IMGDroid: Detecting Image Loading Defects in Android Applications

Wei Song
School of Computer Sci. & Eng.
Nanjing University of Sci. & Tech.
Nanjing, China
wsong@njust.edu.cn

Mengqi Han
School of Computer Sci. & Eng.
Nanjing University of Sci. & Tech.
Nanjing, China
hanmengq77@163.com

Jeff Huang
Parasol Laboratory
Texas A&M University
College Station, TX, USA
jeff@cse.tamu.edu

***Abstract*—Images are essential for many Android applications or apps. Although images play a critical role in app functionalities and user experience, inefficient or improper image loading and displaying operations may severely impact the app performance and quality. Additionally, since these image loading defects may not be manifested by immediate failures, e.g., app crashes, existing GUI testing approaches cannot detect them effectively. In this paper, we identify five anti-patterns of such image loading defects, including image passing by intent, image decoding without resizing, local image loading without permission, repeated decoding without caching, and image decoding in UI thread. Based on these anti-patterns, we propose a static analysis technique, IMGDroid, to automatically and effectively detect such defects. We have applied IMGDroid to a benchmark of 21 open-source Android apps, and found that it not only successfully detects the 45 previously-known image loading defects but also finds 15 new such defects. Our empirical study on 1,000 commercial Android apps demonstrates that the image loading defects are prevalent.**

***Index Terms*—Android app, image loading, defect analysis**

I. Introduction

Android applications (apps for short) often use images not only to enhance user experience but also to better provide their functionalities and services to users. Android SDK and several third-party libraries (frameworks) provide easy-to-use APIs, which allows app developers to implement image loading and displaying conveniently. Our empirical study on 1,000 commercial apps shows that over 89% apps use such SDKs or libraries. However, since the high-resolution images (e.g., the pictures taken by the device camera) are usually very large, inefficient or improper programming practices (e.g., the APIs are used inappropriately) can severely impact the performance and quality of apps, causing memory bloat, GUI lagging, slow responsiveness, or app crashes [1], [2], [3]. These inefficient programming practices of image loading, called *image loading defects* in this paper, are regarded as non-functional bugs [3].

Since the image loading defects mainly impact the performance of apps, existing testing-based approaches that aim at finding app failures may find difficulties in manifesting them [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. Besides testing approaches, although a wide variety of techniques have been proposed to help improve app performance and quality [15], [1], [16], [17], [18], [19], [20], [21], [22], [23], little work focuses on the defects or code smells particularly relevant to image loading and displaying in apps [2], [3].

In this paper, we extract five image loading anti-patterns from both *Android Developers*[1] and the reported issues of real-world apps in GitHub[2], and develop an automated static analysis to detect such image loading defects based on these anti-patterns. The five anti-patterns are all reported to reduce the performance or even the quality of apps, which are summarized below:

- **Image passing by intent** refers to the situation that an app employs intents to pass images between different activities, which severely degrades the app performance.
- **Image decoding without resizing** refers to the situation that an image is decoded without its shape and size reduced to fit for the component (e.g., a view) in which it displays.
- **Local image loading without permission** refers to the situation that an app tries to load images from the local storage of the device with no access permission, which directly makes the app crash.
- **Repeated decoding without caching** refers to the situation that an image is decoded in a component's callback function, and thus the image could be repeatedly decoded each time when the callback function is invoked.
- **Image decoding in UI thread** refers to the situation that an image is decoded in the UI thread of the app, which can degrade the app performance, causing GUI lagging and even ANR (application not responding).

In practice, there may exist other kinds of defects relevant to image loading and displaying, but in this paper we only focus on the five image loading anti-patterns. For each of the five anti-patterns, we develop a static analysis to automatically detect such defects (concrete instances matching the anti-pattern) in apps. Since some anti-patterns can occur to different image frameworks (including the Android SDK), our static analysis varies accordingly and is tailored to different image frameworks. To ensure the soundness of our analysis, all image loading operations in the app are covered and analyzed. To balance accuracy and scalability, our analysis is based on a context-insensitive caller (callee) analysis and a path-sensitive inter-procedural control flow analysis.

[1]https://developer.android.com/guide
[2]https://github.com

We implemented our static analysis in an open-source tool IMGDroid based on Soot [24]. IMGDroid is written in Java and is publicly available on GitHub[3]. To evaluate the effectiveness and efficiency of our approach, we have applied IMGDroid to 21 open-source Android apps. IMGDroid finished the analysis in 2,469 seconds, and it not only successfully detected the 45 previously-known image loading defects but also found 15 new such defects in these 21 apps. To investigate image loading defects extensively, with the help of IMGDroid, we further conducted an empirical study on 1,000 commercial apps randomly selected from Google Play (downloaded in Dec 2018). The empirical results demonstrate image loading defects are pervasive in practice: 865 (86.5%) apps involve at least one kind of image loading defects, and each of the apps has on average 6.37 image loading defects.

In a nutshell, our contributions are highlighted as follows:

1) According to the issue reports of real-world apps, we summarize five common image loading anti-patterns in Android apps that can degrade app performance and quality or even lead to app crashes.
2) We present IMGDroid, a static analysis approach and an open-source tool, to automatically detect image loading defects based on the five anti-patterns. The experiment on a benchmark of 21 open-source apps show the effectiveness and efficiency of IMGDroid.
3) With IMGDroid, we conduct an empirical study on 1,000 real-world commercial apps. The results indicate that image loading defects are severe in practice.

The remainder of the paper is organized as follows. Section II introduces the background on image loading in Android apps. Section III presents our static analysis for detecting image loading defects. Section IV evaluates our approach with real-world apps. Section V reviews the related work, and Section VI concludes the paper.

II. Background

We first introduce background on image loading and displaying in Android apps. Readers familiar with image APIs in Android may skip this section and jump to Section III directly.

A. Frameworks for Managing Images

When designing Android apps, the developers can employ the relevant APIs from the Android SDK to load and display images in apps. In Android SDK, `BitmapFactory`, `Drawable`, and `ImageView` are the classes responsible for processing images. Developers can use the related methods in these classes to create `Bitmap` objects from various sources, including files, streams, and byte-arrays.

Besides Android SDK, there are also some third-party libraries (frameworks) that are often used to manage images in apps, such as `Universal-Image-Loader`[4], `Glide`[5], `Picasso`[6], `Fresco`[7], etc. `Universal-Image-Loader` provides lots of configuration options and good control over the image loading process. `Picasso` allows for hassle-free image loading often in one line of code. `Glide` is an efficient image loading library for Android focused on smooth scrolling; it supports fetching, decoding, and displaying video stills, images, and animated GIFs. `Fresco` puts images in a special region of Android memory, which lets apps run faster and suffer the OOM (out of memory) error much less often. In this paper, we call these third-party libraries *image frameworks* for short. Table I summarizes the basic information of these image frameworks.

[3]https://github.com/wsong-nj/IMGDroid
[4]https://github.com/nostra13/Android-Universal-Image-Loader
[5]http://bumptech.github.io/glide
[6]https://square.github.io/picasso
[7]https://frescolib.org

TABLE I
Popular Third-Party Frameworks for Image Loading and Displaying in Android Apps

Framework	Release time	Size of memory required	GIF supported	Cache method
`Universal-Image-Loader`	2011	150K+	No	Memory and disk cache
`Glide`	2012	500K	Yes	Memory and disk cache
`Picasso`	2013	10K+	No	Memory and HTTP cache
`Fresco`	2015	2M+	Yes	Anonymous shared memory, local heap memory, and disk cache

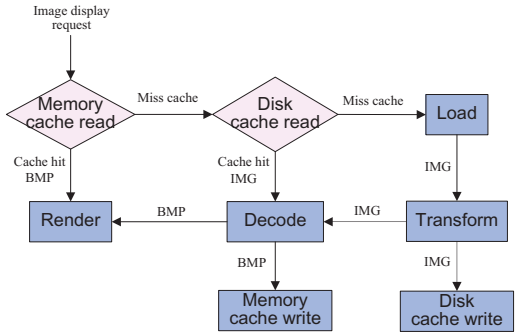

Fig. 1. Workflow of image displaying in Android apps.

B. Image Loading and Displaying Procedure

No matter which image framework (also the Android SDK) is used, the workflow of image loading and displaying in Android apps is almost the same (cf. Figure 1): When receiving a request to show an image, the app first tries to read the corresponding bitmap object from the memory cache before rendering (i.e., displaying the bitmap on the screen of the device). If the bitmap is not in the memory cache, the app then tries to read it from the disk cache. If it succeeds, the app will decode the image (e.g., in JPG format) to obtain the bitmap (which may then be cached in the memory) before rendering. Otherwise, it has to load the image externally from the network or the local storage. After some processing and transformation (e.g., reducing the size of the image), the image can be stored in the disk cache. The transformed image is then decoded (may be cached in the memory) and finally rendered.

Note that the caching mechanism may or may not be used. If it is not used, an external image is displayed in the app by the following steps: loaded, transformed, decoded, and rendered.

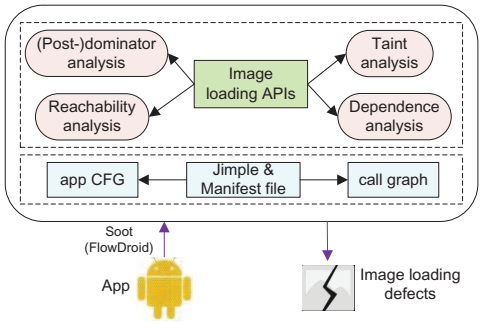

Fig. 2. An overview of IMGDroid.

```
1  public class ImageGridActivity{
2    public void onClick(View v){
3      int id = v.getId();
4      if (id == R.id.btn_preview){
5        Intent intent = new Intent(ImageGridActivity.this,
           ImagePreviewActivity.class);
6        intent.putExtra(ImagePicker.
           EXTRA_SELECTED_IMAGE_POSITION, 0);
7        intent.putExtra(ImagePicker.EXTRA_IMAGE_ITEMS,
           imagePicker.getSelectedImages());
8        intent.putExtra(ImagePreviewActivity.ISORIGIN, isOrigin);
9        intent.putExtra(ImagePicker.EXTRA_FROM_ITEMS, true);
10       startActivityForResult(intent, ImagePicker.
           REQUEST_CODE_PREVIEW);
11     }
12   }
13 }
```

Fig. 3. A reported defect (issue report #49) of image passing by Intent in a real-world app *ImagePicker* (version 0.4.7).

Although the workflow of image loading and displaying is not complex, some image loading defects frequently occur in practice, which are elaborated in Section III.

III. IMGDroid

In this section, we present our static analysis technique IMGDroid for detecting image loading defects in apps.

Figure 2 illustrates the framework of IMGDroid. It takes an app (APK file) as input and generates all image loading defects (could be empty) in the app as output. IMGDroid is based on Soot [24] and FlowDroid [25]. To scale to large commercial apps, IMGDroid detects image loading defects by combining a context-insensitive caller (callee) analysis and a path-sensitive inter-procedural control flow analysis. Starting from the image loading statements, IMGDroid utilizes different static analysis techniques, for example, reachability analysis, dominator and post-dominator (cf. Definitions 1 and 2) analysis, taint analysis [25], and control dependence (cf. Definition 3) analysis, to analyze whether the image loading defects of different kinds exist or not. To ensure the soundness of the static analysis, IMGDroid covers all image loading operations (statements) in the app, and for each image loading statement, IMGDroid searches all possible defects along all potential paths. If such defects exist, the relevant information which is helpful for debugging is also provided, including the kind of defects, the image framework and APIs used, the locations (i.e., in which methods) of the image loading operations.

*Definition 1 (**Dominator**):* In a control flow graph, a node (a statement or a block of statements) N is dominated by another node N' if every path from the entry of the control flow graph to N includes N'. N' is referred to as a dominator of N.

*Definition 2 (**Post-dominator**):* In a control flow graph, a node (a statement or a block of statements) N is post-dominated by another node N'' if every path from N to the exit of the control flow graph includes N''. N'' is referred to as a post-dominator of N.

*Definition 3 (**Control Dependence**):* In a control flow graph, a node (a statement or a block of statements) N is control-dependent on another node C iff there exists a path ρ from C to N such that any statement (node) N' in ρ (excluding C and N) is post-dominated by N, and C is not post-dominated by N.

In each of the following sub-sections, we first describe a kind of image loading defects, and then elaborate on the method how IMGDroid detects such kinds of defects.

A. Detecting Image Passing by Intent

Android uses *intent* for the interaction between different components (e.g., activities). Intent can bring some data of different types, e.g., int, Boolean, String, or Parcelable and Serializable objects. If an activity intends to send some data to another activity, it should first prepare the data through `intent.putExtra(``key'', value)` or `bundle.putParcelable(``key'', value)` and `intent.putExtra(bundle)`, before starting the new activity via `startActivity(intent)`. However, if we use intent to pass data with large size (e.g., 1M or even 0.5M), the app performance can be undermined and exceptions may occur[8]. Thus, using intent to pass image is regarded as a kind of image loading defects (cf. Anti-pattern 1). To avoid this, one can use intent to only pass the address or ID of the image.

*Anti-pattern 1 (**Image Passing by Intent**):* Passing image via intent may degrade the app performance and even cause an exception (i.e., app crash).

With the above explanation, our approach to detecting image passing by intent is straightforward: in the inter-procedural control flow graph of the app, if the `startActivity(intent)` statement is reachable from `intent.putExtra(``key'', value)` or `bundle.putParcelable(``key'', value)`, and the type of *value* is `Bitmap`, `Drawable`, or `BitmapDrawable`, then there is a defect of image passing by intent.

Figure 3 presents such a defect[9], which can be effectively detected by our approach: the `startActivity(intent)` statement at Line 10 is reachable from the `intent.putExtra()` statement at Line 7, and `imagePicker.getSelectedImages()` returns a bitmap.

[8]https://developer.android.com/guide/components/activities/parcelables-and-bundles

[9]https://github.com/jeasonlzy/ImagePicker/issues/49

TABLE II
APIS FOR IMAGE DECODING, RESIZING, AND CACHING IN DIFFERENT FRAMEWORKS

Framework	Decoding	Resizing	Caching
Android native APIs	`BitmapFactory.decodeFile(option), .decodeFileDesccriptor(option), .decodeStream(option), .decodeByteArray(option), .decodeRegion(option)`	`option.Injustdecodebounds = true & option.inSampleSize > 1`	`LruCache.put()`
	`Drawable.createFromPath(), .createFromStream()`		
	`ImageView.setImageURI()`		
`Universal-Image-Loader`	`ImageLoder.displayImage(option)`	`option.imageScaleType()`	`option.cacheInMemory()`
`Fresco`	`Fresco.setController(option)`	`option.setResizeOptions()`	By default
`Glide`	`Glide.with().load()`	`Glide.with().load().override()`	`Glide.with().load().skipMemoryCache()`
`Picasso`	`Picasso.with().load()`	`Picasso.with().load().resize(), Picasso.with().load().resizeDimen(), Picasso.with().load().fit()`	`Picasso.with().load().memoryPolicy(), Picasso.with().load().networkPolicy()`

B. Detecting Image Decoding Without Resizing

The images may come in different shapes and sizes. Their sizes often exceed the requirement of a typical application interface. Displaying an oversized image on a slim view does not bring any visual benefit, and it may have a negative impact on the app performance (e.g., occupying additional much memory) or even lead to OOM (out of memory) exceptions (cf. Anti-pattern 2)[10]. For example, since the resolution of a picture (e.g., 4048 $\times$ 3036 pixels) taken with the camera of an Android device is usually much higher than the screen density, if the bitmap configuration used is ARGB_8888 (the default for Android 2.3 and higher), loading a single picture into memory takes about 48 MB of memory (4048 $\times$ 3036 $\times$ 4 bytes), which may immediately use up all the memory available to the app. Therefore, when displaying a picture, it is expected to compress it, and the size of the compressed image should be similar to the size of the control used to display it.

*Anti-pattern 2 (**Image Decoding Without Resizing**):* A decoded image could be considerably big in size, which is expected to be resized. Otherwise, it may degrade the app performance or even cause the OOM (out of memory) exception.

Since the resource images are usually small and the developers have known their sizes, there is no need to resize them. Thus, for Anti-pattern 2, we only consider external pictures that come from the network and SD card. It is a true positive provided there is a chance that a large picture can be loaded.

Different image frameworks generally use two ways to implement image decoding and resizing. In the first way, the decoding and resizing are implemented in one method, i.e., the image resizing is controlled by a parameter (e.g., *options*) of the decoding method. In the second way, the decoding and resizing are implemented by two separate methods, respectively. Android native APIs from the class `BitmapFactory` and the image frameworks such as `Universal-Image-Loader` and `Fresco` follow the first way, while the image frameworks such as `Glide` and `Picasso` follow the second way. There are Android native APIs that only implement image decoding without considering image resizing, i.e., `Drawable.createFromPath()`, `Drawable.createFromStream()`, and `ImageView.setImageURI()`. If these three methods are used, the corresponding images are decoded without resizing. The second and third columns of Table II summarize the concrete decoding and resizing APIs of different image frameworks.

[10]https://developer.android.com/topic/performance/graphics/load-bitmap.html#read-bitmap

```
1  public class SkiaImageDecoder implements ImageDecoder{
2    public Bitmap decode(Context context, Uri uri) throws Exception
         {
3      Bitmap bitmap;
4      String uriString = uri.toString();
5      Options options = new Options();
6      options.inPreferredConfig = Config.RGB_565;
7      if(uriString.startsWith(ASSET_PREFIX)){
8        bitmap = BitmapFactory.decodeStream(context.getAssets().
             open(uriString.substring(ASSET_PREFIX.length())), null,
             options);
9      }
10     if(bitmap != null){
11       return bitmap;
12     }
13     throw new RuntimeException("Skia image region
           decoder returned null bitmap - image
           format may not be supported");
14   }
15 }
```

Fig. 4. A reported defect (issue report #608) of image decoding without resizing in a real-world app *Leafpic* (version 0.6-beta-1).

According to how image decoding and resizing are implemented, we have the following two ways to check whether an image is decoded without being resized:

- For Android native APIs and the image frameworks such as `Universal-Image-Loader` and `Fresco`, if the decoding statement does not have the argument (e.g., *option*) for image resizing, the image is decoded but not resized. Otherwise, we further check whether the argument has been adapted to reduce the image size before decoding: If there is a statement (statements) that reduces the image size by changing the argument and the statement dominates the decoding statement, then there is no defect; otherwise, the image is decoded without being resized.

```
1  public class ImagePagerAdapterFromCursor extends PagerAdapter{
2    protected View createViewWithContent(int position, ViewGroup
         container, String fullPhotoPath, String debugContext, int
         size){
3      final File imageFile = new File(Environment.
         getExternalStorageDirectory(),Environment.
         DIRECTORY_DCIM);
4      Bitmap bitmap = HugeImageLoader.loadImage(imageFile,
         MAX_IMAGE_DIMENSION, MAX_IMAGE_DIMENSION)
         ;
5    }
6  }
7  public class HugeImageLoader{
8    public static Bitmap loadImage(File file, int maxWidth, int
         maxHeight){
9      BitmapFactory.Options options = new BitmapFactory.Options();
10     options.inJustDecodeBounds = true;
11     BitmapFactory.decodeFile(file.getAbsolutePath(), options);
12     int downscale = calculateInSampleSize(options, maxWidth,
         maxHeight);
13     options.inSampleSize = downscale;
14     options.inJustDecodeBounds = false;
15     return BitmapFactory.decodeFile(file.getAbsolutePath(),
         options);
16   }
17 }
```

Fig. 5. A reported defect (issue report #94) of local image loading without permission in a real-world app *A photo Manager* (version 0.8.3.200315).

- For the image frameworks such as `Glide` and `Picasso`, our detection is based on the post-dominator analysis on the inter-procedural control flow graph of the app. For an image decoding statement in the control flow graph, if it is always post-dominated by a corresponding resizing statement, then the image loading is correct; otherwise, there is a defect of decoding without resizing.

Figure 4 exhibits such a defect[11], which can be effectively detected by our approach: the `decodeStream()` method is invoked at Line 8, but its argument *option* is not adapted between Line 5 and Line 8 as suggested in Table II for image resizing.

C. Detecting Local Image Loading Without Permission

For Android 6.0 or higher, users are not notified of any app permissions when installing apps. Instead, users are asked to grant the permissions at runtime. Users may also have the option to enable or disable permissions one-by-one in system settings. Hence, the apps should always check for and request permissions at runtime to avoid runtime errors. Reading images from the local storage (SD card) is no exception. Otherwise, the app may crash due to no access permission (cf. Anti-pattern 3)[12].

*Anti-pattern 3 (**Local Image Loading Without Permission**):* Before reading images from the local storage, the app should check for and request the access permission at runtime. Otherwise, the app may crash due to no access permission.

[11]https://gitlab.com/HoraApps/LeafPic/-/issues/608

[12]https://developer.android.com/training/permissions/requesting

Our approach to detecting this category of defects for different image frameworks is similar, which includes three steps:

1) We first check whether there is an image decoding statement that uses a variable returned by the invocation of the method `getExternalStorageDirectory()`. If no, there is no such defect. Otherwise, go to step (2).
2) We next query in the AndroidManifest file whether the app declares to get the permission to read the external storage. If no, we find a defect of local image loading without permission. Otherwise, go to step (3).
3) We finally check whether the image decoding statement is control-dependent on a statement that determines whether the app has obtained the permission to read the external storage (i.e., `if(ContextCompat.checkSelfPermission(android.permission.READ_EXTERNAL_STORAGE))` or `if(EasyPermissions.requestPermissions(android.permission.READ_EXTERNAL_STORAGE))`). If no, we find a defect of local image loading without permission.

The first two steps can be implemented based on the data flow analysis (taint analysis) of FlowDroid [25]. Let the declarations of the methods `getExternalStorageDirectory()` (the declaration of the required permission is also included) and `decodeFile()` (or other decoding APIs in Table II) be the source and sink, respectively. FlowDroid can return all statement pairs (e.g., $<from, to>$) that may lead to the defects of local image loading without permission, where $from$ is a statement that invokes `getExternalStorageDirectory()` and to is the corresponding decoding statement. For each pair of statement $<from, to>$, we finally determine whether or not the decoding statement to loads a local image without the permission checked first.

Figure 5 shows such a defect[13], which can be effectively detected by our approach: the statement at Line 4 (Line 11) loads (decodes) an image from the local storage; although the app's AndroidManifest file declares the permission to read the external storage, the app performs the decoding operation without firstly checking whether or not it has obtained the corresponding permission.

D. Detecting Repeated Decoding Without Caching

In many cases, the loaded images are displayed in the GUI components (widgets) such as views (subviews). To save memory, Android limits the memory footprint of GUI components by recycling views (subviews) that move out of the screen, and the garbage collector assumes that no long-term references will be kept, so it also releases the loaded bitmap. Thus, each time the views (subviews) return to the screen, the pictures have to be reprocessed, which needs extra CPU/GPU cycles for image decoding and transformation. Since image decoding is slow, to make sure that the image can be loaded quickly and smoothly, one had better avoid

[13]https://github.com/k3b/APhotoManager/issues/94

```
1  public class RecentAdapter{
2    public View getView(){
3      ThumbnailFile.load(true);
4    }
5  }
6  public class ThumbnailFile extends File{
7    private Bitmap load(final boolean raw){
8      if(this.exists()){
9        final Bitmap stored = BitmapFactory.decodeFile(this.getPath
             ());
10       if (stored != null){
11         return raw ? stored : paint(stored);
12       }
13     }
14     return getDefaultThumbnail();
15   }
16 }
```

Fig. 6. A reported defect (issue report #233) of repeated decoding in a real-world app *Document Viewer* (version 2.7.9).

repeatedly processing these pictures each time they return to the screen[14]. To this end, cache can be used, which allows the component to quickly reload the decoded image (in-memory object). Otherwise, it will lead to a negative impact on the app performance (cf. Anti-pattern 4).

*Anti-pattern 4 (**Repeated Decoding Without Caching**):* When image loading is implemented in the components' callback functions (e.g., `getView()`, `onDraw()`, `onBindViewHolder()`, `getGroupView()`, `getChildView()`), if the image is not cached, the image will be repeatedly decoded each time when the callback function is executed, which not only causes GUI lagging but also requires many extra CPU/GPU cycles for image decoding and processing.

We have the following ways to determine repeated decoding without caching:

- The image frameworks Android native APIs (except `ImageView.setImageURI()`), `Picasso`, and `Glide`), implement image decoding and caching by two separate methods, respectively. In the inter-procedural control flow graph of the app, if the image decoding statement is not post-dominated by a corresponding image caching statement, then the image is not cached. Please refer to the last column of Table II for the caching APIs of different image frameworks.
- For the image framework of `Universal-Image-Loader`, both resizing and caching are controlled by the parameter of the decoding method `displayImage(DisplayImageOptions)`. If the decoding statement does not have the argument, then the image is not cached. Otherwise, we further check whether the argument has been set to open the caching switch via `option.cacheInMemory()`. If `displayImage(option)` is not dominated by `option.cacheInMemory()`, then the image is not cached after decoded.

[14]https://developer.android.com/topic/performance/graphics/cache-bitmap

```
1  private class a extends SimpleOnGestureListener{
2    protected void onDraw(Canvas canvas){
3      if(this.H != -1){
4        a((com.byox.drawview.a.a) this.F.get(this.H), this.r);
5      }
6    }
7    private void a(com.byox.drawview.a.a aVar, Canvas canvas){
8      canvas.drawBitmap(BitmapFactory.decodeByteArray(aVar.l(), 0,
           aVar.1().length), aVar.k(), null);
9    }
10 }
```

Fig. 7. A reported defect (issue report #14) of image decoding in UI thread in a real-world app *Quick-draw-everywhere* (version 1.9).

- The `Fresco` framework caches images by default, which cannot be changed. Hence, repeated decoding without caching cannot occur to the `Fresco` code. For the Android native decoding API `ImageView.setImageURI()`, there is neither the corresponding caching method nor the parameter to control the image caching. Thus, the images decoded with `ImageView.setImageURI()` in those callback functions must suffer from repeated decoding.

Figure 6 presents a defect of repeated decoding without caching[15], and our approach is able to effectively detect it: each time the callback `getView()` is triggered, the statement at Line 3 (Line 9) will load (decode) the image repeatedly without caching through `LruCache.put()`.

E. Detecting Image Decoding in UI Thread

It is suggested that time-consuming operations such as loading pictures should not be performed in the UI (main) thread [17], [26], because loading bitmaps in the UI thread could reduce the performance of the app[16], resulting in GUI lagging, longer response time, and even ANR (application not responding) (cf. Anti-pattern 5). Therefore, it is better to load pictures in background threads.

*Anti-pattern 5 (**Image Decoding in UI Thread**):* When image is loaded in the UI thread instead of the background thread, the app performance could be reduced, resulting in GUI lagging, longer response time, and even ANR (application not responding).

To avoid defects (instances) of Anti-pattern 4, apps can load images in the UI thread with cache, whereas to avoid defects of Anti-pattern 5, apps should not load images in the UI thread no matter whether cache is used.

For the frameworks such as `Universal-Image-Loader`, `Glide`, and `Fresco`, their image decoding methods all load images asynchronously (i.e., in background threads). Therefore, this kind of defects cannot happen to the code of these image frameworks. For Android native APIs, the image decoding method can be invoked in UI thread and background threads. The `Picasso` framework provides both asynchronous

[15]https://github.com/SufficientlySecure/document-viewer/issues/233
[16]https://developer.android.com/topic/performance/graphics

(`Picasso.with().load()`) and synchronous image decoding methods (`Picasso.with().load().get()`).

For Android image decoding methods and the synchronous image decoding method of `Picasso`, we need to check whether they are invoked in background threads. Our analysis is based on the call graph of the app. In the call graph, if the decoding method can only be reachable via the method of starting a background thread, this indicates that the decoding method can only be invoked in a background thread; otherwise, there must exist at least one invocation chain (a path in the call graph) that leads to the decoding method but does not contain a method of starting a background thread. By back tracking the invocation chain, if we can find the callback function (event handler) of a UI event (to exclude the situation when an image decoding method is encapsulated in a third-party API which is never used by the app), then there is a defect of image decoding in UI thread.

Figure 7 illustrates such a defect[17], and our approach is able to effectively detect it: the image decoding operation at Line 4 (Line 8) is performed in the GUI callback `onDraw()`.

IV. Empirical Evaluation

In this section, we first evaluate our approach by an experiment on 21 open-source apps with 45 previously-known image loading defects[18], and then conduct a large-scale empirical study on 1,000 real-world commercial apps[19]. Our experimental evaluation and empirical study aim at answering the following research questions:

- **RQ1** - *Performance of IMGDroid*: What are the effectiveness and efficiency of IMGDroid?
- **RQ2** - *Image usage frequency*: Do apps frequently load and display images to fulfil their functionalities? And, what are the usage frequencies of different image frameworks for Android apps?
- **RQ3** - *Pervasiveness of image loading defects*: Are image loading defects common in commercial apps?
- **RQ4** - *Dominant image loading defects*: Among the five kinds of image loading defects, which kind is the most prevalent?
- **RQ5** - *Distribution of image loading defects*: How are the five kinds of image loading defects distributed in different image frameworks?

We implement IMGDroid in Java and disclose it as an open-source tool. We use IMGDroid to analyze real-world apps on a computer with an Intel Core i7 3.6GHz CPU and 32 GB of memory, running Windows 10, JDK 1.8, and Android 7.1.1.

A. Experiment on Benchmarks

To evaluate the effectiveness and efficiency of IMGDroid, we first conduct an experiment on a benchmark which contains 21 open-source Android apps (cf. Table III). The 21 apps are available from F-droid[20] and their source code is maintained in GitHub except that *Leafpic* is maintained in GitLab[21]. In the 21 apps, there are totally 45 previously-known image loading defects, which were either confirmed by the developers or provided with crash report by the app users. The distribution of the 45 defects is summarized in Table III: For example, "2(#234, #269)" in the second row, fourth column, represents that there are two known defects (instances) of Anti-pattern 2 (AP2 for short in Table III) in the app *Kontalk*, where #234 and #269 refer to the corresponding issue reports in GitHub or GitLab.

We apply IMGDroid to these 21 apps. IMGDroid successfully reproduces (finds) all the 45 previously-known defects in about 2,469 seconds (on average 117.6 seconds for each app), and additionally finds 15 previously-unknown defects which are also shown in Table III: For example, "1(#69)+2" in the sixth row, ninth column, represents that besides the previously-known defect (#69) of Anti-pattern 5 (AP5 for short in Table III), IMGDroid additionally detects two new defects of Anti-pattern 5 in the app *OpenNoteScanner*. We manually inspected the code of the newly-found defects and confirm that these 15 new defects are all true positives.

We further find that 5 of the 15 newly-found defects have been fixed in the latest app versions. Thus, we have recently reported the remaining 10 defects to the corresponding developers, and 3 have been confirmed so far. The other 7 defects have not received feedback yet because the projects have not been updated for a long time.

It is worth mentioning that TAPIR [3] is also a static analysis tool that can detect the defects of Anti-patterns 2, 4, and 5, whereas defects of Anti-patterns 1 and 3 are not considered. Since TAPIR is not open-source or publicly available, we cannot apply it to the 21 apps to compare with our approach. However, from their paper [3] the experimental results on the first 10 apps, as shown in Table III indicate IMGDroid is significantly more effective than TAPIR with respect to soundness: while TAPIR can successfully detect all 29 defects in these 10 apps, it does not find the 11 new defects of Anti-patterns 2, 4 and 5 that are found by IMGDroid. These 11 defects reported by IMGDroid are all manually confirmed as true positives. In addition, since IMGDroid covers all image loading operations (statements), and for each image loading statement, IMGDroid examines all possible defects along all potential paths, IMGDroid has a higher coverage than TAPIR. Other reasons for defects missed by TAPIR include path insensitivity and inaccurate reasoning. Meanwhile, IMGDroid is as precise as TAPIR: according to their paper, TAPIR does not report false positives on these 10 apps, whereas in our experiments IMGDroid does not report false positives on all the 21 apps. The reason why IMGDroid is precise lies in that: 1) The image loading procedure is relatively simple and thus is usually implemented in simple control flow. 2) Image loading can be regarded as a read operation and is suggested to be implemented in a single background thread, and thus there is little non-determinacy.

[17] https://github.com/rosenpin/quick-draw-everywhere/issues/14
[18] https://zenodo.org/record/4392284
[19] https://zenodo.org/record/4392292
[20] https://f-droid.org
[21] https://gitlab.com

TABLE III
BENCHMARKS AND EXPERIMENTAL RESULTS: THE NUMBER IN CYAN REPRESENTS THE NUMBER OF NEWLY FOUND DEFECTS, AND EACH EMPTY CELL REPRESENTS THAT THERE IS NO PREVIOUSLY-KNOWN OR NEWLY FOUND DEFECTS

App name	LOC	#Defects of AP1	#Defects of AP2		#Defects of AP3	#Defects of AP4		#Defects of AP5	
			IMGDroid	TAPIR		IMGDroid	TAPIR	IMGDroid	TAPIR
Kontalk	19.6K		2(#234, #269)+1	2				1(#789)+1	1
Qksms	3.5K		2(#718, #719)	2		2(#718, #719)	2	2(#718, #719)+1	2
Document Viewer	49.6K					1(#233)	1	2(#233)	2
Owncloud	49.1K		3(#1862)	3		2(#1862)	2	1(#1862)	1
OpenNoteScanner	2.7K		2(#12)+1	2				1(#69)+2	1
The blue alliance	31.4K		1(#588)	1					
MoneyManagerEx	30.9K		1(#938)	1				+3	+0
Collect	1.4K		1(#2020)	1		+1	+0	+1	+0
AmazeFileManager	3.5K		1(#577)	1					
Tusky	5.7K		4(#1043)	4					

App name	LOC	#Defects of AP1	#Defects of AP2	#Defects of AP3	#Defects of AP4	#Defects of AP5
Quick-draw-everywhere	6.6K				1(#14)	1(#14)
Droid-comic-viewer	56K		1(#13)+2			
CameraView	45.4K	1(#252)				
Clover	68.5K		1(#745)			
Leafpic	78.6K		1(#608)+2			
A photo Manager	19.1K			1(#94)	1(#74)	
Scarlet-Notes	96.8K		1(#166)			
ImagePicker	23.8K	1(#49)	1(#129)			
GlideImageView	41.4K		1(#51)			
Easy_xkcd	85.5K		1(#135)			1(#107)
PhotoPicker	23.22K		1(#36)	1(#50)		

The experimental results and the conclusion derived from the results are summarized as follows.

Answer to RQ1: IMGDroid successfully detects all the 45 previously-known defects in the 21 apps within about 2,469 seconds, and additionally finds 15 previously-unknown defects which are manually confirmed.
Implication: The static analysis of IMGDroid for detecting image loading defects is efficient, and tends to perform well in terms of both soundness and completeness.

B. Empirical Study on 1,000 Commercial Apps

To investigate image loading defects in practice, we further apply IMGDroid to 1,000 commercial apps randomly selected and downloaded from the app store of Google Play. These 1,000 apps cover various types and functionalities, and with sufficient downloads. IMGDroid scales well on these commercial apps; for example, it only spends 149 seconds in analyzing a large (88M) app. The empirical results are summarized in Table IV and we gain the following interesting findings.

Answer to RQ2: Among the 1,000 apps, 897 (89.7%) involve image decoding and displaying operations. The total numbers (proportions) of apps that use Android native APIs, `Picasso`, `Glide`, `Universal-Image-Loader`, and `Fresco` are 854 (85.4%), 78 (7.8%), 83 (8.3%), 45 (4.5%), and 0 (0%), respectively.
Implication: Image loading APIs are broadly used in Android apps. Among the image frameworks, Android native APIs are the most frequently used, whereas `Fresco` has been seldom used.

The answer and implication of RQ2 indicate that image loading and displaying is one fundamental functionality of many commercial apps no matter whether the apps are media-intensive.

For the 1,000 apps, since there are no previously-known image loading defects reported, we used a substantial manual inspection on 2% of the 1,000 apps to improve our confidence on the results generated by IMGDroid: We first randomly selected 20 apps from the 1,000 apps, where they are large popular apps in different categories (e.g., *BBC News*, *Google Assistant*, *Google PDF Viewer*, *Amazon*, *Airbnb*, *All Football*, *Sogou Explorer*, *Tiktok Live Photo*, *360 Security*). Then, we transformed each APK file into Java code based on the reverse engineering tools (dex2jar[22] and jd-gui[23]). Finally, we manually inspected the defects in these 20 apps found by IMGDroid, and confirmed that the detected defects are true positives. Moreover, we did not manually find other image loading defects in the 20 apps. With this, we assume that IMGDroid performs well on all the 1,000 apps and the answer to RQ3 is trustworthy.

Answer to RQ3: Surprisingly, there are totally 5,514 image loading defects found in the 865 of the 1,000 Android apps. Only 32 apps that involve image loading operations do not have such defects. For the 865 apps, each app has on average 6.37 image loading defects.
Implication: Image loading defects are common and severe in real-world commercial apps. Such defects have not gained extensive attention from the app developers.

[22]https://sourceforge.net/projects/dex2jar
[23]http://jd.benow.ca

TABLE IV
EMPIRICAL RESULTS ON 1,000 RANDOMLY SELECTED ANDROID APPS

Framework	#Apps that use	#Defects of AP1	#Defects of AP2	#Defects of AP3	#Defects of AP4	#Defects of AP5
Android native APIs	854	178	2,437	145	310	2,182
`Picasso`	78	0	255	0	0	0
`Glide`	83	0	3	2	0	0
`Universal-Image-Loader`	45	0	2	0	0	0
`Fresco`	0	-	-	-	-	-

The answer to RQ3 implies that many developers are not aware of the inefficient programming practices relevant to image loading (i.e., anti-patterns 1-5), or at least they do not take such defects seriously. The performance and quality of the apps may be easily undermined when the developers despise such defects.

Answer to RQ4: The numbers (proportions) of defects of Anti-pattern 1-5 in the 1,000 apps are 178 (3.23%), 2,697 (48.91%), 145 (2.67%), 310 (5.62%), and 2,182 (39.57%), respectively.
Implication: Image decoding without resizing (Anti-pattern 2) is the dominant (the severest) kind of image loading defects in practice, whereas local image loading without permission (Anti-pattern 3) is the least common kind of image loading defects.

The reason why the number of the defects of Anti-patterns 1 and 3 are small is that such defects are relatively easy to trigger app crashes, and thus are more likely to be manifested before on the market. Since the defects of Anti-patterns 2, 4, and 5 do not immediately lead to app crashes, developers may not find them.

Answer to RQ5: Among the 854 apps that use Android native APIs for image loading, 5,252 image loading defects are found; among the 78 apps that use `Picasso`, 255 such defects are found; among the 83 apps that use `Glide`, 5 such defects are found; among the 45 apps that use `Universal-Image-Loader`, only 2 such defects are found.
Implication: The usage of Android native APIs has the highest possibility to involve image loading defects, whereas the usage of `Universal-Image-Loader` has a lowest possibility to involve image loading defects (`Fresco` is not considered here).

The answer to RQ5 indicates that although Android native APIs are encouraged to be used in app development, they are more easily to be used improperly, and thus more likely to introduce image loading defects. On the other hand, the third-party image frameworks such as `Universal-Image-Loader` can be used conveniently with a relatively low chance to involve image loading defects.

C. Threats to Validity

In this part, we discuss the major threats to validity of our experimental results.

Construct validity. We use 21 open-source apps with 45 known image loading defects to evaluate the effectiveness and efficiency of our approach IMGDroid. The threats to construct validity come from the following two aspects. First, these 45 defects are all relevant to the use of Android native APIs, and their distribution over the five categories of image loading defects (i.e., Anti-pattern 1-5) is not balanced (i.e., the number of known defects of Anti-pattern 1 and Anti-pattern 3 is relatively small). Second, since TAPIR is not publicly available, to compare IMGDroid with TAPIR, we only use its reported results on the first 10 apps in Table III from their paper [3]. With the same reason, we also cannot compare the runtime overhead of the two approaches. Consequently, the comparison between IMGDroid and TAPIR is insufficient.

External validity. Although IMGDroid successfully rediscovers the 45 defects in the 21 open-source apps and additionally finds 15 previously-unknown defects which are manually confirmed, it does not indicate that IMGDroid does not yield false positives in practice; for instance, IMGDroid may report false positives in dead code. Moreover, the 21 apps are not so complicated, and few of them utilize third-party image frameworks. Therefore, when IMGDroid is applied to large commercial Android apps, the experimental results and conclusion in Section IV-A may not be generalized to those commercial apps.

V. RELATED WORK

In this section, we review related work on analyzing the performance issues in Android apps.

A. Image Loading Inefficiency Analysis

We first concentrate on the related work on analyzing the performance issues of image loading and displaying in Android apps [27], [2], [1], [3].

Wang and Rountev [27] propose an analysis method that characterizes the response time as a function of the size of potentially expensive resources (eg, bitmaps). By scaling these resources, they get a responsiveness profile for each GUI-related callback. Based on the evaluation, although they conclude that many operations could be safely left in the main thread, they also show that the response time of image loading increases significantly as the image size increases.

Carette et al. [2] employ their tool HOT-PEPPER to investigate the impact of three picture smells (bad picture format, compression, and bitmap format) in some sample Android apps, and find that large pictures consume a lot of memory and may severely affect the performance of Android apps. They also find that the use of optimized JPG pictures

with the Android default bitmap format is the most energy-efficient combination in Android apps.

Liu et al. [1] develop a static analysis to detect performance defects in Android apps. However, their approach is not focused on image loading and can only find one type of image loading defects, i.e., loading images in the UI thread.

The recent study by Li et al. [3] is most relevant to our work. The authors first conduct an empirical study to summarize several anti-patterns of inefficient image displaying in Android apps. Based on the anti-pattern rules (lacking the full path sensitivity), they implement a light-weight static analyzer, TAPIR, to automatically detect instances of such anti-patterns. Compared with their study, we go further by considering another two anti-patterns of image loading that can significantly degrade the performance and quality of apps, including Anti-patterns 1 and 3. Besides Android native APIs, we also consider four popular third-party image frameworks, including `Universal-Image-Loader`, `Picasso`, `Glide`, and `Fresco`, and our tool IMGDroid can find image loading defects of the code written by both the Android native APIs and the image frameworks.

B. App Energy Defect Analysis

Since energy is a major concern in app performance analysis and green software engineering [28], [29], [30], energy defects in Android apps receives a significant attention [31], [32], [33], [34], [35], [36], [37].

Lots of work focuses on the energy (performance) defects caused by the improper use of different hardware resources (e.g., CPU, screen, GPS, sensors, camera, etc.) in Android apps [31], [32], [38], [34], [39], [36]. Some work uses static analysis to detect these energy defects. For example, a static analysis approach is proposed to detect GUI-related energy-drain defects relevant to the use of energy-intensive resources (e.g., GPS) [34]. Pathak et al. [31] study the so-called no-sleep bugs that arise from the mis-handling power control Android APIs that use different resources of the mobile device. Their tool uses the reaching definition analysis to detect the no-sleep bugs. Wu et al. [39] point out that missing release operations on the requested resources could lead to app performance degradation or even system crash, and they develop a static analyzer Relda2 for detecting such resource leak.

Other work uses testing and dynamic analysis to detect such energy defects. For instance, based on the measurement of the power consumption, Banerjee et al. [16] present a testing approach to detecting energy defects in Android apps. However, the testing approach is expensive and time-consuming, because it has to measure the power consumption at runtime. To reduce the testing cost, Jabbarvand et al. [40] propose an approach to minimizing the test suite for detecting energy defects. GreenDroid [38] is a dynamic analyzer that can automatically diagnose energy defects caused by missing deactivation of sensors or wake locks and cost-ineffective use of sensory data. Banerjee et al. [36] then go further by combining both static and dynamic analysis. They develop a framework EnergyPatch to detect, validate, and repair energy defects on the use of energy-intensive resources (e.g., GPS, Wifi). Recently, Jabbarvand et al. [41] present a search-based testing framework COBWEB for energy testing of Android apps. COBWEB utilizes a set of novel models which consider the execution context to generate tests which can effectively manifest energy defects.

Besides resource (hardware) usage, bad programming practices and particular API usage patterns can also result in energy defects. Based on a hardware power monitor, Vásquez et al. [42] mine and analyze energy-greedy APIs, and point out that some anomalous energy consumption is due to suboptimal usage or choice of APIs. Since apps may still consume energy when running in the background, Chen et al. [33] measure and quantify the amount of battery drain by background activities. They present a system HUSH to suppress background activities without impacting the user experience. Song et al. [37] summarize four types of service usage inefficiencies in Android apps that could lead to unexpected energy consumption, including premature create, late destroy, premature destroy, and service leak. They develop a static analyzer ServDroid that can efficiently detect such service usage inefficiencies.

C. App Performance Issue Analysis

We finally review the studies that focus on other important performance issues (e.g., app response delay, GUI lagging) in Android apps [1], [43], [22].

GUI responsiveness. Liu et al. [1] conduct an empirical study on 70 real-world performance issues and summarize three major types of performance defects in Android apps, including GUI lagging, energy leak, and memory bloat. They implement a static analyzer, called PerfChecker, to detect the identified performance defects. Similar tools include PA-PRIKA [18] and ADOCTOR [19]. Yang et al. [44] insert an artificial long delay at typical problematic operations to test whether the app could exhibit poor GUI responsiveness. To improve GUI responsiveness, Lin et al. [17] design a tool that enables developers to extract long-running operations into asynchronous tasks. Considering that UI-triggered asynchronous tasks can still impact GUI performance, Kang et al. [26] develop a tool DiagDroid for Android GUI performance diagnosis. DRAW [45] is designed to diagnose short delays caused by app GUI rendering, which can pinpoint the responsible GUI components and the rendering operations.

Response delay. Mantis et al. [46] combines program analysis and machine learning to estimate the execution time of Android apps under given inputs. Ravindranath et al. [15] develop a system, called AppInsight, to instrument app binaries to automatically identify the critical path (e.g., slow execution paths) in user transactions. ShuffleDog [47] is developed to identify all delay-critical threads that lead to the slow response of user interactions. Zhao et al. [23], [48] reduce the network latency by prefetching and caching HTTP requests in Android apps. There are also other studies that reduce the app response delay by optimizing the app code [49], [50].

VI. Conclusions

Android apps often employ images to fulfil their functionalities and to improve user experience. App developers can use Android native SDK or third-party image frameworks to load and display pictures in their apps. However, since images are usually very big, if they are not properly loaded or processed, the performance and quality of the apps can be degraded severely. In this paper, based on *Android developers* and the issue reports of real-world apps, we have reviewed and summarized the improper programming practices in image loading and displaying, and formulated five anti-patterns of image loading defects. We have also presented IMGDroid, a scalable static analysis technique to automatically detect such defects based on these anti-patterns. Our experimental evaluation on a benchmark of 21 open-source Android apps demonstrates both the effectiveness and efficiency of IMGDroid, and our empirical study on 1,000 commercial Android apps based on IMGDroid shows that image loading defects are common in practice, which should be noticed by the practitioners.

To complement static analysis, the future work could be focused on the dynamic techniques that manifest image loading defects in Android apps.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant No. 61761136003.

References

[1] Y. Liu, C. Xu, and S. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *36th International Conference on Software Engineering, ICSE'14, Hyderabad, India - May 31 - June 07*, 2014, pp. 1013–1024.

[2] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER'17, Klagenfurt, Austria, February 20-24*, 2017, pp. 115–126.

[3] W. Li, Y. Jiang, C. Xu, Y. Liu, X. Ma, and J. Lu, "Characterizing and detecting inefficient image displaying issues in android apps," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER'19, Hangzhou, China, February 24-27*, 2019, pp. 355–365.

[4] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE'12, Cary, NC, USA - November 11 - 16*, 2012, pp. 59:1–59:11.

[5] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for Android apps," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26*, 2013, pp. 224–234.

[6] W. Choi, G. C. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'13, part of SPLASH'13, Indianapolis, IN, USA, October 26-31*, 2013, pp. 623–640.

[7] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of Android applications," in *Proceedings of the 38th International Conference on Software Engineering, ICSE'16, Austin, TX, USA, May 14-22*, 2016, pp. 559–570.

[8] Y. M. Baek and D. Bae, "Automated model-based Android GUI testing using multi-level GUI comparison criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE'16, Singapore, September 3-7*, 2016, pp. 238–249.

[9] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for Android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA'16, Saarbrücken, Germany, July 18-20*, 2016, pp. 94–105.

[10] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of Android apps," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE'17, Paderborn, Germany, September 4-8*, 2017, pp. 245–256.

[11] W. Song, X. Qian, and J. Huang, "EHBDroid: beyond GUI testing for Android applications," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE'17, Urbana, IL, USA, October 30 - November 03*, 2017, pp. 27–37.

[12] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical GUI testing of android applications via model abstraction and refinement," in *Proceedings of the 41st International Conference on Software Engineering, ICSE'19, Montreal, QC, Canada, May 25-31*, 2019, pp. 269–280.

[13] Y. Lu, M. Pan, J. Zhai, T. Zhang, and X. Li, "Preference-wise testing for android applications," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE'19, Tallinn, Estonia, August 26-30*, 2019, pp. 268–278.

[14] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *ICSE'20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July*, 2020, pp. 481–492.

[15] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "Appinsight: Mobile app performance monitoring in the wild," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI'12, Hollywood, CA, USA, October 8-10*, 2012, pp. 107–120.

[16] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting energy bugs and hotspots in mobile apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE'14, Hong Kong, China, November 16 - 22*, 2014, pp. 588–598.

[17] 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, FSE'14, Hong Kong, China, November 16 - 22*, 2014, pp. 341–352.

[18] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the software quality of android applications along their evolution (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE'15, Lincoln, NE, USA, November 9-13*, 2015, pp. 236–247.

[19] F. Palomba, D. D. Nucci, A. Panichella, A. Zaidman, and A. D. Lucia, "Lightweight detection of android-specific code smells: The adoctor project," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER'17, Klagenfurt, Austria, February 20-24*, 2017, pp. 487–491.

[20] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: automating the detection of api-related compatibility issues in android apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21*, 2018, pp. 153–163.

[21] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *Proceedings of the 40th International Conference on Software Engineering, ICSE'18, Gothenburg, Sweden, May 27 - June 03*, 2018, pp. 408–419.

[22] S. Habchi, X. Blanc, and R. Rouvoy, "On adopting linters to deal with performance concerns in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE'18, Montpellier, France, September 3-7*, 2018, pp. 6–16.

[23] 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 40th International Conference on Software Engineering, ICSE'18, Gothenburg, Sweden, May 27 - June 03*, 2018, pp. 176–186.

[24] S. Arzt, S. Rmobilesofthofer, and E. Bodden, "The soot-based toolchain for analyzing android apps," in *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE'17, Buenos Aires, Argentina, May 22-23*, 2017, pp. 13–24.

[25] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14, Edinburgh, United Kingdom - June 09 - 11*, 2014, pp. 259–269.

[26] 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, FSE'16, Seattle, WA, USA, November 13-18*, 2016, pp. 410–421.

[27] 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, MOBILESoft'16, Austin, Texas, USA, May 14-22*, 2016, pp. 48–58.

[28] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?" *IEEE Software*, vol. 33, no. 3, pp. 83–89, 2016.

[29] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, and S. Tarkoma, "Modeling, profiling, and debugging the energy consumption of mobile devices," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 39:1–39:40, 2016.

[30] I. Manotas, C. Bird, R. Zhang, D. C. Shepherd, C. Jaspan, C. Sadowski, L. L. Pollock, and J. Clause, "An empirical study of practitioners' perspectives on green software engineering," in *Proceedings of the 38th International Conference on Software Engineering, ICSE'16, Austin, TX, USA, May 14-22*, 2016, pp. 237–248.

[31] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps," in *The 10th International Conference on Mobile Systems, Applications, and Services, MobiSys'12, Ambleside, United Kingdom - June 25 - 29*, 2012, pp. 267–280.

[32] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in android applications," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE'13, Silicon Valley, CA, USA, November 11-15*, 2013, pp. 389–398.

[33] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby, "Smartphone background activities in the wild: Origin, energy drain, and optimization," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking, MobiCom'15, Paris, France, September 7-11*, 2015, pp. 40–52.

[34] H. Wu, S. Yang, and A. Rountev, "Static detection of energy defect patterns in android applications," in *Proceedings of the 25th International Conference on Compiler Construction, CC'16, Barcelona, Spain, March 12-18*, 2016, pp. 185–195.

[35] R. Jabbarvand and S. Malek, "μdroid: an energy-aware mutation testing framework for Android," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE'17, Paderborn, Germany, September 4-8*, 2017, pp. 208–219.

[36] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury, "Energypatch: Repairing resource leaks to improve energy-efficiency of android apps," *IEEE Trans. Software Eng.*, vol. 44, no. 5, pp. 470–490, 2018.

[37] W. Song, J. Zhang, and J. Huang, "Servdroid: detecting service usage inefficiencies in android applications," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE '19, Tallinn, Estonia, August 26-30*, 2019, pp. 362–373.

[38] Y. Liu, C. Xu, S. Cheung, and J. Lu, "Greendroid: Automated diagnosis of energy inefficiency for smartphone applications," *IEEE Trans. Software Eng.*, vol. 40, no. 9, pp. 911–940, 2014.

[39] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang, "Lightweight, inter-procedural and callback-aware resource leak detection for android apps," *IEEE Trans. Software Eng.*, vol. 42, no. 11, pp. 1054–1076, 2016.

[40] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for Android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA'016, Saarbrücken, Germany, July 18-20*, 2016, pp. 425–436.

[41] R. Jabbarvand, J. Lin, and S. Malek, "Search-based energy testing of android," in *Proceedings of the 41st International Conference on Software Engineering, ICSE'19, Montreal, QC, Canada, May 25-31*, 2019, pp. 1119–1130.

[42] M. L. Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. D. Penta, and D. Poshyvanyk, "Mining energy-greedy API usage patterns in android apps: an empirical study," in *11th Working Conference on Mining Software Repositories, MSR'14, Proceedings, May 31 - June 1, Hyderabad, India*, 2014, pp. 2–11.

[43] M. L. Vásquez, 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, ICSME'15, Bremen, Germany, September 29 - October 1*, 2015, pp. 352–361.

[44] S. Yang, D. Yan, and A. Rountev, "Testing for poor responsiveness in android applications," in *Proceedings of International Workshop on the Engineering of Mobile-Enabled Systems, MOBS'13, San Francisco, CA, USA, May 25,*, 2013, pp. 1–6.

[45] 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, INFOCOM'17, Atlanta, GA, USA, May 1-4*, 2017, pp. 1–9.

[46] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek, "Mantis: Automatic performance prediction for smartphone applications," in *USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28*, 2013, pp. 297–308.

[47] 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 Trans. Mob. Comput.*, vol. 16, no. 10, pp. 2913–2926, 2017.

[48] Y. Zhao, P. Wat, M. S. Laser, and N. Medvidovic, "Empirically assessing opportunities for prefetching and caching in mobile apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE'18, Montpellier, France, September 3-7*, 2018, pp. 554–564.

[49] 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, MobiSys '15, Florence, Italy, May 19-22*, 2015, pp. 287–300.

[50] Y. Lyu, D. Li, and W. G. J. Halfond, "Remove rats from your code: automated optimization of resource inefficient database writes for mobile applications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'18, Amsterdam, The Netherlands, July 16-21*, 2018, pp. 310–321.