Automatic Performance Testing for Image Displaying in Android Apps

Wenjie Li, Yanyan Jiang, Jun Ma, and Chang Xu
State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China
wenjielinju@gmail.com, \{jyy, majun, changxu\}@nju.edu.cn

Abstract—Image displaying in Android apps is resource-intensive. Improperly displayed images result in performance degradation or even more severe consequences like app crashes. Existing static performance anti-pattern checkers are conservative and limited to a small set of bugs. This paper presents ImMut, the first test augmentation approach to performance testing for image displaying in Android apps to complement these static checkers. Given a functional test case, ImMut mutates it towards a performance test case by either (1) injecting external-source images with large ones or (2) copy-pasting a repeatable fragment and slightly mutating the copies to display many (potentially distinct) images. Evaluation on our prototype implementation showed promising results that ImMut revealed 14 previously unknown performance bugs that are beyond the capability of state-of-the-art static checkers.

I. INTRODUCTION

The Android system facilitates friendly app-user interactions by intuitive user interfaces on which images are frequently displayed [1–3]. However, the CPU/GPU/memory-demanding procedure of image displaying may incur subtle performance bugs that affect the user experiences. Practical impacts of such bugs include performance degradation, resource/energy overuse, or even more severe consequences like app crashes [2–5].

Despite that the research community has studied empirical characteristics of such performance bugs [5–8] and proposed static analyses [6–9] to identify performance bug related anti-patterns, anti-pattern based detectors came with inherent limitations, including:

1) Static analyses fall short on modeling complex dynamic control and data flow. Therefore, the statically checked anti-patterns have to be conservative to reduce false warnings, leading to missed performance bugs [10].

2) Developers simply ignore warnings. As described in our previous work [8], without a concrete manifestation of a performance bug, developers tend to overlook the severity of static warnings even if they can significantly impact performance.

Performance testing for image displaying in Android apps could complement static analyzers because performance test cases may contain developer-verifiable evidence of performance degradation. Unfortunately, Android app developers usually do not have the budget and expertise in writing performance test cases [11–17].

This paper presents the first automatic stress performance testing tool ImMut for image displaying in Android apps. Based on our previous empirical finding [8] that triggering real-world IIDs issues require large or many images, ImMut mutates a developer’s functional test case (usually with a few displayed images) towards image-displaying intensive performance test cases, over the following two independent directions:

1) To display large images, ImMut replaces external-source images with predefined high-resolution ones. This idea extends the previous work [18] for the detection of UI-blocking decoding events.

2) To display many images, ImMut presents a novel algorithm that first explores for useful mutants that can be concatenated to yield displaying potentially new images. Then, ImMut exploits these mutants by making a lot of copies of them (with carefully injected swipe events) to yield an image-displaying intensive performance test case.

Observing that image decoding is on the critical path of image displaying [19], we further designed a performance test oracle for image displaying that checks against under-utilization or misplacements of decoded pixels.

The experimental results show that ImMut well complements static analyzers by revealing 16 previously unknown performance bugs (in a set of 16 apps that were checked by a static analyzer) with zero false positives, out of which 14 are beyond the capability of existing static analyzers. We received developers’ responses for 9 bugs, and all feedback were positive.

In summary, this paper’s major contribution is demonstrating that Android app’s image-displaying performance testing can be automatic and practical. This paper’s technical contribution is a novel explore-exploit test augmentation algorithm for mutating a functional test case into image-displaying intensive performance test cases.

The rest of this paper is organized as follows. Section II provides necessary background on (inefficient) image displaying and motivation for performance testing. Sections III and IV illustrate our ImMut algorithm. Section VI presents the experimental results followed by related work discussions in Section VII. Finally, Section VIII concludes this paper.

II. BACKGROUND AND MOTIVATION

A. (Inefficient) Image Displaying in Android Apps

Displaying images in Android apps is not so simple as an API invocation in Android. Image displaying undergoes a long loading → decoding → transformation → rendering pipeline [8], which is CPU/GPU/memory intensive and performance-critical.
Developers should carefully manage the life-cycle of displayed images and keep the app responsive.

We refer Image-displaying related performance bugs as *Inefficient Image Display (IID) issues* [8]. Even well-maintained popular apps frequently contain IID issues caused by anti-image-displaying patterns (e.g., repeated, redundant, or UI-blocking image displaying or improperly recycled images), leading to performance degradation or app crashes. Image displaying libraries (e.g., Glide [20] or Fresco [21]) can alleviate image-related performance issues. However, apps may still misconfigure these libraries and suffer from performance degradation [22].

Figure 1 displays the manifestation of a typical IID issue in NewsBlur. The display of thumbnails is significantly delayed after browsing a few screens of news. The root cause is erroneously decoding images with their original resolutions (even if only thumbnails are displayed), resulting in overwhelmed image decoding requests.

### B. Automatic Detection of Inefficient Image Displaying Issues

Image displaying has long been identified as a major source of Android app’s performance bugs [5–7, 18, 23]. However, IID issues were systematically studied only recently. For example, [8, 9] facilitates automatic static IID issue detectors that report program paths satisfying anti-pattern rules like “decoding without reszing”, “duplicate decoding”, and “decoding in the UI thread”, etc.

However, static analyses inherently fall short on reasoning about the complex dynamic behavior of app execution. Android apps frequently use third-party libraries, event queue and callbacks, reflections, and inter-component communications, which all incur non-trivial control and data flows. It is extremely difficult (if not possible) to precisely determine the sequence of image-displaying API invocations and their parameters.

Consequently, existing static analyzers [6, 8, 9] have to adopt the compromise that only conservative anti-patterns are checked to keep false-positive rates reasonable. (This is why the aforementioned patterns looked simple.) The downside is also obvious: they can only identify a limited fraction of real IID issues, leaving performance bugs buried in the apps.

Even if the IID issue in Figure 1 is caused by “decoding without reszing”, existing static analyzers fail to find it because the developers did try to downsample the image using a series of non-trivial computations, but maxDimPX was erroneously set to Integer.MAX_VALUE (the highlighted data flow in Figure 2).

Static analyses cannot effectively model the runtime values of decodeOpts and have to recognize this case conservatively as “resized before decoding”.

### C. Motivation for Test Augmentation

Though difficult to statically analyze, the triggering of the IID issue in Figure 1 has a simple structure: a single action (swipe to show more news) is repeatedly performed. Our previous empirical observation [8] that the necessary condition for triggering IID issues is displaying large and/or many images naturally motivates dynamic IID issue detection by augmenting functional test cases:

1) Replacing external-source images with large ones at runtime can magnify the consequences of IID issues.

2) Copy-pasting any image-displaying related event fragment (with slight mutations, e.g., clicking a sibling widget) may yield displaying reasonably many images and trigger an IID issue.

The above two simple yet powerful ingredients are the foundation of our ImMut approach. For the IID issue in Figures 1 and 2, any swipe-up event on the news list in a functional test case will be considered as related to displaying external-source (Web) images by an API trace analysis. Repeating such swipe-ups will manifest the IID issue. Our ImMut revealed this previously unknown issue, and the bug was quickly fixed after our report.

### III. ImMut Overview

Though functional test cases are generally available to app developers (e.g., for regression testing [24]), effectively aug-
menting them for manifesting IID issues remains a challenge. Particularly, the following two fundamental questions should be answered:

1) How to efficiently mutate a functional test case towards displaying large/many images (and thus IID issue exposure)? Theoretically, one can exhaustively enumerate all possible mutants until IID issues are manifested. However, brute-force enumeration is simply impractical.

2) What should be the runtime evidence of inefficient image displaying? Not all IID issues result in app not-responding or crash. An effective test oracle should be designed to capture inefficient image displaying behaviors that may cause performance degradation.

The rest of this section provides an overview of the key design of ImMut. Technical details are discussed in Section IV.

### Injecting Large Images

All images are decoded before display. In the performance-testing mode for a functional test case, ImMut hooks the 9 image decoding APIs [8, 25] and considers any byte[], InputStream, FileDescriptor, or an external String URL/filename as external-source (in-app static images are referenced via a Resource ID). Any sufficiently large external-source image is considered to be potentially arbitrarily large (and thus external-source static images like icons are filtered out). These images are automatically replaced with our prepared large images.

**Explore-exploit to Display Many Images.** ImMut tries to “copy-paste” an event fragment (a fragment of consecutive events) in the functional test case to display many images. To conduct effective copy-pastes, ImMut first explores a given functional test case’s mutants by changing an event’s receiver to its sibling widget or injecting a swipe to a container widget. Then we validate whether such a mutant can be a basic construct for displaying many images using the following criteria:

1) The mutated test case has a similar execution trace as the original test case, i.e., the mutated event or injected swipe does not break the test case semantics; and

2) The mutated test case displays a different image, such that concatenating the mutants has the potential to display more images.

Then, we exploit these mutated and injected events by copy-pasting a repeatable fragment in the test case (an event fragment that starts from and ends with an identical GUI state) for as many times, and replace the pasted fragments with explored event mutants. For each repeatable fragment, we generate a single long performance test case, which is run and checked against our IID test oracle.

**Decoding-based Test Oracle.** Image decoding is on the critical path of image displaying. Therefore, we trace the image decoding API invocations (and the life-cycle of decoded bitmaps) and design test oracle around our empirical findings in [8] that the root causes of most IID issues are under-utilization of decoded pixels and improperly placed image displaying. The performance bug indicators are:

- Repetitive decoding a same image close in time. This usually happens when images are not properly cached.
- Decoding a significant amount of pixels in the UI thread. This is a major source of app lagging and app not responding.
- Decoded images in the heap are accumulating boundlessly. This correlates to memory leak and misconfigured cache.

### IV. PERFORMANCE TESTING FOR IMAGE DISPLAYING IN ANDROID APPS

#### A. Notations and Definitions

An Android app can be regarded as an event-driven state transition system. Starting from the app’s initial state $s_0$, executing a test case (an event sequence) $t = [e_1, e_2, \ldots, e_n]$ yields execution trace

$$Tr(t) = \left[ s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 \xrightarrow{e_3} \ldots \xrightarrow{e_n} s_n \right]$$

in which each state $s_i$ ($0 \leq i \leq n$) contains a set of tree-structure GUI widgets. In particular, given a state $s_{i-1}$ in $Tr(t)$, “executing $e_i$” (yielding $s_{i-1} \xrightarrow{e_i} s_i$) denotes performing $e_i$’s designated action (e.g., a click) on $e_i$’s receiver widget (denoted by $e_i$, recv’s XPath [26] in which each selection step identifies exactly one widget, i.e., the XPath contains no wildcards like * and /*. Executing $e_i$ may fail when the XPath has no correspondence in the GUI layout tree of $s_{i-1}$, yielding failed states

$$s_i = s_{i+1} = s_{i+2} = \ldots = s_n = \perp.$$

#### B. Injecting Large Images

Images may be displayed during the execution of event $e_i$. We hook the image decoding APIs and replace the input parameter of the API with one of a few predefined large images. Particularly:

1) For decodeFile(), setImageURI(), createFromPath(), and setImageViewUri() that decode an external path or URI, we safely replace the image with predefined large ones.

2) For decodeFileDescriptor(), decodeStream(), createFromStream(), decodeByteArray(), and decodeRegion() that decode a stream or byte array, we optimistically assume they are from external-source. To reduce false positives, if all images decoded for a call site in the “many-image” performance test case are less than 1Mpixels, we do not report this call site as a bug candidate.

Replacing an image may result in inconsistency in program logic (e.g., the code may make assumptions about the sizes of decoded images). However, we did not observe such hazards in our experiments.

#### C. Mutation Towards Many Images

**Explore.** The exploration phase identifies potentially useful event mutants for a functional test case $t$ and creates a labeled transition system whose traversal may result in test cases that display many images.

The first exploration (mutation) direction is towards clicking a sibling widget (e.g., the next element in a grid). Given two widgets $w_1$ and $w_2$ in a GUI layout tree, $w_1$ and $w_2$ are siblings
only if they share a structurally similar path to the root in the layout tree. Specifically, given a widget \( w \), let \( \rho(w) \) be the list of all ancestor widgets of \( w \) in the layout tree sorted in the depth order. Widgets \( w_1 \) and \( w_2 \) are siblings only if

\[ \{w_i.\text{type} \mid w_i \in \rho(w_1)\} = \{w_i.\text{type} \mid w_i \in \rho(w_2)\}. \]

Given \( s \rightarrow s' \) in \( Tr(t) \), \( \mu(e) \) returns an event that performs the same action as \( e \) but on the next sibling of \( e.\text{recv} \) in the in-order traversal sequence of all widgets in the GUI layout tree of state \( s \).

We try to mutate each event \( e_i \in t \) (1 \( \leq i \leq n \)) for as many times \( (k = 1, 2, \ldots) \) until all siblings are enumerated, creating event mutants

\[ \hat{e}_i^k = \mu(e_i) \mu(e_i) \ldots \mu(e_i). \]

Intuitively, \( \hat{e}_i^k \) (on its \( k \)-th sibling) is an event that performs the same action of \( e \) but on \( e.\text{recv} \)'s sibling widget of exactly \( k \)-distance away. Correspondingly, we generate a mutated test case by replacing \( e_i \) with \( \hat{e}_i^k \), yielding

\[ \hat{t}_i^k = [e_1, e_2, \ldots, e_{i-1}, \hat{e}_i^k, e_{i+1}, \ldots, e_n]. \]

The second exploration (mutation) direction is towards injecting a swipe-up/down to yield displaying of more images. Specifically, we try to inject a scroll event \( \hat{e}_i \) of container-height length before \( e_{i+1} \) to update all elements in a scrollable container, yielding

\[ \hat{t}_i^\parallel = [e_1, e_2, \ldots, e_{i-1}, e_i, \hat{e}_i^\parallel, e_{i+1}, \ldots, e_n]. \]

We also add \( \hat{e}_i^\parallel \)'s reset event \( \hat{e}_i^\parallel \) as an event mutant, which conducts a long swipe that resets the scrollable container to its initial position.

All mutated test cases \( \hat{t} \) (\( \hat{t}_i^k \), \( \hat{t}_i^\parallel \), or \( \hat{t}_i^{\parallel} \)) are executed (“explored”) and checked against the following three conditions:

1) The execution of \( \hat{t} \) is successful, i.e., the trace

\[ Tr(\hat{t}) = [s_0 \hat{e}_1 \ldots s_i \hat{e}_i \ldots s_{i+1} \hat{\mu} \hat{e}_i \ldots s_n] \]

satisfies that \( \hat{\mu} \notin \{\hat{s}_i, \hat{s}_{i+1}, \ldots, \hat{s}_n\} \). The underlined \( \hat{e}_i \) denotes the mutated or injected event.

2) All the states \( \{\hat{s}_i, \hat{s}_{i+1}, \ldots, \hat{s}_n\} \) in \( Tr(\hat{t}) \) match their correspondences \( \{s_i, s_{i+1}, \ldots, s_n\} \) in \( Tr(t) \), ensuring that the mutant does not change the test case’s semantics.

3) The execution trace \( Tr(\hat{t}) \) decodes one or more different images compared with \( Tr(t) \).

If all the above conditions are satisfied, \( \hat{t} \)'s corresponding mutated or injected event \( \hat{e} \) should be potentially interesting for performance testing of image displaying. Accordingly, we construct test case \( t \)'s augmentation model, a labeled transition system over graph \( G(V, E) \) defined as follows:

1) For each state \( s_i \) in \( Tr(t) \), there is a vertex \( s_i \in V \) (vertices in Figure 3).
2) For each state transition \( [s_{i-1} \hat{\mu} s_i] \) in \( Tr(t) \), there is an edge \((s_{i-1}, s_i) \in E \) labeled \( e_i \) (black forward edges in Figure 3).
3) For each explored sibling mutant \( \hat{e}_i^{\parallel} \), for each \( k \in \{1, 2, \ldots\} \) there is an edge \((s_{i-1}, s_i) \in E \) labeled \( \hat{e}_i^{\parallel} \) (blue parallel-edges in Figure 3).
4) For each explored swipe mutant \( \hat{e}_j^{\parallel} \) or \( \hat{e}_j^{\parallel} \), there is a self-loop \((s_i, s_i) \in E \) labeled \( \hat{e}_j^{\parallel} \) or \( \hat{e}_j^{\parallel} \), respectively (green and brown self-loops in Figure 3).
5) For each pair of \( s_i, s_j \) (\( i < j \)) that share an identical GUI layout, there is a back-edge \((s_j, s_i) \in E \) labeled \( \hat{e}_j \), denoting that this edge has no associated event (the red back-edge in Figure 3).

**Exploit.** Given any path in \( G \) starting from \( s_0 \), collecting the labels in the traversed edges yields an event sequence (and thus a test case). Theoretically, we can arbitrarily (e.g., randomly) walk on \( G \) and collect a long performance test case for image displaying, based on the following analyses:

Suppose that a back-edge \((s_j, s_i) \in E \) (an \( \epsilon \)-transition) connects two equivalent program states, e.g., a same item-selection menu. There may be multiple \( s_i \rightarrow s_j \) forward paths (path of no \( \epsilon \)-back-edges), but we expect that all such paths are semantically equivalent to \([s_i \hat{e}_{i+1} s_{i+1} \hat{e}_{i+2} \ldots \hat{e}_j s_j]\), because:

1) For a sibling-mutant edge (labeled with \( \hat{e}_i^{\parallel} \), or a blue parallel-edge in Figure 3), the exploration phase has confirmed its interchangeability with \( e_p \).
2) For an injected swipe edge (labeled with \( \hat{e}_i^{\parallel} \), or a green self-loop edge in Figure 3), the exploration phase has confirmed that the swipe yields an identical GUI layout assuming that the container contains sufficiently many widgets.
3) For an injected reset edge (labeled with \( \hat{e}_j^{\parallel} \), or a brown self-loop edge in Figure 3), it simply resets the viewpoint
of a scrollable container and leads to no breaking UI change.

This paper adopts a more systematic exploration on the per-back-edge basis. Specifically, ImMut for each back-edge \((s_i, s_j) \in E\) labeled \(e\) generates a performance test case that exploits all forward paths from \(s_i\) to \(s_j\) by listing them in the “lexical” order on the following recursive definition of

\[
C_{i,j} = \begin{cases} 
S_i \times \{[\epsilon_{i+1}^1], [\epsilon_{i+1}^1], \ldots, [\epsilon_{i+1}^k]\} \times C_{i+1,j}, & i \neq j \\
\{\epsilon\}, & i = j
\end{cases}
\]

in which \(A \times B\) denotes the pairwise concatenation of all lists in ordered sets \(A\) and \(B\):

\[
A \times B = \{a_i::b_j | a_i \in A \land b_j \in B\},
\]

and \(S_i\) denotes all distinct swipe actions that can be performed on state \(s_i\):

\[
S_i = \left\{ \left\{[\epsilon_i^1], [\epsilon_i^2], [\epsilon_i^3], \ldots, [\epsilon_i^k]\right\} \mid (s_i, s_i) \in E \land (s_i, s_i) \notin E \right\}
\]

Following the above definitions, \(C_{i,j}\) consists of a set of event lists, and applying any \(c \in C_{i,j}\) at state \(s_i\) will yield a state equivalent to \(s_i\) that traverses a forward \(s_i \rightarrow s_j\) path on \(G\).

Therefore, we concatenate all event lists in \(C_{i,j}\) and prepend the concatenated event sequence with the prefix \([\epsilon_1, \epsilon_2, \ldots, \epsilon_i]\) in \(t\) (which leads \(s_0\) to \(s_i\)) to form a long performance test case for image displaying, whose execution trace is checked against our test oracle at runtime.

D. Test Oracle for Inefficient Image Displaying

The oracle is designed over the empirical finding that a few root causes cover most (90%) IID issues [8]. The root causes can be roughly categorized as follows:

**Pixel Under-utilization.** Decoded pixels may be wasted by not being displayed. This is usually caused by decoding large images without adaptive downsampling. Determining whether the app conducted proper downsampling is a challenge for both static and dynamic checkers. Static checkers cannot analyze complex control and data flows and ignore any case beyond “decode without resize”, causing many missed bugs (like the motivating example in Figure 2). For dynamic checkers, a taint analysis can link each displayed image to its decoding (and thus know whether decoding is adaptive). However, such a taint analysis is intrusive and expensive.

Alternatively, we observed that an image-decoding site is usually multiplexed for displaying different images of different sizes1, yielding different downsampling ratios. Non-adaptive image decoding is reported when images of different sizes are decoded with the same downsampling ratio. This may sometimes cause a false positive (e.g., when an image is fetched from the Web twice). However, such a warning may still be a helpful performance optimization hint for the developers.

Pixel under-utilization may also happen when images are repetitively decoded, whose root cause is usually improperly managed (or the absence of) cache. We identify all pairs of image-decoding API invocations with identical decoded bitmap. An IID issue is reported if the amount of pixels decoded in between occupies less than 1/16 of the app’s memory limit.

**Improperly Placed Image-displaying.** Image displaying (decoding) may be improperly performed in the UI thread, which may block the execution of subsequent events. Static checkers try to report all provable UI-thread decoding cases, but with both potential false positives (e.g., decoding a small fix-size icon with no performance problem at app start) and false negatives (e.g., a method unreachable in the call graph which can be invoked via inter-component communication). As a dynamic approach, ImMut gives each event handler a “quota” of decoding 1MPixels (sufficiently many pixels to cause sensible performance degradation). Decoding pixels beyond this threshold will be recognized as improper image decoding in the UI thread. When external-source images are injected with large ones, such performance bugs can easily be identified with a minor chance of false positive.

Images may also be misplaced in an unbounded cache or even leak in the heap. We manually inject garbage collection events and observe the memory consumption growth of the performance test case. If the memory-use peak does not flatten over the entire execution of the performance test case (until time out), we also report a warning on unbounded cache or memory leak.

V. Implementation

The ImMut prototype tool is implemented upon Xposed [27] (for instrumenting Android Framework APIs calls for tracing and image injection), UIAutomator [28] (for GUI layout tree dump), and adb [29] (for sending GUI events). Essential implementation details are discussed in this section.

**App State Equivalence.** Our explore-exploit algorithm frequently tests the equivalence of two GUI states \(s\) and \(\hat{s}\). We use the criterion proposed in DetReduce [30]. For each widget \(w\) in a layout tree \(T\), let \(\rho(w)\) be the list of all ancestor widgets of \(w\) in \(T\) sorted in the depth order. We concatenate the attributes (widget type and all registered event handlers) of all ancestors of \(w\) as a list

\[
\pi(w) = [w, attr] \mid \forall w_i \in \rho(w)].
\]

Given layout trees \(T_s\) and \(T_{\hat{s}}\) for \(s\) and \(\hat{s}\), we consider \(s\) and \(\hat{s}\) equivalent if

\[
\{\pi(w) \mid \forall w \in T_s\} = \{\pi(\hat{w}) \mid \forall \hat{w} \in T_{\hat{s}}\},
\]

i.e., \(s\) and \(\hat{s}\) are composed of a “same set” of widgets.

**Replaying a Test Case.** Running a generate test case requires replaying an event sequence. Functional test cases are recorded by translating the coordinates in an event to an XPath in the GUI layout tree. For each test run, ImMut resets the app under

---

1Even if a decoding site is only invoked once, we may still inject different images to the decoding API if it is associated with external-source images.

2If the amount of pixels decoded in between occupies less than the cache size, then the cache must not be totally refreshed since the first invocation. Therefore, the identical bitmap decoded in the second invocation is not (properly) cached. Specially, by manually analyzing source codes of apps (studied by [8]) with reported IID issues, we find the cache size of each app is at least 1/16 of the app’s memory limit, which is used as the threshold here.
test by a fresh reinstall of the app. Events are replayed on their XPaths. During replay, special actions (e.g., login) are hardcoded as predefined rules: when the action’s corresponding GUI layout is detected, a set of predefined events are sent to the app. This is a standard treatment in replaying an Android app’s execution [31–33].

**Injecting Large Images.** A set of images of different sizes is essential in the validation of proper downsampling (Section IV-B). We prepared four images of various size: $960 \times 800$ (0.76MPixels), $1920 \times 1600$ (3.1MPixels), $2835 \times 2120$ (6.0MPixels), $4032 \times 3016$ (12.2MPixels). A large image is helpful for the developer to evaluate the actual performance impact of an IID issue. Image injection is only conducted on the functional test cases and is repeated four times. The injected image is fixed (one of the above four images) in each round of repetition, and all external-source images are replaced.

**Explore-exploit to Display Many Images.** Exploring sibling event mutants $e^k$ for $k = 1, 2, \ldots$ (Section IV-C) can be costly because a widget may have many siblings (e.g., a piece of news in a list), and each $k$ yields a test case to be executed.

ImMut approximates this procedure by randomly sampling a few values $K \subseteq \{1, 2, \ldots\}$. Our implementation samples two widgets (i.e., $|K| = 2$) and explores (executes) $e^k$’s corresponding test cases $i^k$ only for $k \in K$. If the exploration (execution) of any $i^k$ ($k \in K$) fails, we will not consider any sibling mutant $e^k$ in the augmentation model (i.e., drop $e^k$ for all $k$). Otherwise, we add all $e^k$ (for all $k \in \{1, 2, \ldots\}$) to the augmentation model.

Finally, $C_{i,j}$ may contain infinitely many paths due to self-loops (suppose an infinitely long news list in the motivating example). In practice, a scrollable container may hit its bottom. Our implementation did not preprocess all forward paths in $C_{i,j}$. Instead, ImMut dynamically explores the forward paths in $C_{i,j}$ via a depth-first search, which stops to deliver more swipe mutants $e^{\circ}$ or sibling mutants $e^k$ when they are not available on the current GUI layout.

**VI. EVALUATION**

To study the merits of ImMut, our evaluation mainly concerns whether a test-based approach complements existing static analysis tools [8, 9] by revealing more IID issues.

**A. Experimental Setup**

We conduct our experiments on a set of actively maintained apps being well-verified by our static IID issue detector TAPIR [8]. We collected such experimental subjects from the open-source IID data set in [8]. All apps in the data set are related to image-displaying, and each contains at least one user-reported image-displaying performance bug in Github's issue tracking system. All these apps were scanned by TAPIR [8] for IID issues (IID warnings were reported to the developers) and well-suited for our experiments.

Our experiments were conducted on 16 apps from the 36 apps in the data-set by excluding 10 inactive apps (with neither code updates nor responses to issues within 6 months, and thus we are not expecting developer’s responses on detected performance bugs), 8 trivial toy demo apps (all with $<10,000$ downloads, which are not representative for practical apps), and two apps (DocumentViewer and OpenNoteScanner) that cannot start in the emulator (ImMut is implemented on a rooted Android Virtual Device). The selected apps are listed in Columns 1–3 in Table I.

For each app, we asked a recruited post-graduate student to provide a few manual test cases that can cover major functionalities of the app. Then, we ran ImMut to remove image-irrelevant (display no external-source image) or duplicated (the displayed images is a subset of another test case) test cases. The remaining test cases (listed in Column 4 in Table I) are further dynamically analyzed. In our experiments, the exploration time is unlimited and the exploitation has a 30-minute timeout. We merge duplicated bug reports (bug reports with the same image-decoding API invocation site) to obtain a list of unique IID issues. We manually check each of them and report real performance bugs to the developers.

All experiments were performed on a commodity laptop with Intel i7 processor and 16 GB RAM running macOS 10.14 and an emulated Android Virtual Device of Nexus 6 with API Level 28.

**B. Experimental Results**

**Bugs Found.** Given the functional test cases in Table I, ImMut reported 16 unique IID issues as listed in Table II. We manually confirmed all these reported IID issues (by their image decoding/displaying traces) as true positives and enclosed these IID issues into 10 bug reports submitted to the corresponding app’s Github issue tracking system. Developers responded to 9/16 IID issues (other issues are still open) by the time this paper was written, and all of them were confirmed as real performance threats. Two were already fixed by the developers.

One interesting case is that ImMut rediscovered an IID issue (Passandroid-#333) previously reported by TAPIR (Passandroid-#136). The developer considered the static analysis results “lack of evidence of performance degradation” and closed Issue #136 by citing Knuth’s famous quote, “premature optimization is the root of all evil.” However, they confirmed the performance test’s results (displaying a large image) as a real performance bug. This case further validates our claim that a dynamic (testing) approach is more in demand by the developers.

Another interesting case is the fixed bug for Antennapod-#5092. The developers first considered that “Glide (a popular image displaying library) already does in-memory caching” and they “should not influence Glide.” However, a more thorough investigation indicates that the developers created multiple subclass objects of BitmapTransformation but forgot to make them hashable, which essentially disabled the image cache. This case involves complex cross-event data flow and is almost impossible to be precisely identified by a static analyzer.

**Comparisons with Static Analyzers.** We also applied static analyzers TAPIR [8] and IMGDroid [9] to our experimental subjects. They reported considerably more IID issue warnings compared with ImMut. However, nearly all ImMut’s reported

Though these apps were scanned by TAPIR, there may be unconfirmed (open) or newly introduced IID issues.
TABLE I
EVALUATED APPS AND FUNCTIONAL TEST CASES.

<table>
<thead>
<tr>
<th>App Name</th>
<th>Version</th>
<th>LOC</th>
<th>Functional Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conversations</td>
<td>2.9.6</td>
<td>108,079</td>
<td>([#1] Send an image [#2] Send images [#3] Login)</td>
</tr>
<tr>
<td>Kiss</td>
<td>3.15.5</td>
<td>30,277</td>
<td>([#8] Click item)</td>
</tr>
<tr>
<td>Easy xkcd</td>
<td>7.3.9</td>
<td>15,775</td>
<td>([#16] Browse cartoons [#17] Browse items)</td>
</tr>
<tr>
<td>Twidere</td>
<td>4.1.6</td>
<td>152,947</td>
<td>([#18] Login [#19] Open a blog)</td>
</tr>
<tr>
<td>Antennapod</td>
<td>Audio, 50K+</td>
<td>2.1.2</td>
<td>87,071</td>
</tr>
<tr>
<td>MoneyManagerEx</td>
<td>Tool, 100K+</td>
<td>4.1.6</td>
<td>([#30] Open addPodcasts)</td>
</tr>
<tr>
<td>Slide for Redditt</td>
<td>Reading, 100K+</td>
<td>6.6.1</td>
<td>123,514</td>
</tr>
<tr>
<td>Passandroid</td>
<td>Tool, 1M+</td>
<td>3.5.6</td>
<td>([#31] Create count [#32] Browse menu)</td>
</tr>
<tr>
<td>SlideShow</td>
<td>Tool, 10K+</td>
<td>2.9.0</td>
<td>([#33] Browse themes [#34] Browse images [#35] Browse blogs)</td>
</tr>
<tr>
<td>MTG Family</td>
<td>Tool, 500K+</td>
<td>3.6.6</td>
<td>([#36] Add a pass [#37] Edit pass details pass [#38] Add passes)</td>
</tr>
<tr>
<td>Muzei</td>
<td>Theme, 1M+</td>
<td>3.4.3</td>
<td>([#39] Browse images(1) [#40] Browse images(2) [#41] Browse images(3) [#42] Open image list [#43] Open pictureInPicture)</td>
</tr>
<tr>
<td>WordPress</td>
<td>Internet, 10M+</td>
<td>16.3</td>
<td>438,601</td>
</tr>
</tbody>
</table>

TABLE II
REAL IMAGE-RELATED PERFORMANCE BUGS DETECTED BY IMMut.

<table>
<thead>
<tr>
<th>ID</th>
<th>App</th>
<th>Issue</th>
<th>Status</th>
<th>Manifest</th>
<th>Root Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CycleStreets</td>
<td>#463</td>
<td>Open</td>
<td>✓ ✓ ✓ ✓</td>
<td>Decoding a 3.0MPixel icon in the UI thread when loading app’s home screen</td>
</tr>
<tr>
<td>2</td>
<td>Newsblur</td>
<td>#1438</td>
<td>Fixed</td>
<td>✗ ✓ ✓ ✓</td>
<td>Image leak in app’s native heap (unbounded memory growth on swiping the news list)</td>
</tr>
<tr>
<td>3</td>
<td>Newsblur</td>
<td>#1441</td>
<td>Confirm</td>
<td>✗ ✓ ✓ ✓</td>
<td>Repeated (and redundant) decoding of high-resolution app icon in the splash screen</td>
</tr>
<tr>
<td>4</td>
<td>Newsblur</td>
<td>#1450</td>
<td>Open</td>
<td>✓ ✓ ✓ ✓</td>
<td>Same decoded images as above (#4). The decoding happened in the UI thread</td>
</tr>
<tr>
<td>5</td>
<td>Newsblur</td>
<td>#1450</td>
<td>Open</td>
<td>✓ ✓ ✓ ✓</td>
<td>Same decoded images as above (#4). The decoding happened in the UI thread</td>
</tr>
<tr>
<td>6</td>
<td>Antennapod</td>
<td>#5092</td>
<td>Confirm</td>
<td>✗ ✓ ✓ ✓</td>
<td>Repeated (and redundant) reloading and decoding of an entire grid of 12 podcast thumbnails when adding a podcast</td>
</tr>
<tr>
<td>7</td>
<td>Antennapod</td>
<td>#5092</td>
<td>Fixed</td>
<td>✗ ✓ ✓ ✓</td>
<td>Repeated (and redundant) reloading and decoding of a podcast’s high-resolution icon each time the podcast is opened</td>
</tr>
<tr>
<td>8</td>
<td>Slide4Redditt</td>
<td>#3355</td>
<td>Open</td>
<td>✓ ✓ ✓ ✓</td>
<td>Decoding a 3.2MPixel icon on app start in the UI thread</td>
</tr>
<tr>
<td>9</td>
<td>Slide4Redditt</td>
<td>#3355</td>
<td>Open</td>
<td>✓ ✓ ✓ ✓</td>
<td>The above image (#8) is decoded without resizing</td>
</tr>
<tr>
<td>10</td>
<td>Slide4Redditt</td>
<td>#3355</td>
<td>Open</td>
<td>✓ ✓ ✓ ✓</td>
<td>Multiple 2.1MPixel icons are decoded without resizing when displaying the blog list</td>
</tr>
<tr>
<td>11</td>
<td>Slide4Redditt</td>
<td>#3355</td>
<td>Open</td>
<td>✗ ✓ ✓ ✓</td>
<td>Swiping down the blog list (#10) yields repeated decoding of the same set of icons</td>
</tr>
<tr>
<td>12</td>
<td>Passandroid</td>
<td>#333</td>
<td>Confirm</td>
<td>✗ ✓ ✓ ✓</td>
<td>Repeatedly decoding of external storage images on selecting a same image</td>
</tr>
<tr>
<td>13</td>
<td>Passandroid</td>
<td>#333</td>
<td>Confirm</td>
<td>✗ ✓ ✓ ✓</td>
<td>Images load from external storage is not downsampled when creating a PassBook</td>
</tr>
<tr>
<td>14</td>
<td>Passandroid</td>
<td>#333</td>
<td>Confirm</td>
<td>✗ ✓ ✓ ✓</td>
<td>The above image decoding (#13) is performed in the UI thread</td>
</tr>
<tr>
<td>15</td>
<td>Muzei</td>
<td>#717</td>
<td>Confirm</td>
<td>✓ ✓ ✓ ✓</td>
<td>An image (from external storage) was redundantly read and decoded three times when setting an app’s home screen</td>
</tr>
<tr>
<td>16</td>
<td>WordPress</td>
<td>#14392</td>
<td>Confirm</td>
<td>✗ ✓ ✓ ✓</td>
<td>Images in blog posts are improperly cached and repeatedly decoded each time a blog post is opened</td>
</tr>
</tbody>
</table>

For the bug manifestation, “F” denotes bug triggered by the functional test case; “L” denotes bug triggered by injecting large images; “M” denotes bug triggered by the long performance test case generated by explore-exploit; “S” denotes that this bug was reported by static analyses (either TAPIR or IMGDroid).

bugs (14/16, 88%) are not found by static analyses mainly because:

1) Static checkers fall short on modeling the app’s dynamics over time. Repeated and redundant decoding involves the data flow of images across the event boundary, whose analysis is beyond the capability of existing static checkers.

2) Static checkers fall short on analyzing non-trivial computations. Anti-pattern based analysis cannot precisely model non-trivial computations. Take the “decode without resizing” anti-pattern as an example. Static analyses report a warning only if there is a program path that decodes the
image as is, which is definitely an anti-pattern. However, resizing can be erroneous (like the motivating example in Figure 2), but modeling the decoded image size is beyond the capability of static analyses.

We manually inspected and categorized the static analyses’ IID warnings within the app’s source code (excluding external libraries, which is the setting for TAPIR). There are 60 bug reports in total (TAPIR reported 36 and IMGDroid reported 46, among which 22 reports are overlapped).

1) 7 (12%) issues were known open issues pending resolve (reported in our previous work [8]) by the time this paper was written (for over two years). Considering that all the experimental subjects (apps) are still under active maintenance, we speculate that developers hold a controversial attitude toward the static analysis results: they cannot reject the potential performance impacts yet have insufficient evidence of a performance bug.

2) 53 (88%) issues are new warnings that do not appear in the original study of TAPIR or IMGDroid. Considering that these experimental subjects were actively maintained and evolved for over three years since the publication of TAPIR and an image decoding site may contain multiple anti-patterns (e.g., decoding in the UI thread without resize), this result would be reasonable. In these 53 issues, we confirmed 7 as false positives, 30 as anti-pattern matches, and 16 as undecided cases.

Even with confirmed anti-patterns in source code, some cases will not cause real performance impact (e.g., decodeResource() issues reported by IMGDroid), and the root-cause localization is not a simple procedure [8]. Therefore, we are not expecting the practical deployment of static IID checkers for the Android developers (e.g., as an integrated tool in the IDE), accounting that the developers even explicitly rejected reported issues (e.g., Passandroid-#136).

Further considering the 36 warnings in the library code (as reported by IMGDroid, while TAPIR does not report warnings in these libraries to reduce false warnings), most of the results are difficult to validate, even if we are familiar with these libraries’ internal implementation. Among these warnings (either decoding in the UI thread or decoding without resize), we confirmed 4 (11%) as false positives (not matching the anti-patterns in the paper), 12 (33%) as anti-pattern matches, and 20 (56%) as undecided cases. Even for the 12 anti-pattern matches (true positives), static checkers provide little clue (only an image-decoding site) on the image decoding path and performance impact. We believe that these bug reports benefit no developer in finding performance bugs.

Time Cost. The time consumption study results are shown in Figure 4. The exploration time is 6.2 minutes, and exploitation time is 4.0 minutes averaged over all back-edges. It may require a few hours to complete an app’s performance testing. This is considerably more costly than a simple static analysis, which easily scales to millions of lines of code.

C. Discussions

Effectiveness and Efficiency. With functional test cases (usually available to the developers), our test augmentation approach is fully automatic. Each reported bug is with concrete evidence of performance degradation, which can help developers understand the root cause of inefficient image displaying.

On the contrary, static checkers suffer from false positives and lacking performance degradation evidence. All these factors hinder the adoption of static checkers in practice. ImMut may further complement static checkers by asking a developer to create a functional test case that reaches the static warning’s image decoding site. ImMut can then augment such a test case to reveal whether it can result in real performance problems.

Therefore, though considerably more costly than a lightweight static analysis (but still runs within a reasonable amount of time), ImMut can be static analyses’ useful complement for finding image-displaying related performance bugs.

Threats to Validity. We selected 16 real-world open-source Android apps as evaluation subjects, and the representativeness of these apps may affect the reliability of our evaluation results. To mitigate this threat, all selected Android apps are large in size (The median Loc is around 71 KLoC), well-maintained (containing thousands of commits on average), popular (all have 10K+ downloads), and diverse in categories.

The functional test cases leveraged by ImMut are by a human tester, which may be limited in covering the app’s functionalities. Even with such simple test cases, ImMut discovered a substantial amount of IID issues with real performance impact. Therefore, the claim that ImMut is effective in discovering IID issues should be considered valid.
VI. RELATED WORK

Performance Bugs: Testing and Analysis. This paper belongs to the line of work in performance bug testing and analysis. This paper is based on the fundamental observation that the major cause of performance problems is redundant (or wasted) computation. Such an observation facilitated existing profiling or performance testing [34–37] and static analysis [38–40] techniques, which are all based on identifying redundant or inefficient repetitive computations.

This paper also blends ideas from performance testing in other domains. BLeak [41] and EventBreak [42] manifest performance bugs by repetitive execution of “loops” that connect two identical GUI states. However, they do not involve mutation operators as ImMut and contain no explore-exploit phase to effectively generate long performance test cases.

Performance Bugs in Mobile Apps. Profiling is an effective dynamic approach to performance bug detection [43–45] and diagnosis [46]. There are two pieces of work strongly related to the key idea of ImMut:

Yang et al. [7] proposed to test an app’s responsiveness of GUI by generating tests with “amplified” resources (e.g., shared preference, bitmap, or SQLite database) that trigger expensive operations in the UI thread. This resembles our test oracle of “decoding in the UI thread”. On the other hand, ImMut adopts a more thorough image-displaying related test oracle.

LeakDroid [13] proposed a test augmentation approach to executing repeat cycles of events to manifest resource leaks. However, LeakDroid only repeats short neutral cycles (e.g., repeated rotation, onPause/onResume, onStart/Stop, etc.), while ImMut contains a novel explore-exploit algorithm to generate long performance test cases.

Inefficient Image Displaying in Mobile Apps. To identify IID bugs, PerfChecker [6] is a static anti-pattern checker, in which the “lengthy operations in main threads” is first-time proposed. TARPR [8] and IMGDroid [9] followed anti-patterns extracted from empirical study results on real-world IID issues. Due to the limitation of static analysis, these checkers have to be conservative (and thus miss many performance bugs) to keep a reasonable false-positive rate.

To the best of our knowledge, ImMut is the first performance testing technique for Android apps. Existing dynamic analyzers [7, 23, 24] mainly focus on the test oracle design (e.g., repeated decoding [23], decoding in the UI thread [7], or image leaks [13]) and are not capable of generating long performance tests.

The idea of resource amplification proposed by [7] is also related to this work. However, it only injects a large image (decoded by decodeFile()) with an amplified one for the single purpose of manifesting resource leaks, while ImMut systematically replaces an image (decoded by the 9 APIs) with different prepared larges images, so as to manifest and detect not only image leak problems but other IID issues (e.g., pixel under-utilization) as well.

Mutation Testing in Mobile Applications. ImMut adopts event mutation in generating long performance test cases displaying many images. However, our image- and performance-specific mutants have a different design purpose compared with other mutation testing work for Android: mutation operators in MDroid+ [12], µDroid [47], and µSE [48] are not designated for generating long test cases.

VIII. Conclusion

This paper presents the first dynamic approach to performance testing for image displaying in Android apps. With test augmentation and a pixel-flow based performance test oracle, ImMut can reveal subtle performance bugs beyond the capability of static anti-pattern checkers, demonstrating that testing can be an effective complement for static analysis in detecting image-displaying related performance bugs.

Our prototype of ImMut and experimental data are available via the following URL:

https://struggggle.github.io/ImMut/

ACKNOWLEDGMENTS

This work was supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (Grant No. BK202020001) and Natural Science Foundation of China (Grant Nos. 61932021 and 61802165). The authors would like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Yanyan Jiang and Jun Ma are the corresponding authors.

REFERENCES


