Noname manuscript No.
(will be inserted by the editor)

Characterizing and Detecting Inefficient Image Displaying Issues in Android Apps

Wenjie Li · Yanyan Jiang · Chang Xu · Yepang Liu · Xiaoxing Ma · Jian Lyu

Received: date / Accepted: date

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 presents an empirical study of 162 real-world IID issues collected from 243 popular open-source Android apps, validating the presence and severity of IID issues, and then sheds light on these issues' characteristics to support future research on effective issue detection. Based on the findings of this study, we developed a static IID issue detection tool TAPIR and evaluated it with real-world Android apps. The experimental evaluations show encouraging results: TAPIR detected 43 previously-unknown IID issues in the latest version of the 243 apps, 16 of which have been confirmed by respective developers and 13 have been fixed.

Keywords Android app · inefficient image displaying · performance · empirical study · static analysis

1 Introduction

Media-intensive mobile applications (apps for short) must carefully implement their CPU- and memory-demanding image displaying procedures. Otherwise user experiences can be significantly affected [1]. For example, inefficiently displayed images can lead to app crash, UI lagging, memory bloat, or battery

Wenjie Li, Yanyan Jiang, Chang Xu, Xiaoxing Ma, Jian Lyu
State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China
E-mail: wenjielinju@gmail.com, jyy, changxu@nju.edu.cn, xxm, lj@nju.edu.cn

Yepang Liu
Shenzhen Key Laboratory of Computational Intelligence, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
E-mail: liuyp1@sustc.edu.cn

drain, and finally make users abandon the concerned apps even if they are functionally perfect [2].

In this paper, we empirically found 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 to some extent (within the scope of general performance bugs [3–7] or image displaying performance analysis [8, 9]), there still lacks a thorough 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 deeper understanding of IID issues, this paper presents an empirical study towards characterizing IID issues in mobile apps. We carefully localized 162 IID issues (in 36 apps) from 1,826 issue reports and pull requests in 243 well-maintained open-source Android apps in F-Droid [10]. Useful findings include:

1. Most IID issues cause app crash (30.9%) or slowdown (45.1%), and handling lots of images and/or large images are the primary causes. This finding is useful for developing reasonable workloads and test oracles for dynamic manifestation and detection of IID issues in Android apps.
2. A few root causes have covered most (90.1%) examined IID issues: non-adaptive image decoding (45.1%), repeated and redundant image decoding (26.5%), UI-blocking image displaying (11.1%), and image leakage (7.4%). We also extracted sufficient conditions for localizing these issues in an Android app's execution trace, which would benefit both dynamic and static analyzers.
3. Certain anti-patterns can be strongly correlated with IID issues: image decoding without resizing (23.4%), loop-based redundant image decoding (22.2%), image decoding in UI event handlers (11.1%), and unbounded image caching (4.3%). This finding lays the foundation of our pattern-based lightweight static IID issue detection technique.
4. TODO:add other findings

To the best of our knowledge, this paper presents the first systematic empirical study of IID issues in real-world Android apps, and provides key insights on understanding, detection, and fixing of IID issues.

Based on these findings, we design and implement TAPIR, a static analyzer for IID issue detection in Android apps. We experimentally validated the effectiveness of TAPIR, and applied it to the latest versions of all 243 studied apps. TAPIR reported surprisingly encouraging results that 43 *previously-unknown* IID issues in 16 apps (24 of the 43 issues are from eight apps that previously suffered from IID issues) were manually confirmed as true positives and reported to respective developers, among which 16 have been confirmed by the developers and 13 have been fixed.

In summary, our paper makes the following contributions:

1. We conducted a systematic empirical study of IID issues in real-world Android apps. The findings provide key insights to facilitate future research, and our dataset of IID issues are publicly available for follow-up studies[1].
2. For a more intuitive presentation and understanding of IID issues, we extracted IID issues' code slices and graphically represented them in a standard format. The graphical representation of IID issues are publicly available[2].
3. Based on our empirical findings, we devised a static pattern-based IID issue detection technique, TAPIR, and validated its effectiveness using real-world Android apps.

In our preliminary conference version of this work [11] , 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 extended our previous work from the following perspectives:(1)We extracted 162 IID issues' code slices and graphically represented them in a standard format. (2)We extended answers to our research questions (i.e., RQ1, RQ2, and RQ3) based on further empirical study results. (3)We additionally studied and answered two important research questions (i.e., RQ4 and RQ5) to provide a more comprehensive understanding for IID issues. (4) We provided more details of the empirical study, issue examples, technique design, and experimental setup.

The rest of this paper is organized as follows. Section 2 introduces the background knowledge of image displaying in Android apps. Section 3 presents the methodology of our empirical study for IID issues in Android apps and discusses our empirical findings. Section 4 presents the design and implementation of our TAPIR tool. Section 5 experimentally evaluates TAPIR with popular and open-source Android apps and discusses its results and the threats to validity. Section 6 presents related work, and Section 7 concludes this paper.TODO:need to be updated

2 Background

Image displaying, although seemingly straightforward, is actually a non-trivial process in Android apps and can be subject to various performance defects. In this section we introduce the image displaying process in Android apps and its related inefficient image displaying (IID) issues.

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 [1]:

[1] `https://github.com/IID-dataset/IID-issues`.
[2] `https://github.com/IID-dataset/IID-issues-graphical-representation`.

- *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 can also save precious CPU/GPU cycles for image decoding and transformation, but it would incur significant space overhead.
- *Image rendering* for physically displaying an image object on an Android device's screen. Images are rendered natively by the Android framework [12].

2.2 Inefficient Image Displaying (IID)

Image displaying is both computation- and memory-intensive. Displaying a full resolution image on a high-resolution display may cost:

- hundreds of milliseconds of CPU *time* [5], which can cause an observable lag, and
- tens of megabytes of *memory* [13], which 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 experiences:

1. Decoding images in the UI thread can significantly degrade an app's performance, causing its slow responsiveness or even "app-not-responding" anomalies[3].
2. Image objects not being freed in time can consume significantly large amounts of memory, leading to `OutOfMemoryError` and unexpected app terminations[4].
3. Improperly stored (cached) images may cause repeatedly (and unnecessary) processing of the same images, resulting in meaningless performance degradation and energy waste[5].

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] `https://github.com/wordpress-mobile/WordPress-Android/issues/5777`.

[4] `https://github.com/AnimeNeko/Atarashii/issues/6`.

[5] `https://github.com/TeamNewPipe/NewPipe/pull/166`.

To better understand and detect IID issues, in this paper we conduct an empirical study to systematically investigate IID issues in Android apps, and work for automated IID issue detection technique based on our empirical study results.

3 Empirical Study

3.1 Methodology

Our empirical study follows a methodology similar to those adopted in existing work [14, 15] for characterizing real-world Android app bugs. We extracted a set of 162 real-world IID issues by keyword search and manual inspection from 243 well-maintained open-source Android apps of realistic usage in F-Droid [10] using the process in Section 3.1.1. We further analyzed these issues by the methodology in Section 3.1.2.

3.1.1 Dataset collection

We conducted the empirical study based on a collection of IID issues from Android apps. Figure 1 illustrates the overall issue collection process.

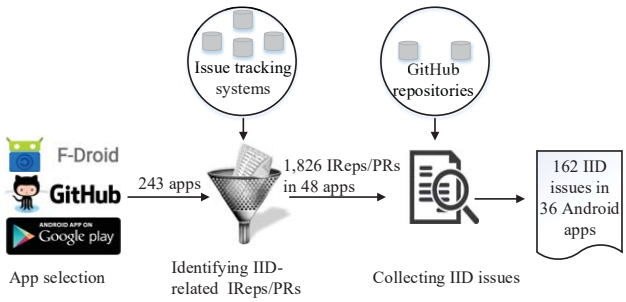

Fig. 1 The IID issue collection process

App selection. We selected all 243 Android apps from 1,093 randomly selected Android apps in F-Droid [10] as our study subject, meeting the following selection criteria:

1. *Open-source*: also 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 IID-related issue reports and pull requests. An app user's *issue report* (IRep for short) usually denotes a manifested app bug from end

users. An app developer's *pull request* (PR for short), on the other hand, possibly contains the developer's perspective on a concerned app bug. Therefore, we collected both of them in the empirical study. We first identified potential IID-related IReps and PRs in the GitHub repositories by a keyword search in their issue tracking systems using the following keywords[6]:

```
image   bitmap     decode   display
picture photograph show   thumbnail
```

Any IRep or PR that contains one of the above keywords in its title, body, or comment was then manually inspected to further confirm whether it indeed *fixed any performance bug*:

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 an image-related 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[7].

After the manual inspection, we obtained a total of 1,826 IReps/PRs in 48 apps, which are from 22,023 IReps/PRs returned by the keyword search from the initially selected 243 Android apps.

Collecting IID issues and their patches. We then inspected the GitHub commits associated with the 1,826 IReps/PRs to decide whether they correspond to IID issues. For each code patch (may patch several places or files in the concerned repository, and a commit may also contain several patches) for fixing a particular image-displaying-related performance bug that is clearly documented in the corresponding IReps/PR, we consider this patch related to a new IID issue. As such, for each decided IID issue, we obtained a patch for fixing it and its textual descriptions in the corresponding IRep/PR, which would suffice for our further manual inspection in order to answer research questions in this study.

Finally, we collected a total of 162 IID issues (distributed in 71 IReps/PRs) in 36/243 (14.8%) studied Android apps. These numbers (162 issues and 14.8% coverage) suggest that IID issues are definitely not rare, and can be considered as common in practice and deserving an in-depth study.

Extracting IID issues code slices.(TODO:Need to be rewritten) We extracted all 162 IID issues' code slices to get a deeper understanding of them from the perspective of the whole execution process of image displaying. An

[6] These keywords are general natural language words related to image displaying. They come from existing research work, e.g., [5,6] and our empirical study experience.

[7] For those issues that do not contain any explicit link to any patch, we conducted a bisect on their GitHub repositories to find potential fixing patches by following the methodology of existing work [16].

IID issue's code slice is a subset of statements that directly or indirectly influence the execution of inefficient image displaying through chains of dynamic data and control dependencies, which can help users focus their attention on a subset of program statements which are responsible for the issue. The process of extracting IID issues' code slices consist of following steps:

1. Obtaining the IID issues buggy code.
2. Obtaining the IID issues issue-triggering context information based on its issue description in its related issue report and comments.
3. Reasoning about the test input that can reveal the IID issue based on its buggy code and triggering context information.
4. Based on the test input, reasoning about the IID issues (hypothetical) execution trace by examining call sequences and arguments of image displaying APIs.
5. Extracting its code slices from the obtained execution trace. It starts with a lifecycle handler (e.g., onCreate()) or an event handler (e.g., onClick()), and ends with a statement that displays or saves (e.g., upload to a server) the image data related to the IID issue.

Graphical representations of IID issues code slices.(TODO:Need to be rewritten)

Graph-based approaches provide a good way to visualize the overall flow of control, where nodes are associated with activities and edges with control or data flow between activities.

To provide a good way to visualize the overall flow of IID issues' execution, we un-parsed the code slice into an execution skeleton. The skeleton is a *directed acyclic graph* (DAG) consisting of function nodes and directed edges. A function node represents the implementation of a functional part of image displaying (e.g., image decoding). A directed edge represents an execution order. The functional parts of image displaying include:

1. *Image decoding*: Decoding an image into an Android-recognizable in-memory object (e.g., Bitmap, Drawable, and BitmapDrawable) and represented by green blocks.
2. *Image resizing*: Decoding an image into an Android-recognizable in-memory object (e.g., Bitmap, Drawable, and BitmapDrawable) and represented by yellow blocks.
3. *Image disk caching*: Adding an image to a disk cache and represented by orange blocks.
4. *Image memory caching*: Adding an image to a disk cache and represented by orange blocks.
5. *Image rendering*: Displaying an image object on an Android devices screen and Represented by blue blocks

3.1.2 Analyzing the IID issues

The analysis of collected IID issues is organized around the following research questions:

RQ1. *What are the triggering conditions and consequences of IID issues?*

RQ1 concerns user-perceived manifestation condition and consequences of IID issues, and thus is answered by inspecting the textual information in the titles, bodies, and comments of the collected IReps and PRs. Recall that since all collected issues contain clear consequence descriptions, we only need to archive them and extract their triggering conditions through these descriptions. We can further confirm the correctness of the description by analyzing their patches.

RQ2. *What are the root causes of IID issues?*

The root causes are extracted by a hypothetical execution of these apps. For most IID issues containing known IID-inducing triggering conditions (displaying lots of images or large images), we take these known conditions as their input. For the other IID issues, we infer their triggering conditions (one image or lots images) by analyzing their patches. We reason about the (hypothetical) execution traces by examining call sequences and arguments of image displaying APIs, and extract characteristics of these traces as root causes of the IID issues.

RQ3. *Are there common anti-patterns for IID issues?*

We inspect the patches of investigated to find whether there are common anti-patterns correlated to IID issues. We are particularly interested in *code* patterns which can facilitate lightweight static `lint`-like checkers.

RQ4. *How did developers inappropriately implement image displaying?*

Understanding different inappropriate implementation types and investigating the frequency of each considering inappropriate implementation type of IID issues can assist IID issue detection approaches or developers in locating IID issues in mobile apps. We investigated how developers implement inefficient image displaying at code level based on IID issues' code slices and patches.

RQ5. *How did developers fix IID issues?*

Historical IID issue fixing data can be used by automatic tools to perform IID issue fixing and also provide useful fixing suggestions to developers. Thus, we decided to perform a manual investigation on the fixing actions of IID issues, such that we gain a deeper understanding of IID issues and find repeated patterns that can be used by other automatic fixing tools or developers.

3.2 Empirical Study Results

3.2.1 RQ1: What are the triggering conditions and consequences of IID issues?

We answer RQ1 by manual inspection of textural information in the collected IReps/PRs, which contain descriptions about IID issues from the perspective of app users. The overall results are summarized as the follows:

Finding 1. Most IID issues can cause app *crash* (30.9%) or *slowdown* (45.1%). In the issues with clearly described triggering conditions, handling *lots* of images (60.0%) and *large* images (41.6%) are the major causes.

This finding is consistent with our intuitions: IID issues typically occur in media-intensive apps, and may result in severe impact on user experiences[8]. Their consequences can be categorized as follows:

- *App crash* (50/162, 30.9%) is the most severe consequence, which is mostly caused by `OutOfMemoryError` in the memory allocation for storing a large image[9].
- *App slowdown* (73/162, 45.1%) is the most common consequence, which includes GUI lagging[10] and slow image displaying[11].
- *Memory bloat* (23/162, 14.2%) in which an apps' consumed memory keeps growing but does not lag or crash the app yet[12], although the app may unnecessarily stop background activities and affect user experiences.
- *Abnormal image displaying* (14/162, 8.6%) occurs when an app's memory is insufficient for decoding large images but does not cause app crash yet, which may also trigger frequent GC (Garbage Collection) and impact user experiences.
- *Application not responding* (2/162, 1.2%) is the extreme case of app slowdown and is usually caused by an app performing time-consuming image displaying operations in the UI thread[13].
- *Others* (13/162, 8.0%) can also result in bad user experiences but their IReps/PRs lack further details for inspection.

We also found that 125 of the 162 studied IID issues contain explicit descriptions about their triggering conditions. All these triggering conditions concern to handling *large* images (50/125, 40.0%), handling *lots* of images (73/125, 58.4%), or both (2/125, 1.6%). For these cases, inefficient handling of large/lots of images mostly cause app crash/slowdown, respectively.

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 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.

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

[9] `https://github.com/the-blue-alliance/the-blue-alliance-android/issues/588`.

[10] `https://github.com/nikclayton/android-squeezer/issues/171`.

[11] `https://github.com/kontalk/androidclient/issues/789`.

[12] `https://github.com/romannurik/muzei/issues/383`.

[13] `https://github.com/ccrama/Slide/issues/1639`.

3.2.2 RQ2: What are the root causes of IID issues?

To further understand on the source-code level how IID issues have occurred, we manually inspected the issues' associated patches to recognize their root causes. The overall results are summarized as follows:

> **Finding 2**. Only a few root causes cover most (90.1%) inspected IID issues: non-adaptive image decoding (45.1%), repeated and redundant image decoding (26.5%), UI-blocking image displaying (11.1%), and image leakage (7.4%).

First, this finding reveals that existing performance bug detectors may have covered only a narrow range of IID issues and it is worthwhile to develop IID-specific analysis techniques. For example, the existing pattern-based analysis [3] detects only part image decoding in the UI thread, the existing resource leakage analysis [17] can be expanded to manual image resource management (the tool itself does not cover), existing image displaying performance analysis [9] 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 root causes 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 which correlate to these root causes (studied later in Section 3.2.3).

The root causes of IID issues are illustrated using the *execution traces* of an app based on a simplified data-flow model. Suppose that executing an app yields 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 (73/162, 45.1%) of the issues are simply caused by directly 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[14], which can waste thousands times of CPU cycles and memory.

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 actual displayed image $im'_{w' \times h'}$ has $w > w' \wedge h > h'$.

Repeated and redundant image decoding. Quite a few (43/162, 26.5%) issues are due to improper storage (particularly, caching) for images such that

[14] `https://github.com/opendatakit/collect/issues/1237`.

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

Fig. 2 Image decoding without resizing in WordPress issue 5701 (simplified)

the same images maybe repeatedly and redundantly decoded, causing unnecessarily 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 associated images im and im' are identical, i.e., $im_{w \times h} = im'_{w \times h}$.

UI-blocking image displaying. Some (18/162, 11.1%) issues are caused by decoding images in the UI thread in an app, even if this has been explicitly discouraged in the Android documentation [1]. A typical example is to decode large images in the UI thread[15], which causes UI blocking, leading obviously slow responsiveness.

Image leakage. Some (12/162, 7.4%) issues are caused by memory (by image objects) leakage such that inactive images cannot be effectively garbage-collected. Memory leakage is another major cause of `OutOfMemoryError` and has been extensively studied in the existing literatures [18, 19].

3.2.3 RQ3: Are there common anti-patterns for IID issues?

Following the analysis of root causes in Section 3.2.2, we inspected the source code of concerned IID issues to identify whether IID issues are related to any particular code *anti-patterns*. The overall results are summarized as follows:

> **Finding 3**. Certain anti-patterns are strongly correlated to IID issues: image decoding without resizing (23.4%), loop-based redundant image decoding (22.2%), image decoding in UI event handlers (11.1%), and unbounded image caching (4.3%). Together with additional bug types mentioned by existing research [9, 17] (29.1%), 90.1% of the examined IID issues could be identified.

These anti-patterns are a firm basis for developing effective static analysis techniques for detecting IID issues, which are further discussed in Section 4 and evaluated in Section 5.

[15] `https://github.com/kontalk/androidclient/issues/789`.

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

Fig. 3 Loop-based redundant image decoding in AntennaPod pull request 1071 (simplified)

```
public class PreviewActivity extends AppCompatActivity {
  protected void onCreate() {
    mediaUri = media.getUrl();
    loadImage(mediaUri); }
  private void loadImage(String mediaUri) {
-   byte[] bytes = ImageUtils.createThumbnail(Uri.parse(mediaUri), ...);
+   new LocalImageTask(mediaUri, size).executeOnExecutor(AsyncTask.
    THREAD_POOL_EXECUTOR);
-   Bitmap bmp = BitmapFactory.decodeByteArray(bytes, ...);
  } }
+ private class LocalImageTask extends AsyncTask<...> {
+   protected Bitmap doInBackground(Void... params) {
+     byte[] bytes = ImageUtils.createThumbnailFromUri(..., Uri.parse(
    mMediaUri);
+     return BitmapFactory.decodeByteArray(bytes, ...);
  } }
public class ImageUtils {
  public static byte[] createThumbnail(Uri imageUri, ...) {
    Bmp = BitmapFactory.decodeFile(imageUri, ...);
  } }
```

Fig. 4 Image decoding in UI event handlers in WordPress issue 5777 (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 Atarashii issue 6 (simplified)

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}`	(*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`	(*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	`universalimageloader.core.Ima getInstance().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 `DiskCache Strategy.{SOURCE, RESULT, ALL}`, but there is no subsequent invocation to `clearDiskCache`.

Image decoding without resizing. IID issues are likely to present if an image potentially from external sources (like network or file system) is decoded with its *original* size. Furthermore, external-source image displaying is specific as a few APIs, which can be detected by a pattern-based analysis.

Surprisingly, this simple anti-pattern already covers 38/162 (23.4%) of all studied IID issues. Fig. 2 gives such an example, in which displaying the thumbnail of a network image may unnecessarily consume about 128MB 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 UI widget used for displaying it (by invoking `createThumbnail()` for resizing images) to reduce unnecessary memory consumption (Lines 4–6).

Loop-based redundant image decoding. IID issues also frequently occur when an image is unintentionally decoded multiple times in a loop. Particularly, Android apps often use some common components (e.g., ListView, GridView, and RecyclerView) to display a scrolling list of images, and these components are all associated with callback methods, which can be frequently invoked.

This anti-pattern covers 36/162 (22.2%) of all studied IID issues. Fig. 3 gives an example, in which the method `getView()` of the Android `ListView` adapter was frequently invoked during the Android app execution (Line 2). *Glide* is a popular third-party library used for image displaying and its method `diskCacheStrategy(DiskCacheStrategy.SOURSE)` specifies that its decoded image read from `podcast.getLogoUrl()` will not be cached and reused (Lines 5 and 7). In this issue, when its user browses a list of images and slides up and down, a lot of images would be decoded repeatedly and result in high unnecessary runtime overhead, leading to GUI lagging. One developer later fixed this issue by modifying `DiskCacheStrategy.SOURSE` to `DiskCacheStrategy.ALL` (Lines 7–8). Then `Glide` can cache and reuse its decoded images, avoiding GUI lagging.

Image decoding in UI event handlers. 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 an UI event handler.

This anti-pattern covers 18/162 (11.1%) of all studied IID issues. Fig. 4 gives such an example, in which a big image read from a local location was decoded in the UI thread and caused the concerned app to run slowly (similar to the code snippet example of *Image decoding without resizing's* in Fig. 2). In this issue, two methods `createThumbnail()` and `decodeByteArray()` are used to decode an image read from a URL site `mediaUri` (Lines 6 and 8) in method `loadImage()`, which was invoked by a callback method `onCreate()`, which was then invoked in the UI thread. This caused the situation that the image decoding was actually in the UI thread and resulted the UI blocking. To fix this issue, one developer later moved the image decoding to a background thread (Lines 7 and 10–13).

Unbounded image caching. 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 7/162 (4.3%) of all studied IID issues. Fig. 5 gives such an example, in which an app crashed because of `OutOfMemoryError` after its user browsed many images. The cause is that the app's image cache `imageCache` was wrongly implemented such that it gathered all decoded images without any image releasing. This made the app's memory consumption keep increasing and quickly exceed an Android device's memory bound (Line 15). Its developer later fixed this issue by adding a soft reference in the image cache so that the cached images could be correctly released when memory usage was tight (Line 16).

3.2.4 RQ4: How did developers inappropriately implement image displaying?

To understand how developers inappropriately implement image displaying in their Android apps, we analyzed the code slices and patches of IID issues. We found that the inappropriate implementation of image displaying can be categorized into two general types: third-party library-specific IID issues and custom implementation-specific IID issues. The former type of inappropriate implementations occur when developers using third-party libraries for image displaying, while the latter type of inappropriate implementations can occur when developers customize the functionality they need. Then we further categorized the two types of inappropriate implementation into subtypes according to their respective implementations.

Third-party library-specific IID issues. Third-party libraries used for image displaying are commonly used in Android apps to reduce implementation efforts. We found that **% percent IID issues are third-party library-specific. Android platform has managed to form a massive and active community of developers in just few years, and the community provides many popular open-source third-party libraries to developers to ease the implementation of image displaying. In practice, it is generally impossible for developers to get familiar with the specification of each third-party library before developing apps. Therefore, they can easily make mistakes when using unfamiliar third-party libraries and IID issues can arise in such cases. We identified two primary reasons for the prevalence of these issues in Android apps: (1) using unsuitable third-party libraries, and (2) third-party library API misuses.

- *Using unsuitable third-party libraries.* Out of the 162 IID issues, ** (**%) issues were caused by using unsuitable third-party libraries. For the third-party libraries used for image displaying, they have different implementation logics and functional emphasis to handle different image displaying scenarios. So for different Android apps that have different execution environments, developers should choose libraries that fits their actual needs. However, many Android apps contain IID issues because of using unsuitable third-party libraries that cannot handle the image display scenes they encounter. For example, Glide are fast and efficient open source media management and image loading library for Android that wraps media decoding, memory and disk caching, and resource pooling into a simple and easy to use interface. ...
- *Third-party library API misuses.* ** (**%)IID issues were caused by third-party library API misuses. Third-party libraries provide mangy functional configuration options for developers to ease app development. In practice, it is generally impossible for developers to get familiar with the specification of each third-party library API before developing apps, and many developers don't fully consider the running environment that Android apps may encounter. Therefore, developers can easily make mistakes when using unfamiliar third-party libraries and IID issues can arise because of third-party library API misuses. The third-party library API misuses consist of

two parts: (1) Lacking necessary third-party library API calls. TODO:add the description;(2) Setting inappropriate API parameter values. For the same API, different value setting can lead to different runtime performance. Such diversity of value setting configuration can easily lead to IID issues, if app developers do not carefully deal with all runtime environments. For example,...

Custom implementation-specific IID issues. Many Android developers choose to customize the implementation of image displaying to facilitate maintenance and meet specific functional requirements. As a result, it brings burden to app developers, who need to ensure that their apps correctly implement the requirements. Unfortunately, this is a non-trivial task and can easily cause IID issues. We found ** such IID issues in our dataset and identified two primary reasons for the prevalence of these issues in Android apps: (1) lack of necessary functions, and (2) inappropriate function implementations.

- *Lack of necessary functions.* ** (**%) IID issues were caused by lacking necessary functions. It is common that a complete image displaying process includes several sub-functional modules. When dealing with image displaying, developers will encounter a variety of running scenarios, such as displaying a large number of images and displaying high resolution images. In order to handle these running scenarios well, developers need to include the necessary function modules in their image display process, such as image caching and image resizing. However, many developers lack experience with efficient image displaying development or don't fully consider the running scenarios that Android apps may encounter, resulting in lacking necessary sub-functional modules in their custom implementation of image displaying and causing IID issues. For example,...
- *Inappropriate function implementations.* ** (**%) IID issues were caused by inappropriate function implementations. The process of image displaying in Android apps is performance-critical and energy-consuming, and implementing an efficient image displaying is a non-trivial task. In practice, developers' custom implementation of image displaying are often problematic, and it is impractical for developers to conduct adequate image displaying tests covering all running scenarios such that IID issues can be easily left undetected before the release of their apps. The inappropriate function implementations consist of two parts: (1) Inappropriate implementation logic; (2) Inappropriate value settings.

3.2.5 RQ5: How did developers fix IID issues?

By analyzing the patches applied to the 162 IID issues in our dataset, we identified the fixing actions applied by the developers to fix the IID issues, which can be a firm basis for developing effective automated program repair techniques for fixing IID issues for follow-up studies. In summary, we identified five common fixing patterns from our dataset.

Adding necessary functions.. The most common fixing action (** out of 162 patches, **%) is to add necessary functions, because many developers lack the experience to implement efficient image displays, their implementations often lack some necessary functionality. This is a common patching strategy for custom implementation-specific IID issues.

Using third-party libraries instead of custom implementations.. Many developers lack the experience to implement an efficient image display, and it is time-consuming for them to locate and fix IID issues by themselves. As a result, many developers chose to replace their custom implementations with mature third-party libraries.

Replacement of third-party libraries.. Different third-party libraries have different performance on different image displaying running scenarios. When developers found that the third-party libraries they originally used cannot effectively handle more complex image displaying scenarios, they chose to use more suitable third-party libraries to replace the third-party libraries used before.

Modifying value settings.. TODO:

Modifying function implementations.. TODO:

The remaining patches are app-specific fixing actions and strongly related to the context of the IID issues.

4 Static Detection of IID Issues

We proposed a static IID issue detector, TAPIR, based on a set of anti-pattern rules extracted from our empirical study results. This section describes the design (in Section 4.1) and implementation (in Section 4.2) of TAPIR.

4.1 IID Issue Anti-pattern Rules

By further inspecting the empirical study results and IID issue cases, we observed 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 [20] official APIs are correlated with IID[16]:

```
decodeFile      decodeFileDescriptor decodeStream
decodeByteArray setImageURI          decodeRegion
createFromPath  createFromStream     setImageViewUri
```

We also observed two popular third-party APIs (APIs invoking third-party library functionalities, not APIs used inside third-party libraries), which are associated with at least two apps in the studied IID issues:

[16] `setImageURI` and `setImageViewUri` both decode and display an image.

```
universalimageloader.core.ImageLoader.
  getInstance().displayImage
Glide.load
```

We call the above eleven image decoding and third-party APIs *issue inducing APIs*. IID issues can occur when these APIs are invoked under issue-inducing *rules*, which consist of API invocation sequence and/or parameter value combinations. These issue-inducing *rules* are characterized in Table 1, which are matched against in the TAPIR static analyzer.

4.2 The TAPIR Static Analyzer

We implemented the pattern-and-rule based static analyzer on top of Soot [21]. TAPIR takes an Android app binary (`apk`) file as input and uses `dex2jar` [22] to obtain a set of Java bytecode files. It then builds the app's context-insensitive call graph, with a few implicit method invocation relations being added, which are used to check rule #4:

1. The methods of `AsyncTask.execute` and `AsyncTask.doInBackground` in the same class have an implicit invocation relationship.
2. The methods of `Thread.start` and `Thread.run` in the same class should have an implicit invocation relationship.

Then TAPIR checks each potential issue-inducing API invocation site (*IS* for short) against the anti-pattern rules in Table 1 using standard program analysis techniques. For each *IS*, we can thus obtain: (1) the data-flow of method parameters by a backward slicing, and (2) the usages of decoded image objects by a forward slicing. In particular, TAPIR checks the anti-pattern rules 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 there lacks the `Option` parameter or its value satisfies the condition specified in Table 1.
2. Rule #2 and #3 (*loop-based redundant image decoding*) are equivalent to checking the call graph reachability between the loop-related method invocations and the *IS*. Furthermore, TAPIR also checks whether there is any data flow between the decoded image and cache-related functions or argument (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 the reachability between the *IS* and method invocations of `Thread.run`, `AsyncTask.DoInBackground`, or `IntentService.onHandleIntent`.
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 *IS* matching at least one anti-pattern rule, an inefficient image display warning is generated, which can be further validated by the respective app developer.

5 Evaluation

In this section, we present the experimental setup (Section 5.1) and results (Sections 5.2 and 5.3) for evaluating TAPIR with: (1) a set of studied IID issues with issue-inducing `apk`s available, and (2) the latest version of all studied 243 apps, followed by a threat analysis (Section 5.4).

5.1 Experimental Setup

To validate the effectiveness of TAPIR, we collected the `apk` archives of all studied apps that have historical `apk`s available (particularly, the `apk`s exactly correspond to our earlier IReps/PRs in our earlier empirical study). This collection process led to a total of 19 confirmed IID issues from nine Android apps, which were used as a ground truth to evaluate whether TAPIR can successfully detect the concerned IID issues. These numbers (19 and 9) seem not large, and it is true that in theory one should be able to compile each IID issue's corresponding app's source code for experiments. However, in practice the dependencies of the concerned Android apps could not be easily resolved, and some large apps failed 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 chose for experiments only those apps whose `apk`s are available corresponding to the studied IID issues and do not suffer from any dependency issue.

Besides, to further evaluate TAPIR's capability of detecting real-world IID issues, we applied it to the latest versions[17] of all the 243 Android apps used in the empirical study to see whether TAPIR can detect previously unknown IID issues. For each TAPIR's reported IID issue, we manually inspected it for confirmation. We submitted the issues confirmed by us to their respective GitHub issue tracking systems for final validation by responsible developers.

In the IID issue reporting process, as most IID issues detected by TAPIR (in an anti-pattern way) are obvious and easy to fix, we did not attach respective patches or open pull requests. We let app developers judge the validity of our reported issues on their own, rather than potentially misleading them by trivial patches. We also obtained some interesting findings, which will be presented later.

Note that in the experiments we applied TAPIR only to analyze the image displaying code of the main logics in the selected apps, i.e., skipping the parts related to third-party libraries, which are out of the apps local source trees. We conducted all experiments on a commodity PC with an Intel Core i7-6700 processor and 16GB RAM.

5.2 Effectiveness Validation Results

[17] The latest `apk` build is always available on F-Droid.

Table 2 Effectiveness Validation Results. Each known IID issue is either a true positive (TP) or a false negative (FN).

App Name (Category, Downloads)	Revision(s)	LOC	#IID (IRep/PR ID)	AP1	AP2	AP3	AP4	TP	FN
OpenNoteScanner [23] (Education, 10K+)	`d34135e`	2.7K	2 (#12)	2	0	0	0	2	0
Subsonic [24] (Multimedia, 500K+)	`68496f6`	23.8K	1 (#299)	0	1	0	0	1	0
WordPress [25] (Internet, 5M+)	`1a8fa65`, `8429f0a`	95.8K	2 (#5290, #5777)	1	0	1	0	2	0
PhotoAffix [26] (Multimedia, 10K+)	`3d8236e`	1.4K	2 (#5)	2	0	0	0	2	0
Kontalk [27] (Internet, 10K+)	`3f2d89d`, `9185a80`	19.6K	3 (#234, #269, #789)	2	0	1	0	3	0
OneBusAway [28] (Navigation, 500K+)	`9f6feea`	15.7K	2 (#730)	2	0	0	0	2	0
NewPipe [29] (Multimedia, 10K+)	`4df4f68`	3.5K	5 (#166)	0	5	0	0	5	0
MoneyManagerEx [30] (Money, 100K+)	`dcf4b87`	63.8K	1 (#938)	1	0	0	0	1	0
BlueAlliance [31] (Education, 10K+)	`c081671`	31.4K	1 (#588)	1	0	0	0	1	0
Total			19	11	6	2	0	19	0

Columns AP1–AP4 respectively denote the number of studied IID issues categorized as a specific anti-pattern.

The overall evaluation results are shown in TABLE 2. All evaluated 19 IID issues belong to three anti-patterns. TAPIR should either correctly detect an IID issue as an anti-pattern instance (i.e., true positive, TP), or fail to detect it (i.e., false negative, FN). The results show that TAPIR correctly identified all the 19 IID issues without any false negative report. Although we have difficulties in evaluating TAPIR against all studied IID issues as explained earlier, we have tried our best to reduce potential bias and the results may reflect the effectiveness of TAPIR to some extent.

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. Still, we conducted such examination on the latest versions of all 243 studied apps as our second part of evaluation.

5.3 Applying TAPIR in Practice

5.3.1 Evaluation Results and Developers' Feedback

Table 3 List of 43 Previously Unknown IID issues found by applying TAPIR to the latest versions of the 243 studied apps.

App Name (Category, Downloads)	Revision	LOC	AP1	AP2	AP3	AP4	Submitted Issue Reports
Newsblur [32] (Reading, 50K+)	`535b879`	20.1K	1	1	1	0	~~#977~~
WordPress [25] (Internet, 5M+)	`30ff305`	95.8K	4	0	0	0	#5232, **#5703**partially-fixed/rejected
Seadroid [33] (Internet, 50K+)	`f5993bd`	37.9K	1	0	3	1	#616, #617, #766
MPDroid [34] (Multimedia, 100K+)	`9b0a783`	20.5K	1	0	0	0	**#837**
Aphotomanager [35] (Multimedia, 1K+)	`9343d84`	12.4K	0	1	1	0	~~#74~~
Conversations [36] (Internet, 10K+)	`1c31b96`	38.0K	0	2	0	0	**#2198**fixed
Owncloud [37] (Internet, 100K+)	`1443902`	49.1K	3(1)	2	1	0	#1862
OpenNoteScanner [23] (Education, 10K+)	`2640785`	3.5K	0	1	0	0	**#69**
Geopaparazzi [38] (Navigation, 10K+)	`71fd81e`	89.9K	2	0	0	0	~~#387~~
Passandroid [39] (Reading, 1M+)	`1382c6a`	6.6K	3	0	0	0	~~#136~~
4pdaclient [40] (Internet, 1M+)	`a637156`	41.9K	0	1	1	0	**#25**fixed
DocumentViewer [41] (Reading, 500K+)	`a97560f`	49.6K	0	1	2	0	#233
Kiss [42] (Theming, 100K+)	`9677dd1`	5.1K	0	0	1	0	**#570**fixed
Bubble [43] (Reading, 10K+)	`9f1e06c`	3.5K	1	0	0	0	#47
Qksms [44] (Communication, 100K+)	`c54c1cc`	55.3K	2(1)	2	2	0	**#718**fixed, **#719**fixed
Photoview [45] (Demo, 10K+)	`6c227ee`	2.1K	0	1	0	0	**#478**
Total			18(2)	12	12	1	

An italic app name denotes it previously suffered from IID issues. Columns AP1–AP4 respectively denote the number of detected issues related to each anti-pattern. Numbers in a bracket are false positives. In the last column, bold/stroke-out issues are explicitly confirmed/rejected by the developers, and the remaining ones are open issues.

Applying TAPIR to the latest version of the 243 apps returned 45 anti-pattern warnings in 16 apps. We manually inspected each warning and categorized it either as a real IID issue (i.e., true positive, TP) or a spurious warning (i.e., false positive, FP). For each TP, we also reported it to its responsible developers. The overall results are listed in TABLE 3.

43 of 45 warnings were manually confirmed to be true instances of anti-patterns, achieving an anti-pattern discovery precision of 95.6%. 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 failed to analyze the `Options` parameter of `decodeByteArray` which contains resized geometries, and thus conservatively reported 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.

We enclosed the 43 issues into 20 issue reports, and submitted them to respective developers (with descriptions of the issues and associated anti-patterns) for their judgement on the validity of these anti-pattern-based IID issues. The last column in Table 3 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%) were confirmed as real performance threats, and 13 of 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. This results also practically validates the effectiveness of the summarized anti-pattern rules in our empirical study.

For the remaining 11 IID issues, developers held various conservative attitudes as discussed below:

1. Most developers rejecting our reports thought 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 acknowledged that their app may encounter performance degradation in some cases, but should be sufficiently fast and thus currently do not plan to fix them.
2. Some developers acknowledged the reported issues, but they claimed to have higher-priority tasks than performance optimization.

Later we shall 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. These results suggest that the future work along this line may focus on systematic generation of testing workloads for manifesting IID issues. Note that we could not have obtained such these findings if we attached trivial patches in the IReps, since developers would be inclined to accept free (and obviously correct) patches for better performance.

5.3.2 Real-world IID Issue Cases

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

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[18].

For this interesting case, we applied TAPIR to the latest version of WordPress and detected one previously detected and two new 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[19]. 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 expected. 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 detected the anti-pattern of *loop-based redundant image decoding* in KISS, and thus we reported this issue to its developers[20]. Unfortunately, the developers explicitly rejected our proposal due to the concern that they believe that the performance impact would be minor and KISS should be kept simple and lightweight.

Interestingly, a year and a half later, one of KISS users encountered and complained a show image displaying problem[21]. Then the developers noticed this and decided that this is truly due to our mentioned IID issue. So 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.

[18] `https://github.com/wordpress-mobile/WordPress-Android/issues/5701`.

[19] Developers consider one report as false positive because they have control of the external image size.

[20] `https://github.com/Neamar/KISS/issues/570`.

[21] `https://github.com/Neamar/KISS/issues/1054`.

5.4 Threats to Validity

We analyze potential threats to the validity of our empirical study and experimental conclusions about TAPIR.

Subject selection. Our empirical study is based on 162 IID issues from 243 open-source Android apps, which, although having a not-small number, may not be completely representative of all IID issues in practice. Nevertheless, we collected these IID issues from well-maintained popular open-source Android apps covering diverse categories to reduce such threats. Furthermore, the evaluation of TAPIR shows that these issues indeed helped detect both previously known and unknown IID issues in practice.

Limitations of TAPIR. TAPIR is lightweight (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 already demonstrates its effectiveness in detecting IID issues. One future work is to develop more sophisticated static and/or dynamic analyses to more precisely detect IID issues.

Custom implementation of image displaying. As mentioned earlier, this work does not consider the source code in third-party libraries used by studied Android apps, which could be another source of IID issues. Developers may also have used ad-hoc implementations for image displaying, causing obstacles to our pattern-based analysis. This aspect of IID issue detection can be a potential future direction.

6 Related Work

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

Understanding performance issues in mobile apps. Understanding performance issues is of critical importance before tackling them. Huang et al. [46] identified several important factors that may impact user-perceived network latencies in mobile apps. Liu et al. [3] 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 et al. [47] performed an in-depth investigation of mobile browser performance by pairwise comparisons between mobile and non-mobile browsers. Huang et al. [48] conducted a systematic measurement study to quantify user-perceived latencies with and without background workloads. Rosen et al. [49] investigated the benefits and challenges of using Server Push on mobile devices for improving mobile performance.

Several studies provide some clues for understanding and detecting IID issues as studied in this work. Wang et al. [5] 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. [4] 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, this paper is the first systematic empirical study of IID issues using real-world Android apps, and provides 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. Diagnosing and detecting performance issues is the basis of fixing and optimizing for performance issues in mobile apps. Mantis [8] estimated the execution time for Android apps on given inputs to identify problem-inducing inputs that can slow down an app's execution. ARO [50] monitored 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 to Android apps. AppInsight [51] instrumented 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 [52] monitored 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 revealed performance issues from inefficient platform code or problematic app interactions. Nistor et al. [53] analyzed sequences of calls to String getter methods to understand the impact of larger inputs on a users perception in Windows Phone apps. Lin et al. [54] proposed an approach, ASYNCHRONIZER, to automatically refactor long-running operations into asynchronous tasks. Kang et al. [55] 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. [3] proposed an approach based on static analysis, which can possibly identify one kind of IID issues: performing bitmap resizing operations in the UI thread. Gao et al. [9] performed 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. In our work, we proposed both common anti-patterns and an effective static analyzer TAPIR to detect real-world IID issues of four types.

Fixing and optimizing performance issues in mobile apps. After performance issue detection, performance optimization is the necessary next step. Lee et al. [56] 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. [48] 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. [57] 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. [58] 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. [59] 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.

7 Conclusion

In this paper we empirically validated the wide existence of inefficient image displaying (IID) issues in open-source Android apps, and studied their root causes, manifestations, and common anti-patterns. Based on these empirical findings, we developed a static IID issue detector TAPIR and evaluated it with real-world apps. The experimental evaluation shows encouraging results: TAPIR detected both previously known IID issues with a high accuracy and previously unknown IID issues confirmed in practice.

Acknowledgments

The authors would like to thank the anonymous reviewers for comments and suggestions. This work is supported in part by National Natural Science Foundation of China (Grants #61690204, #61802165), Science and Technology Innovation Committee Foundation of Shenzhen (Grant No. ZDSYS201703031748284), Program forUniversity Key Laboratory of Guangdong Province (Grant No. 2017KSYS008), and Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

1. https://developer.android.com/topic/performance/graphics.
2. Y. Zhao, M. S. Laser, Y. Lyu, and N. Medvidovic, "Leveraging program analysis to reduce user-perceived latency in mobile applications," in *International Conference on Software Engineering*, 2018.
3. Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1013–1024.

4. A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of Android smells," in *Proceedings of the 24th International IEEE Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2017, p. 10.
5. Y. Wang and A. Rountev, "Profiling the responsiveness of Android applications via automated resource amplification," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*. ACM, 2016, pp. 48–58.
6. M. Linares-Vasquez, C. Vendome, Q. Luo, and D. Poshyvanyk, "How developers detect and fix performance bottlenecks in Android apps," in *IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2015, pp. 352–361.
7. Y. Liu, C. Xu, and S. Cheung, "Diagnosing energy efficiency and performance for mobile internetware applications," *IEEE Software*, vol. 32, no. 1, pp. 67–75, Jan 2015.
8. Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek, "Mantis: Automatic performance prediction for smartphone applications," in *Proceedings of the 2013 USENIX conference on Annual Technical Conference*. USENIX Association, 2013, pp. 297–308.
9. Y. Gao, Y. Luo, D. Chen, H. Huang, W. Dong, M. Xia, X. Liu, and J. Bu, "Every pixel counts: Fine-grained UI rendering analysis for mobile applications," in *IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.
10. F-Droid: A Catalogue of Open-Source Android Apps. https://fdroid.org/.
11. W. Li, Y. Jiang, C. Xu, Y. Liu, X. Ma, and J. Lü, "Characterizing and detecting inefficient image displaying issues in android apps," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 355–365.
12. Android platform architecture. https://developer.android.com/guide/platform/.
13. Loading large bitmaps efficiently. https://developer.android.com/topic/performance/graphics/load-bitmap.
14. Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for Android applications," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 396–409.
15. J. Hu, L. Wei, Y. Liu, S.-C. Cheung, and H. Huang, "A tale of two cities: How WebView induces bugs to Android applications," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 702–713.
16. M. Rath, J. Rendall, J. L. Guo, J. Cleland-Huang, and P. Mäder, "Traceability in the wild: Automatically augmenting incomplete trace links," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 834–845.
17. T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang, "Light-weight, inter-procedural and callback-aware resource leak detection for Android apps," *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1054–1076, 2016.
18. G. Xu and A. Rountev, "Precise memory leak detection for Java software using container profiling," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 151–160.
19. D. Yan, G. Xu, S. Yang, and A. Rountev, "Leakchecker: Practical static memory leak detection for managed languages," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014, p. 87.
20. https://developer.android.com/reference/.
21. Soot project website. http://sable.github.io/soot/.
22. dex2jar. https://sourceforge.net/projects/dex2jar/.
23. OpenNoteScanner. https://github.com/ctodobom/OpenNoteScanner.
24. Subsonic. https://github.com/daneren2005/Subsonic.
25. WordPress. https://github.com/wordpress-mobile/WordPress-Android.
26. PhotoAffix. https://github.com/afollestad/photo-affix.
27. Kontalk. https://github.com/kontalk/androidclient.
28. OneBusAway. https://github.com/OneBusAway/onebusaway-android.
29. NewPipe. https://github.com/TeamNewPipe/NewPipe.
30. MoneyManagerEx. https://github.com/moneymanagerex/android-money-manager-ex.
31. BlueAlliance. https://github.com/the-blue-alliance/the-blue-alliance-android.
32. NewsBlur. https://github.com/samuelclay/NewsBlur.

33. Seadroid. https://github.com/haiwen/seadroid.
34. MPDroid. https://github.com/abarisain/dmix.
35. Aphotomanager. https://github.com/k3b/APhotoManager.
36. Conversations. https://github.com/siacs/Conversations.
37. Owncloud. https://github.com/owncloud/android.
38. Geopaparazzi. https://github.com/geopaparazzi/geopaparazzi.
39. Passandroid. https://github.com/ligi/PassAndroid.
40. 4pdaclient. https://github.com/slartus/4pdaClient-plus.
41. DocumentViewer. https://github.com/PrivacyApps/document-viewer.
42. Kiss. https://github.com/Neamar/KISS.
43. Bubble. https://github.com/nkanaev/bubble.
44. Qksms. https://github.com/moezbhatti/qksms.
45. PhotoView. https://github.com/chrisbanes/PhotoView.
46. J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl, "Anatomizing application performance differences on smartphones," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 165–178.
47. J. Nejati and A. Balasubramanian, "An in-depth study of mobile browser performance," in *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2016, pp. 1305–1315.
48. G. Huang, M. Xu, F. X. Lin, Y. Liu, Y. Ma, S. Pushp, and X. Liu, "Shuffledog: Characterizing and adapting user-perceived latency of android apps," *IEEE Transactions on Mobile Computing*, vol. 16, no. 10, pp. 2913–2926, 2017.
49. S. Rosen, B. Han, S. Hao, Z. M. Mao, and F. Qian, "Push or request: An investigation of HTTP/2 server push for improving mobile performance," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 459–468.
50. F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Profiling resource usage for mobile applications: A cross-layer approach," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 321–334.
51. L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "AppInsight: Mobile app performance monitoring in the wild," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, vol. 12, 2012, pp. 107–120.
52. L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda, "Panappticon: Event-based tracing to measure mobile application and platform performance," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. IEEE, 2013, pp. 1–10.
53. A. Nistor and L. Ravindranath, "Suncat: Helping developers understand and predict performance problems in smartphone applications," in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 282–292.
54. Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for Android applications through refactoring," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 341–352.
55. Y. Kang, Y. Zhou, H. Xu, and M. R. Lyu, "DiagDroid: Android performance diagnosis via anatomizing asynchronous executions," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 410–421.
56. K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn, "Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 151–165.
57. Y. Zhao, M. S. Laser, Y. Lyu, and N. Medvidovic, "Leveraging program analysis to reduce user-perceived latency in mobile applications," in *Proceedings of the International Conference on Software Engineering*, 2018.
58. Y. Lyu, D. Li, and W. G. Halfond, "Remove RATs from your code: Automated optimization of resource inefficient database writes for mobile applications," in *Proceedings of the 27th International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 310–321.

59. D. T. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, and Q. Yang, "Reducing smartphone application delay through read/write isolation," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 287–300.