of images (55.0%) and large images (45.0%) are the major trigger conditions."

This finding[15] is consistent with our intuitions: IID issues typically occur in media-intensive apps and may result in severe impact on user experience[16]. Their consequences can be categorized as follows.

- *App Crash.* Of the 216 studied IID issues, 63 (29.2%) directly cause the corresponding apps to crash. In most cases, an `OutOfMemoryError` would be thrown when allocating memory for storing a large image[17].
- *App Slowdown.* There are 88 studied IID issues (40.7%) lead to GUI lagging[18] and/or slow image displaying[19].
- *Memory Bloat.* For 24 (11.1%) studied IID issues, the corresponding apps' consumed memories keep growing without lag or crash[20], which may lead to unnecessary stops of background activities/services and affect user experience.
- *Abnormal Image Displaying.* Images are failed to display for 21 (9.7%) studied issues. In most cases, the corresponding app's memory is insufficient for decoding large images without causing a crash, which may also trigger frequent garbage collection (GC) and impact user experience.
- *Application Not Responding.* Three (1.4%) studied issues cause application not responding (ANR), which is the extreme case of app slowdown and is usually caused by an app performing time-consuming image displaying operations in the UI thread[21].
- *Others.* Besides, 23 studied issues (10.7%) can also result in bad user experience but their IReps/PRs lack further details for inspection.

We find that 160 of the 216 studied IID issues' annotated code slices contain information about their triggering conditions. All these triggering conditions concern handling large images (72/160, 45.0%), handling lots of images (88/160, 55.0%), or both (2/160, 1.3%). For these cases, inefficient handling of large/lots of images mostly causes app crash/slowdown.

These findings, although seemingly straightforward, provide actionable hints for reasonable workload designs and possible test oracles for the automated detection of IID issues. Simply feeding an app with

[15] The finding of RQ1 is similar to that of our preliminary conference version[9] except for the data statistics. The same situation also appears in the study results of RQ2 and RQ4.

[16] An IID issue may have multiple consequences or causes, and thus the sum of the concerned percentages may exceed 100%.

[17] https://github.com/the-blue-alliance/the-blue-alliance-android/issues/588, Mar. 2024.

[18] https://github.com/nikclayton/android-squeezer/issues/171, Mar. 2024.

[19] https://github.com/kontalk/androidclient/issues/789, Mar. 2024.

[20] https://github.com/romannurik/muzei/issues/383, Mar. 2024.

[21] https://github.com/ccrama/Slide/issues/1639, Mar. 2024.

reasonably large-amount and large-size images would suffice as an IID testing adversary, and test oracles can also be accordingly designed around the studied consequences.

5.2 Answering RQ2: How Are IID Issues Introduced by Developers?

By analyzing error statements in the annotated code slices, we can learn common source-code level problems (made by developers) that introduce IID issues.

Finding 2. "The implementation problems introducing IID issues can be categorized into two general types: custom implementation specific and third-party library specific. The former type occurs when using developers' custom functionalities for image displaying, including inappropriate code implementation (37.5%) and lack of necessary functional modules (28.2%). The latter type occurs when using third-party libraries for image displaying, including misconfiguration of third-party libraries (21.8%) and using unsuitable third-party libraries (12.5%)."

This finding shows that most IID issues are induced by developers' custom implementation of image displaying, and that even though many mature image displaying libraries (e.g., Glide[22], Pacasso[23]) can alleviate image-related performance issues, apps still misuse them and suffer from IID issues. This finding can be used to assist developers and researchers on diagnosing and fixing IID issues in Android apps.

5.2.1 *Custom Implementation Specific IID Issues*

To make Android apps lightweight[24], easy to maintain, and meet specific functional requirements, some developers customize the implementation of image displaying in their apps. As a result, it brings a burden on developers who need to ensure the correctness and efficiency of the implementation. Unfortunately, this is a non-trivial task and 142 (65.7%) IID issues in our dataset are related to custom implementations. Such issues can be divided into two categories: 1) lack of necessary functional modules, and 2) inappropriate code implementation.

1) *Lack of Necessary Functional Modules*. There are 61 (28.2%) IID issues caused by lacking necessary functional modules. When dealing with a variety of image displaying scenarios, such as displaying lots of images and displaying large images, developers need to add necessary functional modules (e.g., image caching, image resizing) in the implementation of image displaying so as to ensure performance efficiency. However, many developers lack experience in implementing efficient image displaying and do not fully consider the running scenarios of image displaying that their apps may encounter, which results in lacking necessary functional modules in their customized implementation of image displaying and raises IID issues. To ease understanding, we take issue report 299 of Subsonic[25], a music player app, as an example (Fig.4 shows the simplified code of this issue). Every time Subsonic opens an image that has been displayed before, it re-decodes the image (line 5), which would affect end-user experience. The cause of the issue is that the app lacks a functional module of image caching to store the decoded images that has been displayed, resulting in duplicate decoding of the same images. Subsonic's developers later fixed the issue by adding an image cache (lines 7-9, 15-25).

2) *Inappropriate Code Implementation*. There are 81 (37.5%) IID issues caused by inappropriate code implementation. For some Android apps, even though developers have included all necessary functional modules in the implementation of image displaying, their customized implementations are often problematic and raise IID issues. For example, issue 921 of ownCloud, mentioned in Subsection 3.3, was induced by an inappropriate implementation of the image loading module that fails to resize (down-sample) image correctly. Another example is issue 6 of Atarashii (shown in Fig.5). The app crashes because of the wrong implementation of image cache, such that it gathers all decoded images without releasing (line 15).

5.2.2 *Third-Party Library Specific IID Issues*

We find that 74 (34.3%) IID issues are third-party library related. There are several popular third-party libraries (e.g., Glide, Picasso) that can be used to reduce implementation efforts and speed up devel-

[22] https://github.com/bumptech/glide, Mar. 2024.
[23] https://github.com/square/picasso, Mar. 2024.
[24] https://github.com/Neamar/KISS/issues/570, Mar. 2024.
[25] https://github.com/AnimeNeko/Atarashii/issues/6, Mar. 2024.

```
  public class DSubWidgetProvider extends AppWidgetProvider {
    private void performUpdate(Context context, …) {
      int size;
       …
-     Bitmap bitmap = currentPlaying == null ? null:FileUtil
                    .getAlbumArtBitmap(context,…, size);
+     ImageLoader imageLoader = SubsonicActivity
                                .getStaticImageLoader(context);
+     Bitmap bitmap = imageLoader == null ? null : imageLoader
                    .getCachedImage(context, …, large);
      …
  } }
  public class ImageLoader {
    …
+   public Bitmap getCachedImage(Context context, …) {
+     int size = large ? imageSizeLarge : imageSizeDefault;
+     Bitmap bitmap = cache.get(getKey(…);
+     if(bitmap == null || bitmap.isRecycled()) {
+       bitmap = FileUtil.getAlbumArtBitmap(…);
+       String key = getKey(entry.getCoverArt(), size);
+       cache.put(key, bitmap);
+       cache.get(key);
+     }
+     return bitmap;
+   }
    …
  }
```

Fig.4. Image decoding without resizing in issue 299 of Subsonic[26] (simplified).

```
  public class CoverAdapter<T> extends ArrayAdapter<T> {
    public View getView(...) {
      a = objects.get(position));
      ImageView cover = v.findViewById(R.id.coverImage);
      imageDownloader.download(a.getImageUrl(), cover);
  } }
  public class ImageDownloader {
    public void download(String url,ImageView imageView) {
      String filename = String.valueOf(url.hashCode());
      File f = new File(getCacheDirectory(imageView.getContext
          ()), filename);
      Bitmap bt = null;
      bt = (Bitmap)imageCache.get(f.getPath());
      if (bt == null){
        bt = BitmapFactory.decodeFile(f.getPath());
-       imageCache.put(..., bt);
+       imageCache.put(..., new WeakReference<Bitmap>(bt));
        imageView.setImageBitmap(bt);
 } } }
```

Fig.5. Unbounded image caching in issue 6 of Atarashii[27] (simplified, taken from [9]).

opment of image displaying in Android apps. However, in practice, due to the unfamiliarity with third-party libraries, it is hard for developers to avoid making mistakes when using them, and IID issues may arise. Concerning our study, there are two primary implementation mistakes in this category: 1) using unsuitable third-party libraries, and 2) misconfiguration of third-party libraries.

1) *Using Unsuitable Third-Party Libraries.* There are 27 (12.5%) IID issues caused by using unsuitable third-party libraries. For the third-party libraries used for image displaying, different libraries may have different functional concerns and performances (e.g., runtime or memory overhead) when handling the

[26] https://github.com/AnimeNeko/Atarashii/issues/6, May 2022.
[27] https://github.com/daneren2005/Subsonic/issues/299, Mar. 2024.

same running scenario of image displaying. Thus, developers should choose a suitable third-party library that satisfies their actual requirements. However, some Android apps contain IID issues for the reason that developers use unsuitable third-party libraries that cannot well handle the running scenarios of image displaying these apps encounter. Taking pull request 577 of AmazeFileManager as an example (see Fig.6), AmazeFileManager's developers originally used Glide for image displaying and encountered an IID issue of `OutOfMemory` exception (line 8). To prevent the exception, AmazeFileManager's developers replaced Glide with another popular third-party library Picasso (line 9). Picasso can display an image at the lowest possible resolution without affecting user experience on the quality of displayed images, which can significantly reduce AmazeFileManager's memory consumption for images displaying.

2) *Misconfiguration of Third-Party Libraries.* There are 47 (21.8%) IID issues caused by misconfiguration of third-party libraries. Third-party libraries for image displaying are often equipped with dozens of configuration options[29] allowing customization to different workloads, many of which greatly affect image displaying performance. Unfortunately, properly setting these configurations is challenging for developers due to the complex and dynamic nature of image displaying workloads, which makes misconfiguration of third-party libraries one of the major root causes of IID issues. Taking issue 1 071 of AntennaPod as an example (in Fig.7), the app's GUI lags due to the misconfiguration of the third-party library Glide. In this issue, when a user browses a list of images and slides up and down, a lot of images would be decoded repeatedly, which results in high unnecessary runtime overhead, leading to GUI lagging. The issue is caused by the `DiskCacheStrategy.SOURSE` option used in `diskCacheStrategy()` (line 7). The option indicates that Glide caches only the original full-resolution image but not those transformed versions (e.g., obtained by resizing the original one) which are actually displayed in the app. Then, each time the app displays an image, it redoes the transformation process. One developer later fixed this issue by replacing `DiskCacheStrategy.SOURSE` with `DiskCacheStrategy.ALL` (lines 7 and 8) so that Glide can cache and reuse all image versions.

```
1    public class ImageViewer extends BaseActivity {
2      public void onCreate(Bundle savedInstanceState) {
3        AspectRatioImageView imageView = (AspectRatioImageView)
4        findViewById(R.id.image);
5        Intent intent = getIntent();
6        if(intent!=null){
7          String path=intent.getStringExtra("path");
8  -       Glide.with(this).load(path).into(imageView);
9  +       Picasso.with(this).load("file://"+path).fit()
                 .into(imageView);
10       }
11   } }
```

Fig.6. Using unsuitable third-party library in PR 577 of AmazeFileManager[28](simplified).

```
1    public class PodcastListAdapter extends ArrayAdapter<
          GpodnetPodcast> {
2      public View getView(int position, ...) {
3        GpodnetPodcast podcast = getItem(position);
4        Glide.with(convertView.getContext())
5             .load(podcast.getLogoUrl())
6             .placeholder(R.color.light_gray)
7  -          .diskCacheStrategy(DiskCacheStrategy.SOURCE)
8  +          .diskCacheStrategy(DiskCacheStrategy.ALL)
9             .into(holder.image);
10   } }
```

Fig.7. Loop-based redundant image decoding in pull request 1 071 of AntennaPod[30](simplified, taken from [9]).

[28]https://github.com/TeamAmaze/AmazeFileManager/pull/577, Mar. 2024.

[29]Please refer to https://bumptech.github.io/glide/doc/configuration.html or https://guides.codepath.com/android/Displaying-Images-with-the-Picasso-Library for an example, Mar. 2024.

[30]https://github.com/AntennaPod/AntennaPod/pull/1071, Mar. 2024.

5.3 Answering RQ3: What Is the Runtime Behavior of IID Issues?

Finding 3. "Only a few runtime behavior types cover most (82.9%) inspected IID issues: non-adaptive image decoding (49.1%), repeated and redundant image decoding (19.9%), UI-blocking image displaying (8.3%), and image leakage (5.6%)."

This finding reveals that existing performance bug detectors cover only a narrow range of IID issues and that it is worthwhile to develop IID-specific analysis techniques and tools. For example, the existing pattern-based analysis[2] detects only a small portion of image decoding in the UI thread, the existing resource leakage analysis[14] can be expanded to manual image resource management (the tool itself does not cover), and existing image displaying performance analysis[8] can help developers improve the rendering performance of slow image displaying.

Besides, this finding also suggests that static program analysis techniques concerning these particularly recognized runtime behavior may be effective for detecting IID issues, as long as one can semantically model the image displaying process in an app's source code, or find particular code anti-patterns that correlate to these runtime behaviors (studied later in Subsection 5.4).

We describe the runtime behavior of an IID issue using a simplified data-flow model. An IID issue's code slice can be represented by a sequence of chronologically sorted events $E = \{e_1, e_2, \ldots, e_m\}$. Some events may be the results of image-related API invocations. Each of such events is associated with an image object $im_{w\times h}$ in the heap of resolution $w \times h$. We use the notation $e' \rightarrow e$ to denote that event e' is data-dependent on event e, i.e., the result of e' is computed directly or indirectly involving the result of e.

Non-Adaptive Image Decoding. Nearly half (106/216, 49.1%) of the issues are simply caused by decoding a large image without considering the actual size of the widget that displays this image, resulting in significant performance degradation and/or crash. A typical example is to decode a full-resolution image for merely displaying a thumbnail[31]. Issue 921 of ownCloud (Fig.1) and issue 5 701 of WordPress (Fig.8) are also examples of this kind.

For a non-adaptive image decoding case, there exists an image object $im_{w\times h}$ associated with event $e_{\text{dec}} \in E$ which is the result of an image decoding API invocation, and im is finally displayed by event $e_{\text{disp}} \in E$, which is an image displaying API invocation and $e_{\text{dec}} \rightarrow e_{\text{disp}}$. However, the actually displayed image $im'_{w'\times h'}$ is smaller than $im_{w\times h}$ (i.e., $w' < w \wedge h' < h$).

Repeated and Redundant Image Decoding. Quite a few (43/216, 19.9%) issues are due to improper storage (particularly caching) of images such that the same images may be repeatedly and redundantly decoded, causing unnecessary performance degradation and/or battery drain. An indicator of this type of IID issues is that there are two image decoding API invocation events $e_{\text{dec}}, e_{\text{dec}}' \in E$ whose respective associated images im and im' are identical, i.e., $im_{w\times h} = im'_{w\times h}$. The pull request 1 071 of AntennaPod (shown in Fig.7) is an example of this type.

UI-Blocking Image Displaying. Although the Android documentation[33] explicitly discourages decoding images in an app's UI thread, a few (18/216, 8.3%) issues still fall into this category. A typical example is to decode large images in the UI thread[34], which causes UI blocking, leading to noticeably slow responsiveness. Issue 5777 of WordPress shown in Fig.9 is an example of this type.

```
1    public class AztecImageLoader implements Html.ImageGetter {
2      public void loadImage(String url, ..., int maxWidth) {
3  -     Bitmap bitmap = BitmapFactory.decodeFile(url);
4  +     int orientation = ImageUtils.getOrientation(..., url);
5  +     byte[] bytes = ImageUtils.createThumbnail(Uri.parse(url),
                 maxWidth, ...);
6  +     Bitmap bitmap = BitmapFactory.decodeByteArray(bytes, 0,
                 bytes.length);
7        BitmapDrawable bitmapDrawable = new  BitmapDrawable(
                 context.getResources(), bitmap);
8        callbacks.onImageLoaded(bitmapDrawable);
9    } }
```

Fig.8. Image decoding without resizing in issue 5701 of WordPress[32](simplified, taken from [9]).

[31]https://github.com/opendatakit/collect/issues/1237, Mar. 2024.
[32]https://github.com/wordpress-mobile/WordPress-Android/issues/5701, Mar. 2024.
[33]https://developer.android.com/topic/performance/graphics/load-bitmap, Mar. 2024.
[34]https://developer.android.com/topic/performance/graphics, Mar. 2024.

```
1    public class PreviewActivity extends AppCompatActivity {
2      protected void onCreate() {
3        mediaUri = media.getUrl();
4        loadImage(mediaUri); }
5      private void loadImage(String mediaUri) {
6  -     byte[] bytes = ImageUtils.createThumbnail(Uri.parse(
                 mediaUri), ...);
7  +     new LocalImageTask(mediaUri, size).executeOnExecutor(
       AsyncTask.THREAD_POOL_EXECUTOR);
8  -     Bitmap bmp = BitmapFactory.decodeByteArray(bytes, ...);
9    } }
10 +   private class LocalImageTask extends AsyncTask<...> {
11 +     protected Bitmap doInBackground(Void... params) {
12 +       byte[] bytes = ImageUtils.createThumbnailFromUri(...,
                 Uri.parse(mMediaUri);
13 +     return BitmapFactory.decodeByteArray(bytes, ...);
14   } }
15   public class ImageUtils {
16     public static byte[] createThumbnail(Uri imageUri, ...) {
17       Bmp = BitmapFactory.decodeFile(imageUri, ...);
18     } }
```

Fig.9. Image decoding in UI event handlers in issue 5777 of WordPress[35] (simplified, taken from [9]).

Image Leakage. A few (12/216, 5.6%) issues are caused by memory (by image objects) leakage, where inactive images cannot be garbage-collected effectively. Memory leakage is a major cause of `OutOfMemoryError` and has been extensively studied in the existing literatures[15, 16]. Issue 6 of Atarashii (shown in Fig.5) is an example of this type[35].

5.4 Answering RQ4: Are There Common Anti-Patterns for IID Issues?

Following the analysis of runtime behavior of IID issues in Subsection 5.3, we further inspect the statement sequences of concerned IID issues' annotated code slices to identify whether IID issues are related to any particular code anti-patterns. The overall results are summarized as follows.

Finding 4. "Certain anti-patterns are strongly correlated to IID issues: image decoding without resizing (23.1%), loop-based redundant image decoding (16.7%), image decoding in UI event handlers (8.3%), and unbounded image caching (3.2%). Together with additional bug types mentioned by existing studies[8, 14] (21.8%), 73.1% of the examined IID issues could be identified. This finding lays the foundation of our pattern-based lightweight static IID issue detection technique."

Image Decoding Without Resizing (*AP*1). IID issues are likely to present if an image potentially from external sources (like a network or a file system) is decoded with its original size.

Surprisingly, this simple anti-pattern covers 50/216 (23.1%) of all studied IID issues. Fig.8 gives such an example, in which displaying the thumbnail of a network image may unnecessarily consume about 128 MB of memory in decoding (using the image decoding API `decodeFile`() at line 3) and result in app crash. One developer later fixed this issue by resizing the image's resolution according to the actual size of the widget for displaying it (by invoking `createThumbnail`() for resizing images, lines 4–6).

Loop-Based Redundant Image Decoding (*AP*2). IID issues also frequently occur when an image is unintentionally decoded multiple times (e.g., in a loop). Particularly, Android apps often use a ViewGroup (e.g., a ListView, a GridView, or a RecyclerView) to display a scrolling list of images, and a ViewGroup is generally associated with some callback methods (e.g., for loading image resources) that can be frequently invoked.

This anti-pattern covers 36/216 (16.7%) of all studied IID issues. Fig.7 gives an example, in which the callback method `getView`() of `PodcastListAdapter` is frequently invoked (line 2) when a user browses a list of images and slides up and down. However, the cache option is miss-configured in `getView`(), as mentioned in Subsection 5.2.

Image Decoding in UI Event Handlers (*AP*3). Image decoding in the UI thread also contributes to a significant amount of studied IID issues, which are found to invoke (directly or indirectly) image decoding APIs in a UI event handler.

[35] https://github.com/wordpress-mobile/WordPress-Android/issues/5777, Mar. 2024.

This anti-pattern covers 18/216 (8.3%) of all studied IID issues. Fig.9 gives such an example, in which a big image read from a local location is decoded in the UI thread and causes the concerned app to run slowly (similar to the code snippet example of image decoding without resizing's in Fig.8). In this issue, methods `createThumbnail()` and `decodeByteArray()` are used to decode an image read from a URL site `mediaUri` (lines 6 and 8) in method `loadImage()`, which is invoked by a callback method `onCreate()`, which is then invoked in the UI thread. Therefore, the image decoding is actually done in the UI thread and causes UI lag. To fix this issue, one developer later moved the image decoding to a background thread (lines 7 and 10–13).

Unbounded Image Caching (AP4). Finally, an incorrectly implemented unbounded cache, in which a pool of decoded images is maintained but no image can be released, is another source of IID issues, since the ever-increasing cache size would cause memory bloat or `OutOfMemoryError`.

This anti-pattern covers 7/216 (3.2%) of all studied IID issues. Fig.5 gives such an example, in which an app crashes because of `OutOfMemoryError` after a user browses many images. The app's image cache `imageCache` is wrongly implemented in a way where it gathers all decoded images without releasing them. Its developer later fixed this issue by adding a soft reference for each image in the cache so that the cached images could be correctly released if necessary (line 16).

The four anti-patterns mentioned above are orthogonal, and they form a firm basis for developing effective static analysis techniques for detecting IID issues.

6 Design and Evaluation of Static IID Issue Detection Tool

In this section, we introduce the design and evaluation of our static anti-pattern-based prototype TAPIR for detecting IID issues.

6.1 Static Detection of IID Issues

6.1.1 IID Issue Anti-Pattern Rules

By further inspecting the empirical study results and IID issue cases, we observe that most IID issues are correlated with image decoding APIs concerning external images, which are essentially a small portion of all image decoding APIs. In particular, only the nine following Android official APIs are correlated with IID[36]: `decodeFile()`, `decodeFileDescriptor()`, `decodeStream()`, `decodeByteArray()`, `setImageURI()`, `decodeRegion()`, `createFromPath()`, `createFromStream()` and `setImageViewUri()`. Besides, we observe two popular third-party APIs, `ImageLoader.displayImage()` and `Glide.load()`, which are associated with at least two apps in the studied IID issues.

We call the eleven APIs "issue inducing APIs". IID issues can occur when these APIs are invoked under anti-pattern rules, which consist of API invocation sequences and/or parameter value combinations. These issue-inducing rules are characterized in Table 1, which are matched against in the TAPIR static analyzer. Specifically, compared with our conference version, four more issue-inducing APIs are added for rule #1 and two more issue-inducing APIs are added for rule #2.

6.1.2 TAPIR Static Analyzer

We implement the pattern-and-rule based static analyzer on top of Soot[37]. TAPIR takes an Android app binary (`apk`) file as input and uses `dex2jar`[38] to obtain the corresponding Java bytecode files. It then builds the app's context-insensitive call graph, with a few implicit method invocation relations being added, used to check rule #4. For example, `AsyncTask.execute()` implicitly invokes `AsyncTask.doInBackground()` defined in the same class; while `Thread.start()` method implicitly invokes `Thread.run()` defined in the same class.

For each potential issue-inducing API call site (CS), TAPIR obtains: 1) the data-flow of method parameters by a backward slicing, and 2) the usages of decoded image objects by a forward slicing. Then, TAPIR checks each image storage (IS) against the anti-pattern rules in Table 1 as follows.

1) Rule #1 (image decoding without resizing) is checked by analyzing the data-flow of the `Option` parameter, and a warning is raised if the `Option` parameter is missing or its value satisfies the condition specified in Table 1.

[36] `setImageURI()` and `setImageViewUri()` both decode and display an image.
[37] https://github.com/soot-oss/soot, Mar. 2024.
[38] https://sourceforge.net/projects/dex2jar/, Mar. 2024.

Table 1. Static IID Anti-Pattern Rules for IID Issue-Inducing APIs

#	Issue-Inducing API	Anti-Pattern Rule
1	`decode{File, FileDescriptor, Stream, ByteArray, Region}`, `setImage{URL, ViewUri}`★, `create{FromPath, FromStream}`★	("Image decoding without resizing") An external image is decoded with a `null` value of `BitmapFactory.Options`, or the fields in the option satisfy `inJustDecodeBounds` = 0 and `inSampleSize` ⩾ 1
2	`decode{File, FileDescriptor, Stream, ByteArray, Region}`, `create{FromPath, FromStream}`, `Glide.diskCacheStrategy`, `setImage{URL, ViewUri}`★	("Loop-based redundant image decoding") An external image is decoded (directly or indirectly) in `getView`, `onDraw`, `onBindViewHolder`, `getGroupView`, `getChildView`. However, if the developer explicitly stores decoded images in a cache (e.g., using `LruCache.put`), we do not consider this case as IID
3	`Glide.load`, `Glide.diskCacheStrategy`	("Loop-based redundant image decoding") An external image is decoded (directly or indirectly) in `getView`, `onDraw`, `onBindViewHolder`, `getGroupView`, `getChildView`. However, if the developer explicitly sets the argument of `Glide.diskCacheStrategy` to be `DiskCacheStrategy.ALL`, we do not consider this case as IID
4	`decode{File, FileDescriptor, Stream, ByteArray, Region}`, `create{FromPath, FromStream}`, `setImage{URL, ViewUri}`	("Image decoding in UI event handlers") An external image is decoded but is *not* invoked in an asynchronous method: overridden `Thread.run`, `AsyncTask.doInBackground`, or `IntentService.onHandleIntent`
5	`decode{File, FileDescriptor, Stream, ByteArray, Region}`, `create{FromPath, FromStream}`	("Unbounded image caching") An external image is decoded and added to an image cache by `LruCache.put()`, but there is no subsequent invocation to `LruCache.evictAll()` or `LruCache.remove()`
6	`ImageLoader.displayImage`	("Unbounded image caching") There exists method invocation to `ImageLoaderConfiguration.Builder.{memoryCache, diskCache}`, but there is no subsequent invocation to `clearMemoryCache` or `removeFromCache`
7	`Glide.load`	("Unbounded image caching") Caching images by `Glide.diskCacheStrategy` with `DiskCacheStrategy.`, `{SOURCE, RESULT, ALL}` but there is no subsequent invocation to `clearDiskCache`

Note: ★ denotes newly added issue-inducing APIs (compared with our conference version[9]).

2) Rules #2 and #3 (loop-based redundant image decoding) are equivalent to checking the call graph reachability from the loop-related method invocations to the CS. Furthermore, TAPIR also checks whether there is any data flow from the decoded image to cache-related methods or arguments (in particular, `LruCache.put()`, `DiskCacheStrategy.All`) to exclude non-IID cases.

3) Rule #4 (image decoding in UI event handlers) is another case of checking reachability from invocations of `Thread.run()`, `AsyncTask.doInBackground()`, or `IntentService.onHandleIntent()`, to the CS.

4) Rules #5, #6, and #7 (unbounded image caching) follow the same pattern of checking whether a series of designated method invocations are reachable in the call graph.

For each CS matching at least one anti-pattern rule, TAPIR generates an IID warning, which can be further validated by the respective app developer. Note that we currently focus on IID issues introduced by developers of the selected apps. So, TAPIR only analyzes the image displaying codes of the selected apps, skipping codes of image displaying third-party libraries, which are out of the apps' local source trees.

6.2 Evaluation of TAPIR Static Analyzer

In this subsection, we conduct experiments to investigate whether TAPIR can help developers fight with IID issues in real-world Android apps. The evaluation is driven by the following two research questions.

*RQ*5. *(Effectiveness and Efficiency): Can TAPIR efficiently and effectively identify IID issues in real-world Android applications?*

*RQ*6. *(Performance Impact and Improvement): What is the performance impact of the IID issues detected by TAPIR? How much performance can be improved if the IID issues detected by TAPIR are fixed?*

We conduct all experiments on a PC with an Intel® Core i7-6700 processor and 16 GB RAM.

6.2.1 Effectiveness and Efficiency of TAPIR

Validation Against Existing IID Issues. We collect the "buggy" `apks` corresponding to the IID issues in our previous studies. Specifically, for studied apps whose historical `apks` are available, we select directly the corresponding buggy `apks`. Meanwhile, for an app that no longer provides the corresponding historical buggy `apk`, we try to build these buggy `apks`

from the corresponding source code. In theory one should be able to compile each IID issue's corresponding app's source code. However, in practice, the dependencies of the concerned Android apps could not be easily resolved, and some large apps fail for compilation due to their stale dependencies. To reduce the possible bias that can be caused by our manual modifications to the apps' dependencies, we choose only those apps whose apks corresponding to the studied IID issues can be built from source code without suffering from any dependency issue. Finally, we collect buggy apks corresponding to 25 confirmed IID issues from ten Android apps (as shown in Table 2) as ground truth to evaluate TAPIR. As a comparison, we also apply IMGDroid[17] to the collected apks.

The overall evaluation results are shown in Table 3. All evaluated 25 IID issues belong to three anti-patterns. TAPIR correctly identifies all the 25 IID issues without any false negative (FN) report. At the same time, IMGDroid only successfully detects 13 (out of 14) issues of AP1 (i.e., image decoding without resizing) and one (out of two) issues of AP2 (i.e., loop-based redundant image decoding) but misses the other 11 issues (two of AP1, seven of AP2, and two of AP3).

We note that in practice TAPIR may possibly detect previously unknown IID issues in these app versions. However, we are unable to examine them in this part of the evaluation due to the lack of a ground truth of all IID issues in these apps' historical versions.

Discovering Previously Unknown IID Issues. In our conference version[9], we have applied TAPIR (without the new issue-inducing APIs noted in Table 1) to the latest versions (available at the end of September 2017) of all the 243 Android apps used in the empirical study and detected 45 previously unknown IID issues in 16 apps. In this paper, we also apply TAPIR (with the new issue-inducing APIs noted in Table 1) to both the old versions evaluated in [9] and the latest versions (available at the beginning of November 2021) of the 16 apps, and Table 4 shows their basic information.

This time, TAPIR reports totally 51 anti-pattern

Table 2. Effectiveness Validation Subjects

App Name	Category	Number of Downloads	Revision	KLOC
OpenNoteScanner	Education	10k+	d34135e	2.7
Subsonic	Multimedia	500k+	68496f6	23.8
PhotoAffix	Multimedia	10k+	3d8236e	1.4
WordPress	Internet	5M+	1a8fa65 8429f0a 9f87bc0 dcb7db1 8d3e9e6	95.8
OneBusAway	Navigation	500k+	9f6feea	15.7
Kontalk	Internet	10k+	3f2d89d 9185a80	19.6
NewPipe	Multimedia	10k+	4df4f68	3.5
MoneyManagerEx	Money	100k+	dcf4b87	63.8
BlueAlliance	Education	10k+	c081671	31.4
Collect	Tool	1M+	6b05133	52.0

Table 3. Effectiveness Validation Results

App Name	#IID (IRep/PR ID)	TAPIR						IMGDroid					
		AP1	AP2	AP3	AP4	TP	FN	AP1	AP2	AP3	AP4	TP	FN
OpenNoteScanner	2 (#12)	2	0	0	0	2	0	2	0	0	0	2	0
Subsonic	1 (#299)	0	1	0	0	1	0	0	0	0	0	0	1
WordPress	7 (#5290, #5777, #6267, #6676, #7057)	4	2	1	0	7	0	3	1	0	0	4	3
PhotoAffix	2 (#5)	2	0	0	0	2	0	2	0	0	0	2	0
Kontalk	3 (#234, #269, #789)	2	0	1	0	3	0	2	0	0	0	2	1
OneBusAway	2 (#730)	2	0	0	0	2	0	2	0	0	0	2	0
NewPipe	5 (#166)	0	5	0	0	5	0	0	0	0	0	0	5
MoneyManagerEx	1 (#938)	1	0	0	0	1	0	1	0	0	0	1	0
BlueAlliance	1 (#588)	1	0	0	0	1	0	1	0	0	0	1	0
Collect	1 (#2985)	1	0	0	0	1	0	0	0	0	0	0	1
Total	25	15	8	2	0	25	0	13	1	0	0	14	11

Note: Each known IID issue is either a true positive (TP) or a false negative (FN). Columns AP1–AP4 denote the number of studied IID issues categorized as a specific anti-pattern, respectively.

Table 4. List of 16 Android Apps Detected with Previously Unknown IID Issues by Applying TAPIR to the 243 Studied Apps

App Name	Category	Number of Downloads	Evaluated in [9]		Latest	
			Revision	KLOC	Revision	KLOC
Newsblur	Reading	50k+	535b879	20.1	ecdcacf	20.8
WordPress	Internet	10M+	30ff305	95.8	31ee0d2	262.0
Seadroid	Internet	100k+	f5993bd	37.9	45dad57	34.6
MPDroid	Multimedia	100k+	9b0a783	20.5	069baaa	20.6
Aphotomanager	Multimedia	10k+	9343d84	12.4	f7abd36	36.0
Conversations	Internet	100k+	1c31b96	38.0	3f31575	66.2
ownCloud	Internet	100k+	1443902	49.1	e2085ad	58.4
OpenNoteScanner	Education	10k+	2640785	3.5	8fff44c	3.8
Geopaparazzi	Navigation	10k+	71fd81e	89.9	078f90c	74.5
Passandroid	Reading	1M+	1382c6a	6.6	d671360	8.3
4pdaclient	Internet	1M+	a637156	41.9	26c6246	56.7
DocumentViewer	Reading	500k+	a97560f	49.6	c9bcd30	60.4
Kiss	Theming	100k+	9677dd1	5.1	8b8d8c2	13.5
Bubble	Reading	10k+	9f1e06c	3.5	3dbcd37	3.5
Qksms	Communication	500k+	c54c1cc	55.3	2bce012	57.0
Photoview	Demo	10k+	6c227ee	2.1	6c227ee	2.1

Note: An italic app name denotes it previously suffered from IID issues.

warnings (i.e., six more warnings than those reported in the conference version[9]) in the old versions (i.e., the versions evaluated in [9]) of the 16 apps. In general, it is hard to reproduce warnings reported by tools based on static analysis, and there is no guarantee that they could be triggered. Therefore, as done by most static-analysis-based studies, we do not attempt to reproduce these warnings as well. Instead, we manually inspect each of the reported warnings to determine whether it is an IID issue or not. Specifically, for each reported issue, we hypothetically execute the issue's associated Android app, and check whether any typical runtime behavior of IID issues (e.g., non-adaptive image decoding, repeated and redundant image decoding, UI-blocking image displaying, and image leakage as described in Subsection 5.3) could be found. If any typical runtime behavior is found, we categorize it as an IID issue (i.e., true positive, TP); otherwise, we categorize it as a spurious warning (i.e., false positive, FP). For each TP, we also report it to its developers for final validation. As most IID issues detected by TAPIR (in an anti-pattern way) are obvious and easy to fix, we do not attach respective patches or open pull requests. We let developers judge the validity of our reported issues on their own rather than potentially misleading them by trivial patches. The results are listed in Table 5.

Finally, 49 of 51 warnings are manually confirmed to be true instances of anti-patterns, achieving an anti-pattern discovery precision of 96.08%. For the FP case of Qkstms in which an image is decoded by `decodeByteArray()` without resizing, such an image is, however, not from an external source. TAPIR fails to analyze the `Options` parameter of `decodeByteArray()` which contains resized geometries, and thus conservatively reports it as an IID issue. The FP in ownCloud is also due to the limitation of static analysis: displayed images are from a disk cache, which stores already resized images.

Among the 49 unique IID issues, there are multiple issues of the same type (e.g., decoding without resizing) in the same app; at the same time, there are also multiple issues of different types within a single small code segment. For both of the cases, we believe that the involved issues are closely related, and we encapsulate them in a single report so as not to disturb the developers too much. Finally, we have enclosed the 43 issues reported by the conference version[39] into 20 issue reports and submitted them to respective developers (with descriptions of the issues and associated anti-patterns). The last column in Table 5 shows the reported IRep IDs. So far, we have received feedback from the developers on 27 issues. The remaining 16 reported IID issues are still pending (their concerned apps may no longer be under active maintenance).

Among the issues with feedback, 16/27 (59.3%) are confirmed as real performance threats, and 13 of

[39]We do not report the six issues detected with newly added issue-inducing APIs, as the corresponding reversions are too old.

Table 5. List of 49 Previously Unknown IID Issues Found in the 16 Android Apps (Old Versions)

App Name	AP1	AP2	AP3	AP4	Submitted Issue Reports
Newsblur	1	1	1	0	~~#977~~
WordPress	4	0	0	0	#5232, **#5703**partially-fixed/rejected
Seadroid	1	0	3	1	#616, #617, #766
MPDroid	1	0	0	0	**#837**
Aphotomanager	0	1	1	0	~~#74~~
Conversations	0	2	0	0	**#2198**fixed
ownCloud	3(1)	2	1	0	#1862
OpenNoteScanner	0	1	0	0	**#69**
Geopaparazzi	2	0	0	0	~~#387~~
Passandroid	3	0	0	0	~~#136~~
4pdaclient	0	1	1	0	**#25**fixed
DocumentViewer	0	1	2	0	#233
Kiss	0	0	1	0	**#570**fixed
Bubble	1	0	0	0	#47
Qksms	2(1)	2	2	0	**#718**fixed, **#719**fixed
Photoview	0	1	0	0	**#478**
Aphotomanager★	1	1	0	0	n/a
OpenNoteScanner★	0	1	0	0	n/a
Geopaparazzi★	1	0	0	0	n/a
Kiss★	1	0	0	0	n/a
Photoview★	1	0	0	0	n/a
Total	23(2)	14	12	1	

Note: ★ denotes issues detected with newly added issueinducing APIs and we do not report these six issues as the versions are too old. An italic app name denotes it previously suffered from IID issues. AP1–AP4 denote the number of detected issues related to each anti-pattern respectively. Numbers in a bracket are false positives. In the last column, blue-bold/red-strikethrough issues are explicitly confirmed/rejected by the developers, and the remaining ones are open issues. An superscript indicates whether a confirmed issue is "fixed" or "partially fixed and rejected". Results of IMGDroid are not available, as IMGDroid requires corresponding historical APKs that were not available for most APPs at the time the experiment was carried out.

the 16 IID issues (81.3%) have already been fixed by developers. This indicates that TAPIR can indeed detect quite a few new IID issues that affect the performance in real-world Android apps. The result also practically validates the effectiveness of the summarized anti-pattern rules in our empirical study.

For the remaining 11 IID issues, developers hold various conservative attitudes as discussed below.

1) Most developers rejecting our reports think that the performance impact might be negligible, and would only be convinced if we can provide further evidence about the performance degradation. For example, Aphotomanager's developers acknowledge that their app might encounter performance degradation in some cases but should be sufficiently fast and thus do not plan to fix them.

2) Some developers acknowledge the reported issues, but they claim to have higher-priority tasks than performance optimization.

We have also applied TAPIR (with the new issue-inducing APIs), IMGDroid[17], and PerferChecker[2], to the latest versions of the 16 apps (as shown in Table 4), and Table 6 shows the results[40]. TAPIR reports totally 67 anti-pattern warnings for the latest versions of the 16 apps. After manual checking, we finally confirm 64 warnings as true instances of anti-patterns and three false alarms. The three FPs come from the same app, Newsblur. In two FPs, an icon image is decoded by `decodeStream()` and `decodeFile()` without resizing. Such an image is, however, a small image prepared by the developer in advance. TAPIR fails to analyze the size of the icon (i.e., 128×128), and thus conservatively reports it as an IID issue. The third FP is also due to the limitation of static analysis, where displayed images are loaded (without resizing) from a disk cache, which stores already resized images. These three FPs are reported by IMGDroid as well.

Specifically, among the 64 manually confirmed warnings, there are 11 warnings that are also detected in the old versions (i.e., included in the 49 warnings shown in Table 5). Among the 11 identical warnings, there are nine warnings that have been reported to the developers (in our conference version) and two warnings that are not reported as they are recently detected by TAPIR with the newly added issue-inducing APIs. Of the nine reported warnings, four are still pending and three are rejected. The remaining two warnings (from the same app of Conversations) are interesting. The developer confirmed and fixed them (see issue #2283 of Conversations); however, TAPIR detects the same issues in the latest version of Conversations, indicating that the original fix does not solve the issues completely.

The other 37 warnings detected in the old versions are not detected by TAPIR in the latest versions. The reasons for the disappearances of these warnings might be two-fold. 1) They might be fixed intentionally by the developers (e.g., the 13 fixed issues with explicit responses as shown in Table 5). 2) They might be removed unintentionally by the developers during the normal software evolution process as

[40]We also plan to evaluate another tool, namely DRAW proposed by Gao *et al.*[8]. However, it is not public available currently.

Table 6. List of Previously Unknown IID Issues Found in the 16 Android Apps (Latest Versions)

App Name	Detected IID Issues																	Execution Time (s)		
	AP1			AP2			AP3				AP4			Total				TAPIR	IMGDroid	PerferChecker
	TI	T	I	TI	T	I	TI	T	I	P	TI	T	I	TI	T	I	P			
Newsblur	1(3)	0	0	0	1	0	4	1	2	0	0	0	0	8	2	2	0	44.6	11.7	3.8
WordPress★	0	0	1	0	0	0	0	0	0	–	0	0	0	0	0	1	–	89.8	84.2	n/a
Seadroid	0	1	0	0	0	0	2	0	11	0	0	1	0	2	2	11	0	93.8	47.4	1.5
MPDroid	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	6.7	6.6	2.5
Aphotomanager	1	0	1	1	1	0	2	0	8	0	0	0	0	4	1	9	0	32.1	10.6	12.0
Conversations	0	2	1	0	0	1	12	0	7	0	0	0	0	12	2	9	0	124.0	30.6	7.6
Owncloud★	0	0	0	0	0	5	0	0	0	–	0	0	0	0	0	5	–	134.3	44.2	n/a
OpenNoteScanner	0	0	0	0	0	0	1	0	2	0	0	0	0	1	0	2	0	145.7	93.8	5.4
Geopaparazzi	0	8	2	0	0	0	1	8	3	0	0	0	0	1	16	5	0	157.9	54.2	8.9
Passandroid	3	0	3	0	0	0	3	0	4	0	0	0	0	6	0	7	0	46.0	14.7	4.0
4pdaclient	0	0	1	0	0	0	0	0	8	0	0	0	0	0	0	9	0	98.4	33.6	6.0
DocumentViewer	1	0	1	0	0	1	1	1	0	0	0	0	0	2	1	2	0	29.8	50.4	4.2
Kiss★	1	1	1	0	1	0	1	0	1	–	0	0	0	2	2	2	–	9.7	3.5	n/a
Bubble	0	0	3	0	0	0	0	0	3	0	0	0	0	0	0	6	0	32.3	12.0	3.0
Qksms★	0	0	0	0	0	0	0	1	0	–	0	0	0	0	1	0	–	111.0	27.4	n/a
Photoview	0	1	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	7.5	2.5	0.9
Total	7(3)	13	15	1	3	7	28	11	49	0	0	1	0	39	28	71	0	–	–	–

Note: PerferChecker crashes with errors during processing the apps annotated with ★. An italic app name denotes it previously suffered from IID issues. TI: detected by both TAPIR and IMGDroid, T: uniquely detected by TAPIR, I: uniquely detected by IMGDroid, P: uniquely detected by PerferChecker. AP1–AP4 denote the number of detected issues related to each anti-pattern respectively. Numbers in a bracket are false positives (currently, only issues reported by TAPIR are manually checked).

well (e.g., related codes might be updated or removed).

For the remaining new 53 warnings reported by TAPIR in the latest app versions, by May 2022, we have enclosed 22 warnings in ten issue reports and submitted them to respective developers. So far we have received four responses involving 12 warnings. The first issue report[41] involves seven AP1 (i.e., decoding without resizing) warnings in Geopaparazzi, which is a tool providing georeferenced notes and pictures to do engineering/geologic surveys. The developer admits that "an image will be read in at its original size" but "this must be passed on the map renderer in its original size, which then resizes the combined, individual, images as needed", "otherwise disaster is guaranteed". The second issue report[42] involves one AP3 (i.e., decoding in UI event handlers) warning in NewsBlur. The developer admits that the favicon image is "decoded on the main thread without caching" but "the impact to the end user would be very low if noticeable on current hardware", and assigns a low priority to it. The third issue report[43] involves two AP1 warnings in KISS. One developer responds that he would "have a look at the code to see if it could be a problem". The developer of Conversations rejects the last issue report[44] involving two AP1 warnings.

As shown in Table 6, among the 67 warnings reported by TAPIR, there are totally 39 warnings (including the three false alarms) that can also be detected by IMGDroid. TAPIR uniquely reports 28 warnings, while IMGDroid uniquely reports 71 warnings of AP1, AP2, and AP3[45]. The reasons for this difference can be attributed to the following aspects. 1) Different issue-inducing APIs. For example, IMGDroid includes `decodeResource`() as its issue-inducing API for detecting IID issues. `decodeResource`() is usually used by an app to load images from the `drawable` directory, which usually consists of small images for different densities. Decoding such small images would not introduce too much overhead and further affect user experience in most cases, therefore we

[41] https://github.com/geopaparazzi/geopaparazzi/issues/661, Mar. 2024.
[42] https://github.com/samuelclay/NewsBlur/issues/1573, Mar. 2024.
[43] https://github.com/Neamar/KISS/issues/1838, Mar. 2024.
[44] https://github.com/iNPUTmice/Conversations/issues/4236, Mar. 2024.
[45] IMGDroid also reports one warning of the anti-pattern "passing image via intent", which is out of the scope of TAPIR.

do not include `decodeResource()` as an issue-inducing API in TAPIR to avoid reporting warnings that do not affect user experience. Besides, IMGDroid includes decoding APIs of two more third-party libraries (i.e., Picasso and Fresco) as its issue-inducing APIs, which are currently not included (but can be easily added) in TAPIR. 2) Different criteria. For example, for situations like those shown in Fig.10, where `getView()` is invoked in the UI thread, IMGDroid would report two "decoding in UI thread" issues for the two `decodeFile()` invocations (lines 12 and 16). Whereas we believe the first invocation (line 12) would not bring in much overhead, as the option `inJustDecodeBounds` is set to `true` (line 11), TAPIR reports only one issue for the second invocation of `decodeFile()` (line 16). 3) Different implementation details. For example, although both tools are built on top of soot, different options adopted by them as well as the ways they handle inter-procedural calls may lead to different results.

The tool of PerfChecker[2] could possibly identify one kind of IID issue: decoding bitmap (i.e., calling `decodeFile()`) in the UI thread, which is a special case of AP3. Specifically, to analyze an app, PerfChecker needs the bytecode of the app as well as the `.jar` files of (usually tens of) dependent libraries (with specific version requirements), and it requires a lot of non-trivial configuration for each subject app. In our evaluation, we successfully apply it to only 12 of the studied apps without detecting any IID issue. While for the remaining four apps, PerfChecker crashes with errors without generating any IID issue report as well.

Table 6 also shows the time consumption of TAPIR, IMGDroid, and PerferChecker on each evaluated app. It takes only a few seconds to a few minutes for both TAPIR and IMGDroid to analyze an app, while PerfChecker could analyze an app within ten seconds.

Here, we present two interesting real-world IID issue cases to show the effectiveness of TAPIR when applying it in practice[9]. We could see how developers have overlooked the severity of our reported IID issues, and in fact, seemingly minor IID issues can indeed cause poor app experience.

WordPress. The first case is from WordPress, which is one of the most popular blogging apps. TAPIR identifies two anti-pattern instances of image decoding without resizing, and thus one issue report is composed. However, the app's developers did not realize the severity of our reported issue, and assigned a low priority to it.

Two months later, a user reported an image-related bug that WordPress crashed when loading a large image. The developers then made extensive efforts in diagnosing this issue, and proposed several fixes. However, twenty days later, another user encountered a similar problem with the same triggering condition. The developers once again attempted to diagnose its root cause, but did not reach a clear verdict[47].

For this interesting case, we apply TAPIR to the latest reversion (i.e., `c94b1b5`) of WordPress in 2017 and detected one previously detected and two new

```
1   public View getView(int position,…) {
2     ...
3     final File file = new File(…);
4     Bitmap bitmap = HugeImageLoader.loadImage(file, …);
5     holder.image.setImageBitmap(bitmap);
6      ...
7   }
8
9   public static Bitmap loadImage(File file,…){
10    BitmapFactory.Options options = new BitmapFactory.Options();
11    options.inJustDecodeBounds = true;
12    BitmapFactory.decodeFile(file.getAbsolutePath(), options);
13    int downscale = calculateInSampleSize(options, …);
14    options.inSampleSize = downscale;
15    options.inJustDecodeBounds = false;
16    Bitmap b = BitmapFactory.decodeFile(file.getAbsolutePath(),
                options);
17    return b;
18  }
```

Fig.10. Decoding in UI thread in issue 74 of Aphotomanager[46].

[46]https://github.com/k3b/APhotoManager/issues/74, Mar. 2024.
[47]https://github.com/wordpress-mobile/WordPress-Android/issues/5701, Mar. 2024.

IID issues, which all belong to the anti-pattern of "image decoding without resizing". We reported all three issues and the developers quickly fixed two of them in three days[48]. After fixing these TAPIR's reported issues, similar image-related performance issues have never been reported again since July 2017 until the day this paper was written.

This case suggests that providing consequence verification may make developers more active in dealing with our reported IID issues. In addition, IID issues can be more complicated than one expects. Developers may have overlooked the actual difficulty of diagnosing such issues, and ad-hoc fixings may not be efficient in addressing IID issues.

KISS. The second case is from KISS, an Android app launcher with searching functionalities, the consequences of whose suffered IID issue might have also been overlooked by its developers. TAPIR detects the anti-pattern of "loop-based redundant image decoding" in KISS, and thus we reported this issue to its developers[49]. Unfortunately, the developers explicitly rejected our proposal due to the concern that they believed that the performance impact would be minor and KISS should be kept simple and lightweight.

Interestingly, a year and a half later, one of the KISS users encountered and complained about an image displaying problem[50]. Then the developers noticed this and decided that this is truly due to our mentioned IID issue. Thus they quickly fixed this issue. This encouraging result suggests that pattern-based program analysis can be naturally effective for defending against practical IID issues in Android apps.

6.2.2 *Performance Impact and Improvement Study*

We also conduct an experiment to answer RQ6 about the performance impact of the IID issues detected by TAPIR and the performance improvement after fixing these detected IID issues.

Experimental Setup. We conduct our experiment on a set of IID issues selected from the previously unknown IID issues detected by TAPIR. These detected IID issues belong to four IID issue anti-patterns and we randomly select no more than two IID issues per each anti-pattern as experimental subjects. To measure runtime performance of an IID issue's corresponding buggy code, we compile the source code of the app revision corresponding to the selected IID issue, generate the `apk` file, and install `apk` file on a Xiaomi Redmi Note4 smartphone running Android 6.0, which is a very popular Android device with 26.8 million active devices in 2017 (the time when these selected IID issues were detected by TAPIR)[51]. We use this Android device for the reason that Android developers commonly use current popular mobile devices for testing when they release their apps, which can measure the actual impact of the IID issues on end-users.

Next, we carefully study IID issues' corresponding apps and design one test case for each IID issue. For these designed test cases, their execution should cover the buggy code of these selected IID issues. Finally, six IID issues are selected and Table 7 lists their basic information, including 1) the IID issue index, 2) the anti-pattern type of an IID issue, 3) the name of an IID issue's corresponding Android app, 4) the buggy class, 5) the buggy revision in which TAPIR detected the IID issue, 6) the issue report of the IID issue. The designed test cases of these IID issues are public available[52]. It is not an easy work to design test cases that can cover specific code in an Android app[18], and we have not found a test case that covers the buggy code of any IID issue belonging to AP4 in our selected subjects.

Table 7. Selected IID Issues for Performance Analysis

Index	IID Pattern	App Name	Class	Revision	Issue Report
1	AP1	Qksms	Contact	`c54c1cc`	**#719**[fixed]
2	AP1	Newsblur	ImageLoader	`535b879`	~~977~~
3	AP2	Qksms	ContactHelper	`c54c1cc`	**#718**[fixed]
4	AP2	Qksms	Contact	`c54c1cc`	**#719**[fixed]
5	AP3	Qksms	Contact	`c54c1cc`	**#718**[fixed]
6	AP3	Newsblur	PrefsUtils	`535b879`	~~977~~

Note: In the last column, bold/stroke-out issues are explicitly fixed/rejected by the developers.

[48]Developers consider one report as false positive because they have control of the external image size.
[49]https://github.com/Neamar/KISS/issues/570, Mar. 2024.
[50]https://github.com/Neamar/KISS/issues/1054, Mar. 2024.
[51]https://newzoo.com/insights/articles/insights-into-the-2-3-billion-android-smartphones-in-use-around-the-world/, Mar. 2024.
[52]https://github.com/StruggleLi/Test-cases, Mar. 2024.

As all the selected IID issues would cause performance degradation because of improper implementation of image decoding (such as decoding an image directly without considering the actual size of the widget that displays the image, decoding large images in the UI thread), for each selected IID issue, we manually instrument the source code of the corresponding app revision to obtain the time consumption used for image decoding in the buggy code (i.e., the execution time of the statements responding for image decoding). After these preparations, for each selected IID issue, we then run the corresponding instrumented app with the designed test case 100 times to measure the average time consumption. For each test run, we reset the app under test by reinstalling the app.

To measure the performance improvement of the selected IID issues after fixing them, we prepare an "optimized version" for each IID issue. The optimized version is the app revision that the IID issue has been fixed. If an IID issue has been fixed by developers, we replace the buggy version's buggy code with that of the bug fixing revision[53]. If there is no fix provided by developers for an IID issue, we manually fix it by referring to the fix suggestions provided in the Android documentation[54].

• For the IID issues belonging to AP1, we fix them by resizing the resolution of the displayed images according to the size of the widgets used for displaying them, so as to fit the widget's size and reduce unnecessary image decoding.

• For the IID issues belonging to AP2, we fix them by adding an image cache for the displayed images, so as to reduce redundant image decoding.

• For the IID issues belonging to AP3, we fix them by moving image decoding to background threads, so as to avoid UI thread blocking.

Next, as we do in the performance impact experiment, we use the same designed test cases and execution strategy to execute the optimized version of each IID issue and record the time consumption, which is then compared with that of the buggy version.

Experimental Results. In order to keep an app's user interface (UI) smooth, developers need to make sure that the Android system can render the UI at a frame rate of 60 FPS or above[55]. To achieve this target, an app typically needs to be able to prepare the items in the UI in a couple of milliseconds. Therefore, even one millisecond saved in decoding an image is helpful to guarantee satisfactory user experience. Now, let us discuss the results of our experiments.

• IID Issues of AP1. For the IID issue 1 in Qksms, it involves a UI of the contact list and each contact item is attached with a contact image. The displayed contact images all have a fixed resolution of 720×720. When displaying these images in the buggy version, the average decoding time of an image consumes 15.2 milliseconds. However, the required image resolution of the widgets used to display contact images is 140×140. In the optimized version, where proper down-sampling is implemented, the average time consumption for image decoding reduces to 3.8 milliseconds. For the IID issue 2 in Newsblur, it involves a UI of the news list and each news item is attached with a widget to display a thumbnail image. The required image resolution of the widgets is 240×40. However, in the buggy version, the resolution of the thumbnail images actually displayed in these widgets ranges from 432×462 to 1 534×2 560, and the time consumption for decoding an image ranges from 10.1 milliseconds to 132.6 milliseconds, which decreases to 4.6 milliseconds in the optimized version.

• IID Issues of AP2. IID issues 3 and 4 involve repeated displaying and redundant decoding contact images with resolution of 720×720 and the average image decoding time is 15.2 milliseconds and 13.6 milliseconds, respectively. In the optimized versions, a displayed image only needs to be decoded on the first display, and the decoded image object would be cached for future use, leading to a time consumption of only about 0.000 3 milliseconds (i.e., the average time for getting an image object from cache) for both the two apps.

• IID Issues of AP3. IID issue 5 in Qksms involves displaying seven contacts' images with a resolution of 720×720 in a contact list in the UI thread, and the buggy version takes 157.2 milliseconds on average to decode all these seven images. IID issue 6 in Newsblur involves displaying a user thumbnail with a resolution of 180×180 in the UI thread, which takes about 3.5 milliseconds in the buggy version. In the optimized versions, where the image decoding logic is

[53]To ensure functional equivalence, we do not use developers' own issue fixing revision of some IID issue as their optimized version because there commonly exist additional revisions between the buggy version and the issue fixing version and those additional revisions may introduce functional changes. What is more, the code changes in those revisions may have side effect on the performance of the Android application.

[54]https://stuff.mit.edu/afs/sipb/project/android/docs/training/displaying-bitmaps/index.html, Mar. 2024.

[55]https://developer.android.com/, Mar. 2024.

moved to a background thread, the time consumption (in the background thread) changes to 103.6 milliseconds and 4.1 milliseconds, respectively. Although the improvements in time consumption for decoding images are not very significant or even negative, it improves the smoothness of UI thread execution by reducing the workload of the UI thread.

From the experiment, we find that decoding a relatively large image (e.g., $1\,534\times2\,560$) or a relatively small number of images (e.g., several 720×720 images) can take more than 100 milliseconds (on the evaluated device). In addition, once an IID issue is resolved, the performance can be significantly improved in most cases. The finding shows the effectiveness of TAPIR as the IID issues detected by it indeed cause significant performance degradation.

7 Threats to Validity

Subject Selection. The empirical study is based on 216 IID issues from 243 open-source Android apps, which may not be fully representative of all IID issues in practice. Therefore, the generalizability of the anti-patterns and the associated rules for identifying IID issues is not guaranteed. To reduce such threats, we collect these IID issues from well-maintained popular open-source Android apps covering diverse categories. Besides, we determine the issue-inducing APIs in each rule shown in Table 1 not only directly based on APIs encountered in studied issues, but based on our experience/knowledge of the image displaying process as well (i.e., for each rule, there would be some APIs that have never been found in the studied issues). In the future, we plan to collect more IID issues in more Android apps to study their characteristics.

Another threat is that we only evaluate TAPIR with a small number of apps. Although we have tried our best, we only successfully build a relatively small ground truth with ten apps involving 25 known IID issues. At the same time, we only apply TAPIR to the studied apps for detecting previous unknown IID issues. As a result, we might not guarantee the generalizability of our evaluations. In the future, we plan to evaluate TAPIR on more Android apps.

Code Slice Extraction. The IID issues' annotated code slices used in our study are extracted manually, and it is possible that such a process is subject to mistakes. To reduce the threat, we cross-validate all results. We also release our datasets for public access.

Limitations of TAPIR. TAPIR is lightweight (e.g., lacking the full path sensitivity) and identifies only the extracted code anti-patterns. Therefore, it may report spurious warnings (false positives) or miss certain anti-patterns (false negatives). We intentionally design TAPIR to be simple, and the evaluation demonstrates its effectiveness in detecting IID issues. A future effort is to develop more sophisticated static and/or dynamic analysis for more accurate detection of IID problems.

Besides, as mentioned earlier, TAPIR currently does not consider the source code of third-party libraries used by studied Android apps, which could be another source of IID issues. Developers may also use ad-hoc implementations for image displaying, presenting obstacles to our pattern-based analysis. This aspect of IID issue detection can be a potential future direction.

8 Related Work

Performance has become a major concern for mobile app developers and has been widely studied in the community. In this section, we briefly summarize and discuss existing literatures on this concern.

Understanding Performance Issues in Mobile Apps. Huang *et al.*[19] identified several important factors that may impact user-perceived network latencies in mobile apps. Liu *et al.*[2] studied the characteristics of Android app performance issues and identified their common patterns. These findings can support performance issue avoidance, testing, debugging, and analysis for Android apps. Nejati and Balasubramanian[20] performed an in-depth investigation of mobile browser performance by pairwise comparisons between mobile and non-mobile browsers. Huang *et al.*[21] conducted a systematic measurement study to quantify user-perceived latency with/without background workloads. Rosen *et al.*[22] investigated the benefits and challenges of using Server Push on mobile devices to improve mobile performance.

Several studies provide some clues for understanding and detecting IID issues as studied in this work. Wang and Rountev[4] provided evidence that the response time of image decoding can grow significantly as the image's size increases, and thus IID may be a significant source of performance issues, while Carette *et al.*[3] discussed that large images may potentially impact the performance of Android apps.

These studies either focus on general performance issues in Android apps and thus provide limited insights to tackle specific IID issues, or do not systematically investigate IID issues in practical Android

apps. To the best of our knowledge, in [9], we conducted the first systematic empirical study of IID issues using real-world Android apps, and in this paper we extend that study with 54 new IID issues. Our studies provide key insights (e.g., common anti-patterns derived from real-world issues and patches) on understanding and detection of IID issues in Android apps.

Diagnosing and Detecting Performance Issues in Mobile Apps. Mantis[7] estimates the execution time for Android apps on given inputs to identify problem-inducing inputs that can slow down an app's execution. ARO[23] monitors cross-layer interactions (e.g., those between the app layer and the resource management layer) to help disclose inefficient resource usage, which can commonly cause performance degradation of Android apps. AppInsight[24] instruments app binaries to identify critical paths (e.g., slow execution paths) in handling user interaction requests, so as to disclose root causes for performance issues in mobile apps. Panappticon[25] monitors the application, system, and kernel software layers to identify performance problems stemming from application design flaws, underpowered hardware, and harmful interactions between apparently unrelated applications, and further reveals performance issues from inefficient platform code or problematic app interactions. Nistor and Ravindranath[26] analyzed sequences of calls to String getter methods to understand the impact of larger inputs on a user's perception in Windows Phone apps. Lin *et al.*[27] proposed an approach, ASYNCHRONIZER, to automatically refactor long-running operations into asynchronous tasks. Kang *et al.*[28] tracked asynchronous executions with a dynamic instrumentation approach and profiled them in a task granularity, equipping it with low-overhead and high compatibility merits.

For the work on diagnosing and detecting IID issues, Liu *et al.*[2] proposed a tool, PerfChecker, based on static analysis, which can possibly identify one kind of IID issues: decoding bitmap (i.e., calling `decodeFile()`) in the UI thread. Draw proposed by Gao *et al.*[8] performs two UI rendering analyses to help app developers pinpoint rendering problems and resolve short delays. However, these pieces of work can cover only a small proportion of IID issues studied in this paper. Song *et al.*[17] proposed a static analysis tool, IMGDroid, that can detect the IID issues of anti-patterns 1–3. However, the scopes of these two tools are not exactly the same. TAPIR focuses on the image displaying process that starts from image decoding and ends with image rendering, whereas IMGDroid focuses on image loading process. Specifically, TAPIR can additionally detect the anti-pattern of "unbounded image caching" while IMGDroid can additionally detect the anti-patterns of "image passing by intent" and "local image loading without permission".

Fixing and Optimizing Performance Issues in Mobile Apps. Lee *et al.*[29] proposed a technique that can render speculative frames of future possible outcomes, delivering them to the client device entire RTT ahead of time, and recover quickly from possible mis-speculations when they occur to mask up the network latency. Huang *et al.*[21] developed a lightweight tracker to accurately identify all delay-critical threads that contribute to the slow response of user interactions, and build a resource manager that can efficiently schedule various system resources including CPU, I/O, and GPU, for optimizing the performance of these threads. Zhao *et al.*[1] leveraged the string analysis and callback control flow analysis to identify HTTP requests that should be prefetched to reduce the network latency in Android apps. Lyu *et al.*[30] rewrote the code that places database writes within loops to reduce the energy consumption and improve runtime performance of database operations in Android apps. Nguyen *et al.*[31] reduced the application delay by prioritizing reads over writes and grouping them based on assigned priorities. In our work, the detection results of TAPIR provide the location and anti-patterns of its detected IID issues in Android apps, which can then be used to help developers quickly fix IID issues as our experiments and case analyses show.

9 Conclusions

Improper handling of (potentially large) images in an Android app can lead to inefficient image display (IID) issues, which can cause the app to crash or slow down. In this paper, we proposed a descriptive framework for the image displaying procedures of IID issues. Based on the framework, we conducted an empirical study of 216 real-world IID issues and found that most IID issues are strongly correlated with certain anti-patterns in the source code (e.g., image decoding without resizing, loop-based redundant image decoding, image decoding in UI event handlers, and unbounded image caching). We proposed a static tool TAPIR based on such anti-patterns. The evaluation results show that the anti-pattern-based static technique can effectively and efficiently detect IID issues in practice.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Zhao Y X, Laser M S, Lyu Y J, Medvidovic N. Leveraging program analysis to reduce user-perceived latency in mobile applications. In *Proc. the 40th International Conference on Software Engineering*, May 27–Jun. 3, 2018, pp.176–186. DOI: 10.1145/3180155.3180249.

[2] Liu Y P, Xu C, Cheung S C. Characterizing and detecting performance bugs for smartphone applications. In *Proc. the 36th Int. Conf. Software Engineering*, May 31–Jun. 7, 2014, pp.1013–1024. DOI: 10.1145/2568225.2568229.

[3] Carette A, Younes M A A, Hecht G, Moha N, Rouvoy R. Investigating the energy impact of Android smells. In *Proc. the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering*, Feb. 2017, pp.115–126. DOI: 10.1109/SANER.2017.7884614.

[4] Wang Y, Rountev A. Profiling the responsiveness of Android applications via automated resource amplification. In *Proc. the 2016 International Conference on Mobile Software Engineering and Systems*, May 2016, pp.48–58. DOI: 10.1145/2897073.2897097.

[5] Linares-Vásquez M, Vendome C, Luo Q, Poshyvanyk D. How developers detect and fix performance bottlenecks in Android apps. In *Proc. the 2015 IEEE Int. Conf. Software Maintenance and Evolution*, Sept. 29–Oct. 1, 2015, pp.352–361. DOI: 10.1109/ICSM.2015.7332486.

[6] Liu Y P, Xu C, Cheung S C. Diagnosing energy efficiency and performance for mobile internetware applications. *IEEE Software*, 2015, 32(1): 67–75. DOI: 10.1109/MS.2015.4.

[7] Kwon Y, Lee S, Yi H, Kwon D, Yang S, Chun B G, Huang L, Maniatis P, Naik M, Paek Y. Mantis: Automatic performance prediction for smartphone applications. In *Proc. the 2013 USENIX Conference on Annual Technical Conference*, Jun. 2013, pp.297–308.

[8] Gao Y, Luo Y, Chen D Q, Huang H C, Dong W Y, Xia M Y, Liu X, Bu J J. Every pixel counts: Fine-grained UI rendering analysis for mobile applications. In *Proc. the 2017 IEEE Conference on Computer Communications*, May 2017. DOI: 10.1109/INFOCOM.2017.8057023.

[9] Li W J, Jiang Y Y, Xu C, Liu Y P, Ma X X, Lyu J. Characterizing and detecting inefficient image displaying issues in Android apps. In *Proc. the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*, Feb. 2019, pp.355–365. DOI: 10.1109/SANER.2019.8668030.

[10] Agrawal H, Horgan J R. Dynamic program slicing. *ACM SIGPLAN Notices*, 1990, 25(6): 246–256. DOI: 10.1145/93548.93576.

[11] Liu Y P, Xu C, Cheung S C, Terragni V. Understanding and detecting wake lock misuses for Android applications. In *Proc. the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Nov. 2016, pp.396–409. DOI: 10.1145/2950290.2950297.

[12] Hu J J, Wei L L, Liu Y P, Cheung S C, Huang H X. A tale of two cities: How WebView induces bugs to Android applications. In *Proc. the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Sept. 2018, pp.702–713. DOI: 10.1145/3238147.3238180.

[13] Rath M, Rendall J, Guo J L C, Cleland-Huang J, Mäder P. Traceability in the wild: Automatically augmenting incomplete trace links. In *Proc. the 40th International Conference on Software Engineering*, May 27–Jun. 3 2018, pp.834–845. DOI: 10.1145/3180155.3180207.

[14] Wu T Y, Liu J R, Xu Z B, Guo C R, Zhang Y L, Yan J, Zhang J. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. *IEEE Trans. Software Engineering*, 2016, 42(11): 1054–1076. DOI: 10.1109/TSE.2016.2547385.

[15] Xu G Q, Rountev A. Precise memory leak detection for Java software using container profiling. In *Proc. the 30th International Conference on Software Engineering*, May 2008, pp.151–160. DOI: 10.1145/1368088.1368110.

[16] Yan D C, Xu G Q, Yang S Q, Rountev A. LeakChecker: Practical static memory leak detection for managed languages. In *Proc. the 2014 Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Feb. 2014, pp.87–97. DOI: 10.1145/2544137.2544151.

[17] Song W, Han M Q, Huang J. IMGDroid: Detecting image loading defects in Android applications. In *Proc. the 43rd IEEE/ACM Int. Conf. Software Engineering*, May 2021, pp.823–834. DOI: 10.1109/ICSE43902.2021.00080.

[18] Jensen C S, Prasad M R, Møller A. Automated testing with targeted event sequence generation. In *Proc. the 2013 Inter. Symp. Software Testing and Analysis*, Jul. 2013, pp.67–77. DOI: 10.1145/2483760.2483777.

[19] Huang J X, Xu Q, Tiwana B, Mao Z M, Zhang M, Bahl P. Anatomizing application performance differences on smartphones. In *Proc. the 8th International Conference on Mobile Systems, Applications, and Services*, Jun. 2010, pp.165–178. DOI: 10.1145/1814433.1814452.

[20] Nejati J, Balasubramanian A. An in-depth study of mobile browser performance. In *Proc. the 25th International Conference on World Wide Web*, Apr. 2016, pp.1305–1315. DOI: 10.1145/2872427.2883014.

[21] Huang G, Xu M W, Lin F X, Liu Y X, Ma Y, Pushp S, Liu X Z. ShuffleDog: Characterizing and adapting user-perceived latency of Android apps. *IEEE Trans. Mobile Computing*, 2017, 16(10): 2913–2926. DOI: 10.1109/TMC.2017.2651823.

[22] Rosen S, Han B, Hao S, Mao Z M, Qian F. Push or request: An investigation of HTTP/2 server push for improving mobile performance. In *Proc. the 26th International Conference on World Wide Web*, Apr. 2017, pp.459–468. DOI: 10.1145/3038912.3052574.

[23] Qian F, Wang Z G, Gerber A, Mao Z Q, Sen S, Spatscheck O. Profiling resource usage for mobile applications: A cross-layer approach. In *Proc. the 9th International Conference on Mobile Systems, Applications, and Services*, Jun. 28–Jul. 1, 2011, pp.321–334. DOI: 10.1145/1999995.2000026.

[24] Ravindranath L, Padhye J, Agarwal S, Mahajan R, Obermiller I, Shayandeh S. AppInsight: Mobile app performance monitoring in the wild. In *Proc. the 10th USENIX Conference on Operating Systems Design and Implementation*, Oct. 2012, pp.107–120. DOI: 10.5555/2387880.2387891.

[25] Zhang L D, Bild D R, Dick R P, Mao Z M, Dinda P. Panappticon: Event-based tracing to measure mobile application and platform performance. In *Proc. the 2013 International Conference on Hardware/Software Codesign and System Synthesis*, Sept. 29–Oct. 4, 2013, pp.1–10. DOI: 10.1109/CODES-ISSS.2013.6659020.

[26] Nistor A, Ravindranath L. SunCat: Helping developers understand and predict performance problems in smartphone applications. In *Proc. the 2014 International Symposium on Software Testing and Analysis*, Jul. 2014, pp.282–292. DOI: 10.1145/2610384.2610410.

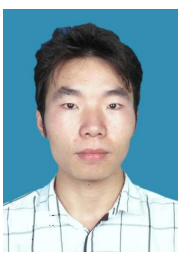
[27] Lin Y, RadoiC, Dig D. Retrofitting concurrency for Android applications through refactoring. In *Proc. the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Nov. 2014, pp.341–352. DOI: 10.1145/2635868.2635903.

[28] Kang Y, Zhou Y F, Xu H, Lyu M R. DiagDroid: Android performance diagnosis via anatomizing asynchronous executions. In *Proc. the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Nov. 2016, pp.410–421. DOI: 10.1145/2950290.2950316.

[29] Lee K, Chu D, Cuervo E, Kopf J, Degtyarev Y, Grizan S, Wolman A, Flinn J. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proc. the 13th Annual International Conference on Mobile Systems, Applications, and Services*, May 2015, pp.151–165. DOI: 10.1145/2742647.2742656.

[30] Lyu Y J, Li D, Halfond W G J. Remove RATs from your code: Automated optimization of resource inefficient database writes for mobile applications. In *Proc. the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Jul. 2018, pp.310–321. DOI: 10.1145/3213846.3213865.

[31] Nguyen D T, Zhou G, Xing G L, Qi X, Hao Z J, Peng G, Yang Q. Reducing smartphone application delay through read/write isolation. In *Proc. the 13th Annual Int. Conf. Mobile Systems, Applications, and Services*, May 2015, pp.287–300. DOI: 10.1145/2742647.2742661.



Wen-Jie Li is a lecturer with the School of Computer and Information, Anhui Normal University, Wuhu. He received his Ph.D. degree from Nanjing University, Nanjing, in 2022. His research interests include Android application analysis and testing.

Jun Ma is an associate professor in the Department of Computer Science and Technology at Nanjing University, Nanjing. He received his Ph.D. degree from Nanjing University, Nanjing, in 2015. His research interests include software testing and analysis, and software composition analysis.

Yan-Yan Jiang received his Ph.D. degree from Nanjing University, Nanjing, in 2017. He is currently an associate professor in the Department of Computer Science and Technology, Nanjing University, Nanjing. His research interests include software testing/analysis and program synthesis.

Chang Xu is a professor in the Department of Computer Science and Technology at Nanjing University, Nanjing. He received his Ph.D. degree in computer science and engineering from The Hong Kong University of Science and Technology, Hong Kong, in 2008. His research interests include big data software engineering, software testing and analysis, and adaptive and embedded system.

Xiao-Xing Ma received his Ph.D. degree in computer science and technology from Nanjing University, Nanjing, in 2003. He is currently a professor in State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing. His research topics include adaptive software systems, software architectures, middleware systems, and assurance of non-functional software qualities.