Profiling the Responsiveness of Android Applications via Automated Resource Amplification

Yan Wang
Ohio State University

Atanas Rountev
Ohio State University

ABSTRACT

The responsiveness of the GUI in an Android application is an important component of the user experience. Android guidelines recommend that potentially-expensive operations should not be performed in the GUI thread, but rather in separate threads. The responsiveness of existing code can be improved by introducing such asynchronous processing, either manually or automatically.

One simple view is that all potentially-expensive operations should be removed from the GUI thread. We demonstrate that this view is too simplistic, because run-time cost under reasonable conditions may often be below the threshold for poor responsiveness. We propose a profiling approach to characterize response times as a function of the size of a potentially-expensive resource (e.g., shared preferences store, bitmap, or SQLite database). By manipulating and "amplifying" such resources automatically, we can obtain a responsiveness profile for each GUI-related callback. The profiling is based on a static analysis to generate tests that trigger expensive operations, followed by a dynamic analysis of amplified test execution. Based on our evaluation, we conclude that many operations can be safely left in the GUI thread. These results highlight the importance of choosing carefully—based on profiling information—the operations that should be removed from the GUI thread, in order to avoid unnecessary code complexity.

1. INTRODUCTION

The explosive growth in the use of mobile devices such as smartphones and tablets has led to substantial changes in the computing industry. Android is one of the major platform for such devices [7]. Since Android applications are GUI-based, the *responsiveness* of the GUI is an important component of the user experience. Poor responsiveness can be perceived by the user if an application takes more than 200 ms to respond to a GUI event [9]. If the application is perceived to be sluggish—or, in the worst case, if it leads to an "Application Not Responding" error reported by the Android run-time—the user may decide to uninstall the application and/or rate it negatively in the app market.

Android guidelines [9, 6] are very clear on the importance of designing responsive applications. The general rule is the following: "In any situation in which your app performs a potentially lengthy operation, you should not perform the work on the UI thread, but instead create a worker thread and do most of the work there." [9].

There are various mechanisms for achieving this goal. Typical examples include user-managed threads, `AsynchTask`, and `IntentService`. The responsiveness of existing code can be improved by introducing these mechanisms either through manual refactoring or by using automated transformations (e.g., [12, 11]). A natural question that arises in this context is the following: *which operations should be removed from the GUI thread and placed in a separate thread?* There are several categories of expensive API calls discussed in prior work: typical examples include network access operations, access to local information stored in shared preferences files and SQLite database files, disk I/O operations, image processing operations, and inter-process communication. In developer jargon, such operations are referred to as "jank" [6]. As we demonstrate, not all janky operations have to be moved out of the GUI thread because in many cases their run-time cost under reasonable conditions is well below the threshold for user-observable poor responsiveness.

Our Proposal. Our goal is to characterize the run-time cost of potentially-expensive operations, in order to distinguish the ones that are harmless to the responsiveness from the ones that indeed have to be removed from the GUI thread. The motivation for this characterization is twofold. First, the use of asynchronous constructs complicates the application because it requires additional code to manage the asynchronous work and to communicate with the GUI thread. What used to be a single API call in the GUI thread can become several callback methods in multiple classes, with complex multi-threaded interaction patterns. Furthermore, the increased complexity of the code can lead to defects such as data races, memory leaks, and energy drain [12, 11]. While automated code refactoring could alleviate some of these problems, ultimately the simplest and most desirable solution is to keep in the GUI thread as many of these operations as possible.

We propose a profiling approach to characterize the range of response times as a function of operating conditions. Our focus are potentially-expensive operations whose cost depends on a particular local resource. By manipulating and "amplifying" this resource automatically, we can obtain a

MobileSoft'16, May 16-17, 2016, Austin, TX, USA
 ISBN 978-1-4503-4178-3/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2897073.2897097

responsiveness profile for each GUI-related callback in the application. Specifically, our approach considers operations that involve shared preferences stores, bitmaps, and SQLite databases. In all three categories there is a well-defined application-managed resource whose size is one of the main factors affecting the cost of janky operations. We amplify this size automatically and characterize the range of corresponding response times. As a result, it becomes possible to determine whether the application's responsiveness is acceptable under most/all operating conditions.

The profiling is based on a combination of static and dynamic analysis. The static analysis component automatically (1) creates a model of GUI structure, (2) identifies GUI-related callbacks that invoke janky operations, and (3) generates test cases to execute these callbacks. Subsequently, the dynamic analysis component automatically (1) executes the test cases to collect information about resources used at run time, (2) generates several amplified versions of each resource, and (3) executes the test cases again to measure the running time of each relevant callback under different resource configurations.

Our experimental evaluation and case studies lead to the following conclusion: for the applications we analyzed, janky operations related to SQLite databases and shared preferences can be safely left in the GUI thread. Even for bitmaps, whose processing cost is higher, the response times under realistic operating conditions may sometimes be acceptable. These results highlight the importance of choosing carefully—based on profiling information—the operations that should be removed from the GUI thread, in order to avoid unnecessary code complexity. Our approach provides an automated solution to obtain such profiling information for some commonly-used operations.

2. OVERVIEW

This section presents an overview of relevant Android features together with a high-level description of our approach for responsiveness profiling through resource amplification.

2.1 Relevant Features of Android GUIs

We first describe the event-driven control flow in the GUI of the application (i.e., in the main application thread), followed by a summary of categories of expensive ("janky") operations that may occur along this control flow. Figure 1 contains a code example to illustrate these features. Method `setAsHome` (lines 20–24) contains API calls to manipulate shared preferences. Such calls are one of the standard examples of janky operations [9, 12].

Our goal in defining these aspects of Android behavior is to set the foundation for (1) *automated generation of GUI tests* that trigger janky behaviors, as well as (2) *automated insertion of instrumentation* to obtain a responsiveness profile for run-time behavior.

2.1.1 Windows and Views

Activities are core Android components, defined by subclasses of `android.app.Activity`. An activity displays a window containing several GUI widgets ("views" in Android terminology).

▷ *Example:* Figure 1 shows an example derived from the APV PDF reader [2]; for the sake of simplicity, several non-essential details are elided. Class `ChooseFileActivity` defines an activity. In this case this is the start activity of the application: it is started by the Android launcher when the user launches the application. The `onCreate` lifecycle callback (discussed shortly) initializes a widget representing a list of files and directories (line 7). Field `filesListView` refers to this widget object. At line 8, the activity adds itself as a listener for click events on list items. ◁

```
1  public class ChooseFileActivity extends Activity
2        implements OnItemClickListener {
3    private ListView filesListView;
4    private MenuItem setAsHomeContextMenuItem;
5    private MenuItem deleteContextMenuItem;

     // --- lifecycle callbacks ---
6    public void onCreate() { ...
7      filesListView = (ListView) findViewById(R.id.files);
8      filesListView.setOnItemClickListener(this);
9      registerForContextMenu(filesListView); }
10   public void onResume() {...}
11   public void onCreateContextMenu(ContextMenu menu) {
12     if (...)
13       setAsHomeContextMenuItem = menu.add(R.string.sethome);
14     if (...)
15       deleteContextMenuItem = menu.add(R.string.delete); }

     // --- widget event handler callbacks ---
16   public void onItemClick(View v) { ... }
17   public void onContextItemSelected(MenuItem item) {
18     if (item == setAsHomeContextMenuItem) { setAsHome(); }
19     if (item == deleteContextMenuItem) {...} }
20   public void setAsHome() {
21     SharedPreferences.Editor edit =
22       getSharedPreferences(PREF_TAG, 0).edit();
23     edit.putString(PREF_HOME, currentPath);
24     edit.commit();}}
```

Figure 1: Example derived from the APV application.

We also consider windows for menus and dialogs, which are both used for short interactions with the user. Options menus are associated with activities and context menus are associated with widgets. Dialogs instantiate `android.app.Dialog` or its subclasses.

▷ *Example:* In Figure 1, the call at line 9 allows a context menu to be associated with an element of `filesListView`. When the user performs a long-click event on a list item, an instance of `ContextMenu` is created and callback `onCreateContextMenu` is invoked on it by the Android framework. Inside this callback, two menu items are initialized and added to the menu. Item `setAsHomeContextMenuItem` is used to set a directory as the home directory, while `deleteContextMenuItem` can be used to delete the selected file. These menu items will also referred to as "views" (although technically they are not instances of class `View`). After the initialization, the context menu is displayed as a floating window next to the list element that was long-clicked, allowing contextual information for this element to be displayed. ◁

Let **Win** be the set of run-time windows that correspond to activities, menus, and dialogs. The set of run-time widgets in $w \in \mathbf{Win}$ will be denoted by **View**. This and other related notation is summarized in Figure 2.

2.1.2 Events

A window can be associated with events for the widgets that appear in this window. Such widget events will be denoted by $e = [v,k]$ where v is a widget and k is the kind of event. For the example in Figure 1 we have events $[li,click]$ and $[li,longclick]$ where li is the list item widget on which the event was triggered by the user. We also have events $[mi,click]$ to represent the clicking on a menu item mi from the context menu.

$w \in$	**Win**	window
$v \in$	**View**	view
$e = [v,k] \in$	**Event**	widget event on v
$e = [w,k] \in$	**Event**	default event on w
$c \in$	**Cb**	callback method
$[c,o] \in$	**Cb** $\times$ (**Win** $\cup$ **View**)	callback invocation
$s \in$	**Cbs**	callback inv. sequence
$t = [w,w'] \in$	**Trans**	window transition
$\epsilon(t) \in$	**Event**	event that triggered t
$\sigma(t) \in$	**Cbs**	callback inv. sequence for t
$T \in$	**Trans**$^+$	transition sequence

Figure 2: Notation for run-time semantics.

Default events correspond to hardware buttons. For example, *back* represents pressing the BACK button, which typically returns to some previous window. For example, if *back* is triggered on the context menu in Figure 1 before any menu item is selected, the menu is closed and control returns back to the activity. Event *menu* occurs when the MENU button is pressed to show an options menu (a menu associated with an activity). Other defaults events correspond to the HOME button (*home*), the POWER button (*power*), or rotating the screen (*rotate*) [30]. We will use the notation $e = [w,k] \in$ **Win** $\times \{back, menu, \ldots\}$ for default events; here w is the current window. In Figure 1, let a denote the activity and m denote the context menu. Then we have five default events $[a,\ldots]$, as well as default events $[m,\ldots]$ for *back*, *rotate*, *home*, and *power*.

Let **Event** be the set of widget events $e = [v,k]$ and default events $e = [w,k]$ for an Android application. This set of events—more precisely, a static abstraction of this set—is the basis for the proposed test/instrumentation generation.

2.1.3 Callbacks

The processing of an event e can result in a sequence of callback invocations $[c_1,o_1][c_2,o_2]\ldots[c_m,o_m]$. Here c_i denotes an application-defined callback method. This method is invoked on a run-time object o_i by the Android framework code, in response to the GUI event e. Each callback invocation $[c_i,o_i]$ completes before the next one starts.

An widget event $e = [v,k]$ is processed by a widget event handler callback; examples are `onItemClick` and `onContextItemSelected` in Figure 1. The lifetime of windows is managed with the help of lifecycle callbacks. For example, in Figure 1, creation callback `onCreate` indicates the start of the activity's lifetime, and callback `onResume` indicates a stage of the lifecycle after the activity is reactivated. Similarly, `onCreateContextMenu` in the figure corresponds to the start of the lifetime of the context menu.

▷ *Example:* In Figure 1, consider widget event $[li,click]$ on some list item widget li. This event will trigger a callback invocation [`onItemClick`,li] and formal parameter `v` of the callback method will refer to li. Similarly, consider $[li,longclick]$. The Android framework defines an internal event handler for this event; this handler creates a `ContextMenu` instance m and triggers [`onCreateContextMenu`,m]. Finally, let s be the context menu item referred to by `setAsHomeContextMenuItem` and d be the context menu item referred to by `deleteContextMenuItem`. Event $[s,click]$ triggers [`onContextItemSelected`,s] and $[d,click]$ triggers a similar callback invocation for d. ◁

In these examples the sequence contains only one invocation. However, in the general case there can be a complex callback sequence in response to a single GUI event. For example, suppose that `onItemClick` invoked an Android API call to start some new activity a' (which, in fact, is what the actual code in APV does). In the general case, the invocation sequence would be [`onItemClick`,li][`onPause`,a] [`onCreate`,a'][`onStart`,a'][`onResume`,a'][`onStop`,a]; here a denotes `ChooseFileActivity`. Events *back*, *home*, etc. also trigger callbacks. For example, *home* for an activity may result in a sequence of lifecycle callbacks `onPause`, `onStop`, `onRestart`, `onStart`, `onResume` being invoked on that activity. Additional details of the general structure of callback sequences triggered by GUI events are available in our earlier work [30, 27].

Each callback invocation that appears in such a sequence could potentially invoke janky operations that affects responsiveness. Key steps of our profiling approach include (1) a static analysis to identify callback invocations that may trigger janky operations in response to GUI events, (2) test generation to create sequences of GUI events that execute all such callback invocations, and (3) insertion of instrumentation to record the execution times of these invocations.

2.1.4 Window Transitions

A *run-time window transition* is a pair $t = [w,w'] \in$ **Win** $\times$ **Win** such that when w was active, a GUI event caused the new active window to become w'. Note that w' could be the same as w. A transition t is associated with the event $\epsilon(t)$ that caused it and with the sequence $\sigma(t)$ of triggered callback invocations $[c_i,o_i]$, as discussed earlier. A window transition t may open new windows and/or close existing ones. These effects can be captured by a *window stack* model, which represents the stack of windows that are currently active. A transition t performs a sequence of push/pop operations on the window stack.

Consider $T = \langle t_1, t_2, \ldots, t_n \rangle$, a sequence of window transitions t_i such that the target of t_i matches the source of t_{i+1}. Sequence T is *valid* if the window push/pop operations along the transitions are properly matched, as determined by the state of the window stack [30, 27]. In general, these effects could involve several windows and can trigger complicated callback sequences. In the context of our profiling approach, we perform static analysis of valid transition sequences in order to generate test cases that trigger expensive operations (e.g., as illustrated in method `setAsHome` in Figure 1). Each test case is a valid transition sequence t starting from the initial activity of the application, such that $\sigma(t)$ contains at least one callback invocation $[c_i,o_i]$ for which the execution of callback method c_i for context o_i may trigger (in c_i or in the transitive callees of c_i) a janky operation.

▷ *Example:* For the code in Figure 1, a few of the possible transitions t, together with their triggering GUI events $\epsilon(t)$, are as follows (also shown in Figure 3):

$t_1 = [launcher,a]$	$\epsilon(t_1) = [a,launch]$
$t_2 = [a,a']$	$\epsilon(t_2) = [li,click]$
$t_3 = [a,m]$	$\epsilon(t_3) = [li,longclick]$
$t_4 = [m,a]$	$\epsilon(t_4) = [s,click]$
$t_5 = [m,a]$	$\epsilon(t_5) = [d,click]$

where a represents `ChooseFileActivity`, a' represents another activity opened by a, li is an item in the list displayed by a, m is the context menu, s is the set-home-

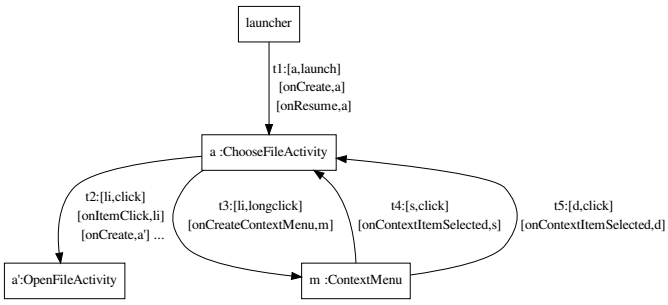


Figure 3: Window transition graph for the running example.

menu-item, and d is the delete-menu-item. Transition t_1 is artificial and represents the starting of the application from the Android application launcher. Transition sequence $T = \langle t_1, t_3, t_4 \rangle$ is valid because t_1 pushes a on top of the window stack, t_3 pushes m on top of a, and t_4 pops m and leads back to a. The callback invocations for T are [`onCreate`,a], [`onResume`,a], [`onItemClick`,li], [`onCreateContextMenu`,m], [`onContextItemSelected`,s] and janky method `setAsHome` is invoked by the last callback. Thus, T could be one of the test cases generated by our approach. Note that a similar valid sequence $T' = \langle t_1, t_3, t_5 \rangle$ will not be selected as a test case by our technique, since no callback invocation along T' invokes janky operations: when `onContextItemSelected` is invoked on d, method `setAsHome` is not called. ◁

Figure 3 shows the *window transition graph* (WTG) for the running example. The WTG is a static representation of the possible run-time window transitions, introduced in our prior work [30]. The figure shows the event that triggers each transition, together with the callbacks invoked as part of the transition. For edge t_2, additional lifecycle callbacks (denoted by ...) are omitted for brevity. The WTG representation can be used to statically identify (1) callback invocations that invoke janky operations, and (2) valid window transition sequences that trigger such callbacks. In the example, transition sequence $T = \langle t_1, t_3, t_4 \rangle$ will be detected and used to generate a test case, as described later.

2.2 Expensive Operations in the UI Thread

The callbacks and window transitions described above are executed in the UI thread of the application. This is the main application thread and it is responsible for dispatching and handling of GUI events (clicks, etc.) as well as events related to other device features (e.g., location awareness, battery state changes, etc.). We aim to identify window transitions $t = [w,w']$ that may trigger expensive operations. The main categories of janky operations discussed in prior work [9, 6, 28, 12] are as follows: (1) *network access* operations, such as connecting to a network, storing and retrieving data from a web-based server, etc.; (2) *shared preferences* operations, which store and retrieve data from a key-value store; (3) *bitmap* operations such as loading and decoding an image from a file; (4) *SQLite database* operations to store and retrieve data from a database; (5) *storage access* operations, which read or write a file in internal or external flash storage; (6) *RPC calls* between two processes; (7) *content provider* operations, which access a structured set of data managed by a provider component, and are used to share data among processes.

Network operations are the canonical example of janky operations and Android developer guidelines suggest that they should be moved out of the UI thread because their cost depends on factors outside of the application's control. The guidelines for the remaining categories of operations are less clear. Although database operations and bitmap operations are given as other typical examples of jank [9], in many cases their cost is actually predictable and harmless. To gain understanding of performance implications, standard profiling can be applied to such operations: any callback that invokes (directly or transitively) a janky operation can be instrumented before execution, and the running time from callback-entry to callback-exit can be measured and compared against guidelines for good responsiveness: "generally, 100 to 200 ms is the threshold beyond which users will perceive slowness in an application" [9].

Our profiling approach aims to go beyond standard profiling, and to characterize the range of response times as a function of different operating conditions. In particular, we are interested in janky operations whose cost depends on a particular resource we can manipulate and "amplify" automatically. Thus, we focus on operations that involve shared preferences, bitmaps, and SQLite databases. In all three categories there is a well-defined resource, managed by the application, such that the size of the resource affects the cost of janky operations. As we discuss later, this size is one of the main factors that determine the cost of janky API calls. By amplifying the size automatically we can characterize the range of acceptable response times and can determine whether the application remains in this range under most (or even all) realistic operating conditions.

▷ *Example:* For the code in Figure 1 we can determine that the execution time of `onContextItemSelected`, when executed on the set-home menu item, will remain under 200 ms even when the size of the shared preferences (i.e., the number of pairs in the key-value store) is amplified to around 60,000 pairs. The application uses the shared preferences to save the path of the home folder. This path is a string value, associated with the string key `"Home"`. Method `setAsHome` stores this pair into a shared preferences XML file located in the application's shared preferences folder. There does not exist an execution scenario in which this shared preferences store contains any other key-value pair. Since the threshold for bad responsiveness is around 60,000 pairs, clearly it is unnecessary to remove `setAsHome` from the UI thread. ◁

Note the three categories of operations we consider access files that are stored on the device's flash storage: XML files for shared preferences, image files for bitmaps, and database files for SQLite operations. For these categories of files automated amplification is possible by increasing the file size. A similar amplification may also be possible for general files stored in flash storage. However, being able to increase the size of a file in a meaningful way, and to reorganize the data in it to trigger different run-time access patterns, requires knowledge of the meaning of the application-specific data being stored in that file. In future work we plan to investigate how such knowledge could be provided (with little effort) by the programmer.

2.2.1 Shared Preferences

Shared preferences is a general mechanism in Android to facilitate saving and retrieving a collection of key-value pairs. It can be used to store any primitive data includ-

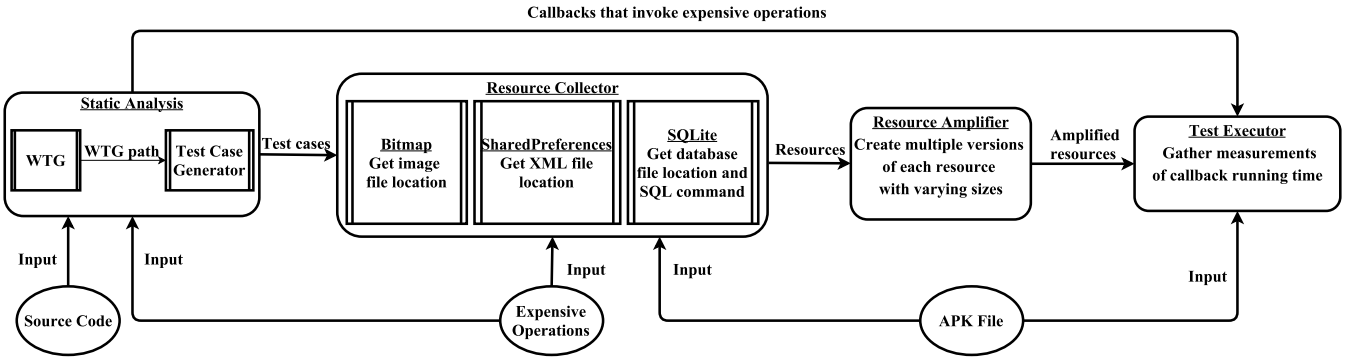


Figure 4: Workflow for responsiveness profiling.

ing booleans, ints, floats, longs, and strings. The data will persist across user sessions in an XML file under the application's private data directory. This is one of the most widely used techniques to store a small amount of persistent data in Android applications. Before using the store, a `SharedPreferences` object has to be obtained either by providing a file name or by using the default one for the application. When this happens, our approach (through instrumentation) can record which file is being used and later can amplify the file, as described shortly.

To read values, methods such as `getString`, `getBoolean`, etc. in class `SharedPreferences` are invoked, with a string key provided as an actual parameter. To perform a write operation, a `SharedPreferences.Editor` has to be used (see lines 21–24 in Figure 1). The values are added using corresponding put methods such as `putString`, `putBoolean`, etc. The new values are written with a call to `commit`. If the size of the shared preferences store is large, the cost of some of these operations can affect responsiveness. The experiments described in Section 4 provide further details on the relationship between the size of the store and the response time for GUI events.

2.2.2 Bitmap Processing

A *bitmap* is one of the drawable resources in Android. Normally, it is a graphic file that can be drawn on the screen. There are several possible formats for a bitmap (e.g., PNG). In order to load and decode an image from a file, Android provides a helper class `BitmapFactory`. Inside this class, methods `decodeResource` and `decodeFile` will load images from the resource folder of the application and the file system, respectively. Such decoding operations, as well as other related operations such as re-sizing, can be expensive for large images. Our technique determines which graphics files are accessed by the application and amplifies them by automatically creating larger versions (e.g., by increasing the number of pixels in both dimensions). The effects on GUI event handling response time are then characterized as a function of image size.

2.2.3 SQLite Database

Android provides SQLite databases as a convenient mechanism to save structured data. Two classes are used to manipulate such databases. Helper class `SQLiteOpenHelper` manages database creation and versioning. Class `SQLiteDatabase` contains APIs to perform database reads and writes. Generally, these APIs can be divided into two categories. One category is intended to work on a single table. For example, `SQLiteDatabase` provides a method `query(...)` to query a given table and return a `Cursor` to the result set. As another example, methods `insert`, `delete`, and `update` perform inserting, deleting, and updating for a certain table. The second category contains general operations that could work on several tables; for example, an SQL command could be executed using `execSQL(String sql)`.

We are interested in the common scenario where the application creates and manages its own database. During initial testing, we execute a freshly-installed application copy which creates the database file. By instrumenting the relevant API calls we can determine the file being created and the structure of each database table. The tables are then amplified, by adding additional rows, before the profiling runs. The cost of a database operation depends on several factors: for example, table sizes, query type and complexity, and size of the result. However, there is evidence that the typical pattern of database usage in Android applications is rather simple [10]. As a result, one would expect that database size would be the main factor affecting cost. This observation is supported by the results presented in Section 4.

3. PROFILING APPROACH

As outlined earlier, our profiling approach is a combination of static and dynamic analysis. Figure 4 illustrates the overall process. The static analysis component identifies callbacks that invoke janky operations and generates test cases to trigger these callbacks. The dynamic analysis uses the output of the static analysis to (1) collect the list of resources used at run time, (2) generate several amplified versions of each resource, and (3) measure the running time of each relevant callback.

3.1 Input

The source code of the application is the input of the static analysis and the corresponding APK file is the input of the dynamic analysis. Note that the static analysis is capable of processing APKs instead of source code [19], but for our case studies we wanted to examine manually the internals of the application to understand better the patterns of usage of janky operations. An additional input to both analyses is a specification of the janky API calls for shared preferences, bitmaps, and SQLite databases [9, 6, 28, 12]. Both the static analysis and the instrumentation component of the

Algorithm 1: GeneratePaths

```
Input: wtg = (N, E) : window transition graph
Input: Relevant ⊆ N : set of relevant nodes
Input: Cover : set of janky [c_i,o_i] to be covered
1  t ← [launcher,main] ∈ E
2  path ← ⟨ t ⟩
3  stack ← ⟨ main ⟩
4  UpdateTests(Cover, t, path)
5  Traverse(main, path, stack)
6  procedure Traverse(w, path, stack)
7      foreach edge t = [w,w'] such that t ∉ path do
8          if w' ∈ Relevant ∧ CanAppend(t, path, stack)
           then
9              DoAppend(t, path, stack)
10             UpdateTests(Cover, t, path)
11             Traverse(w', path, stack)
12             UndoAppend(t, path, stack)
13 procedure UpdateTests(Cover, t, path)
14     Expensive ← set of janky [c_i,o_i] executed by t
15     if Expensive ∩ Cover ≠ ∅ then
16         record path for subsequent test generation
17         Cover ← Cover − Expensive
18         if Cover = ∅ then
19             terminate traversal
```

dynamic analysis utilize the Soot framework [23] and the Gator analysis toolkit for Android [19].

3.2 Static Analysis

The first step of the static analysis builds the WTG outlined at the end of Section 2.1.4. Next, each callback invocation in the WTG is analyzed to determine whether it invokes expensive operations. For example, the WTG in Figure 3 shows 7 distinct callback invocations $[c_i,o_i]$, including both widget event handlers such as `onContextItemSelected` and lifecycle callbacks such as `onCreate`. Each unique $[c_i,o_i]$ is analyzed separately. An inter-procedural control-flow graph (ICFG) [21] is constructed for c_i and its transitive callees in the application code. Then, a constant propagation analysis is performed to identify and remove infeasible ICFG edges, based on the knowledge that the calling context of c_i is o_i. This analysis is defined in prior work [29], where it was shown to produce more precise control-flow models. For the example in Figure 1, this analysis will determine that when `onContextItemSelected` is invoked on menu item `setAsHomeContextMenuItem`, the body of the first if-statement (line 13) is executed but the body of the second one (line 15) is not. Similarly, the analysis determines that when this callback method is invoked on `deleteContextMenuItem`, janky method `setAsHome` is not executed.

Next, a backward traversal of the WTG is performed from the source nodes of all edges that contain expensive callbacks. In the WTG from Figure 3, t_4 is such an edge. The goal of the traversal is to identify all WTG nodes w that may belong to paths leading to edges that need to be covered during profiling. We refer to such nodes as *relevant* nodes. In the running example, a backward traversal from m (which is the source node of t_4) identifies m, a, and *launcher* as relevant nodes. The shortest distance from the launcher node to each relevant node can then be computed with a simple breadth-first traversal. The largest such distance is used during path traversal (described next) as a threshold for path length.

Next, a forward traversal of the WTG is used to identify a set of valid WTG paths (validity was defined in Section 2.1.4) that start from the launcher node, contain only relevant nodes, and cover each janky $[c_i,o_i]$ at least once. During the traversal, the current path is recorded and a new edge e is appended to T only if (1) $e \notin T$, (2) the target node of e is relevant, and (3) the resulting path is valid. If a path T covers a pair $[c_i,o_i]$ that has not been covered already, T is recorded and later used to create a test case. In the running example, $T = \langle t_1, t_3, t_4 \rangle$ is a path that is constructed during the traversal and is recorded for test case generation.

Algorithm 1 provides further details on this approach. The current WTG path *path* and the corresponding window stack *stack* are maintained during the traversal. For brevity, the threshold on path length mentioned above is not shown. Helper function CanAppend determines whether the window push/pop operations for $t = [w,w']$ can be applied to the window stack—that is, whether the sequence of push/pop operations along *path* appended with t is valid. If this is the case, helper function DoAppend appends t and modifies *stack* accordingly. After the recursive traversal is finished, helper function UndoAppend removes t from *path* and reverts the stack changes performed by DoAppend.

During the traversal, set *Cover* contains the janky callback invocations $[c_i,o_i]$ that remain to be covered. This set is updated as new paths are constructed, and the traversal terminates when the set becomes empty. Given a set of paths $T_1, T_2, \ldots$ recorded at line 16, the test case generator creates a test case for each T_i. The test cases are implemented in the Robotium testing framework [18]. For each transition t in a path T_i, event $\epsilon(t)$ is mapped to a corresponding Robotium API call. This process is automated, but in some circumstances may have to be followed by manual setup steps—for example, introducing certain files in the file system, mocking the camera (i.e., to simulate scanning of barcodes), providing login/password information, etc.

The output of Algorithm 1 depends on the order in which successor nodes w' are visited at line 7. To ensure that the set of generated test cases is deterministic, we impose an ordering of successors, and use that same ordering for each run of the algorithm.

3.3 Dynamic Analysis

The dynamic analysis has three components. First, the test cases are executed without amplification, in order to obtain information about the resources they utilize. This is done by a *resource collector* component through appropriate instrumentation at certain API calls. Next, a *resource amplifier* component creates several versions of each resource, with a range of different sizes. Finally, a *test executor* executes again the test cases, using the amplified resources, and gathers measurements about the execution times of callbacks that may invoke janky operations.

3.3.1 Resource Collector

The first component is the *resource collector*. Its purpose is to get the actual resources used by the expensive operations at run time. For shared preferences, the resource is the name of the XML file. There are several methods to obtain a `SharedPreferences` object; an example is `getSharedPref-`

App	Shared Preferences		Bitmap	SQLite Database		Test Cases
	read	write		read	write	
APV	7	5	0	2	2	6
BarcodeScanner	4	0	1	6	3	9
OpenManager	2	1	0	0	0	1
SuperGenPass	2	4	0	0	0	3
TippyTipper	17	0	0	0	0	10
VuDroid	3	2	1	0	0	2

Table 1: Number of callbacks and test cases

erences. The shared preference XML file can be obtained by instrumenting such call sites and recording the file name at run time. For bitmaps, the Android libraries provide a helper class BitmapFactory, which is used to decode and load images from an input source. We analyze two kinds of sources. One is the name of an image file, which is used as a parameter of method decodeFile. The other source uses the resource folder of the application. Here method decodeResource takes as input an integer parameter id to identify the desired resource. In Android each resource id is represented by a constant field in an automatically-generated class R. A field name gives the corresponding file name. For example, if field R.drawable.arrowup is initialized with 0x7f020000 in the application code, and parameter id of decodeResource has that same value at run time, the image file name is arrowup in the resource folder in the application.

For SQLite databases, Android provides an abstract class SQLiteOpenHelper to manage database creation; applications override certain methods defined in this class. The name of the database file is provided as input to the class constructor. A related callback onCreate is defined by subclasses in the application code and is invoked when the database is created for the first time. This method is intended to be used to create database tables using an API call execSQL(String sql) where sql is a string with an SQL statement. An example of such a string is CREATE TABLE bookmark VALUES(id integer primary key, name text). Here bookmark is a table with two columns id and name. By instrumenting the relevant call sites, we gather the name of the database file and the name/structure of individual tables. An underlying assumption is that the application is responsible for creating its own database; although this does not have to be the case, in all examples we have seen so far this assumption is satisfied.

The instrumentation is performed by decompiling the application's APK file to Soot's intermediate representation, adding appropriate additional statements in this IR, and then reconstructing the APK file. The test cases generated by the static analysis are then executed and the gathered resource information is used for resource amplification.

3.3.2 Resource Amplifier

After gathering all accessed resources, the next step is to amplify them. For shared preferences, a given number of random key-value pairs is inserted in the XML file. For bitmaps, an image file can be enlarged using the standard ImageMagick tools. For each database file, a simple JDBC client is used to insert random rows in the database tables, using the structural information (number/type of columns) extracted by the resource collector. The modified store/image/database files are created off-line and are then used by the test harness during test execution. In our current implementation we perform per-category amplification: in one set of experiments only (and all) shared preferences for an application are amplified, another set of experiments amplifies only the bitmaps, and a third set considers only the databases. Of course, more fine-grained amplification can be easily achieved.

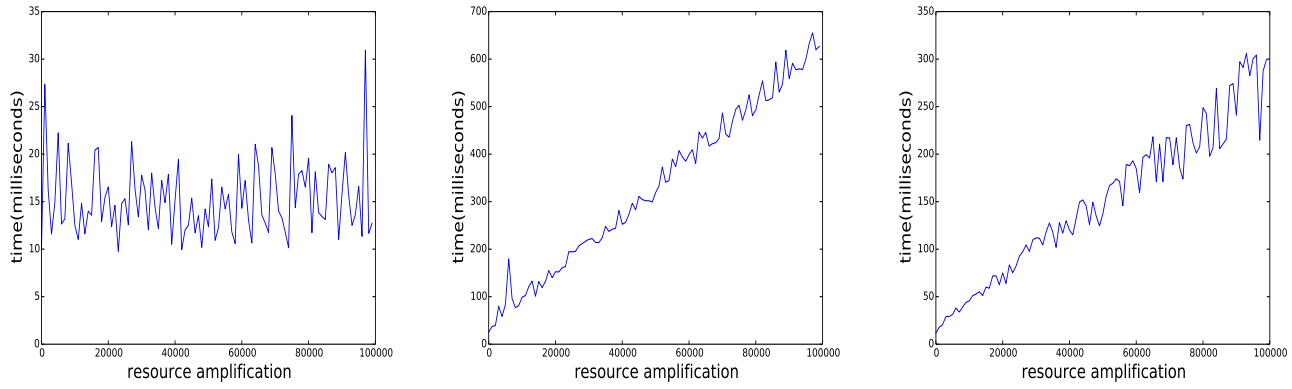
3.3.3 Test Executor

After amplification, the same set of test cases is executed over a range of amplification sizes. Before execution, instrumentation is inserted at the entry and exit of each callback method c_i that was identified by the static analysis as potentially invoking expensive operations. Before each test run, the application is uninstalled and then reinstalled to ensure that the same starting state is used. For a particular run of the test cases, with a particular set of amplified resources, the average execution time of each callback c_i is measured. Each test run typically takes one to two minutes; we perform three runs and record the average of the three measurements. When firing a GUI event through a Robotium call, it is necessary to wait until the effects of the event are processed and shown in the GUI, so that the next event can be fired. We automatically introduce a delay after each event in the test cases, which is the reason a test run can take up to two minutes. Since the overall cost ultimately depends on the number of amplified sizes, the execution time can be controlled by choosing that number. These amplified sizes could be selected by various criteria. For example, as described in the next section, we used 100 amplified versions of the shared preference stores, with the first version containing 1000 additional key-value pairs in each store, the second one containing 2000 additional pairs, and the last one containing 100,000 additional pairs.

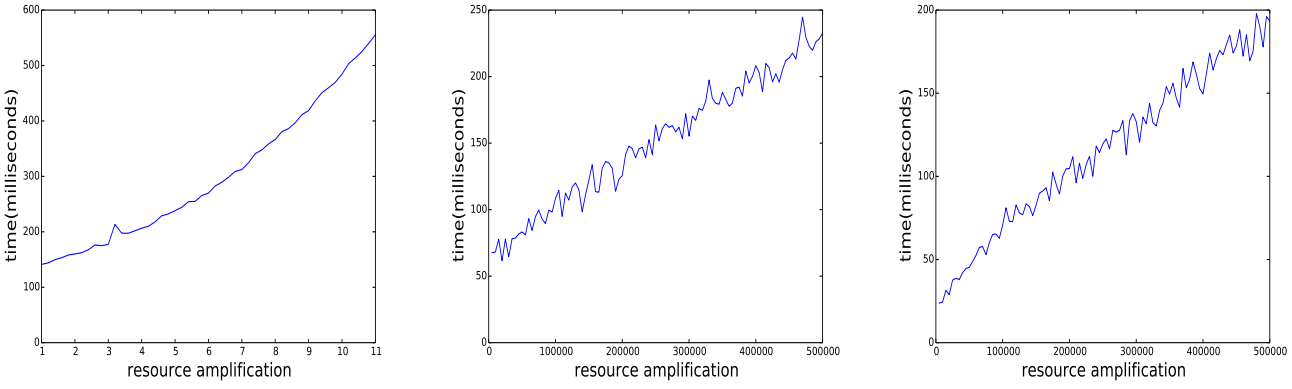
4. EXPERIMENTAL EVALUATION

We evaluated the profiling approach on six open-source applications. These applications were chosen to reduce the manual effort needed to interpret the results: in prior work [29, 30] we have investigated parts of their source code, for purposes unrelated to performance analysis. The applications are from several domains: PDF and e-book reader (APV, VuDroid), file management (OpenManager), password generation (SuperGenPass), barcode processing (BarcodeScanner), and simple calculations (TippyTipper).

Application characteristics are shown in Table 1. For each of the three categories, the table shows the number of callback methods that may invoke (directly or transitively) expensive APIs from this category. For shared preferences and databases, there is separation into read operations (that do not modify the resource) and write operations (that modify



(a) Preferences read (b) Preferences read (first read) (c) Preferences write



(d) Bitmap (e) Database read (f) Database write

Figure 5: Representative sample of profiling results.

the resource). We make this distinction in order to investigate whether the run-time cost is different for these two sub-categories. The last column is the number of test cases generated to trigger all these callbacks. For all programs but one, the time to analyze the WTG and generate all test cases is around one second. `BarcodeScanner` takes 47 seconds because the WTG has a large number of edges, caused by the high degree of polymorphism in the type of barcodes: eleven types of barcodes are possible, as discussed in [30]. As a result, the number of paths is more than two orders of magnitude larger than that for the other applications.

4.1 Experimental Results

We executed the generated test cases, with amplified resources, on a Nexus 5X smartphone with a 1.8GHz 64-bit Qualcomm Snapdragon 808 CPU and 32GB storage. The Android SDK version is 6.0 (Marshmallow). A representative sample of the profiling results is shown in Figure 5. The other results observed in our experiments exhibit the same trends as the ones shown in the figure.

4.1.1 Shared Preferences

Figure 5 shows three charts for the responsiveness of callbacks that access shared preferences in application `APV`. Figure 5a and Figure 5b are for callback `onCreate` in class `ChooseFileActivity`, while Figure 5c is for callback `onContextItemSelected` which was illustrated in the running example (Figure 1). The y-axis represents the average execution time of potentially-expensive callbacks (in milliseconds) and the x-axis shows the number of key-value pairs added to the store before running the test cases. As discussed in Section 3.3.3, 100 amplified runs were executed and the number of added key-value pairs was increased by 1000 in each run. Thus, the largest amplification is 100,000 key-value pairs. We also performed measurements to ensure that the number of callback invocations (i.e., the denominator in the computation of the average callback execution time) does not change from run to run.

Figure 5a and Figure 5b show different execution times, although both are for the same callback method. This is because Figure 5b corresponds to the first run of this method while Figure 5a shows the average of the remaining (not first) invocations of this method during the same run of the test cases. We make this distinction because the behavior is significantly different in the two cases. In Android, the shared preferences will be loaded into memory when the application calls `getSharedPreferences` (or as few other similar APIs) for the first time. The key-value pairs will be stored in a new hash map. The building of this map is done by separate thread. However, if the call to `getSharedPreferences` is followed by a read operation (i.e., `getBoolean`), this read has to wait for the completion of the map. This

is exactly what happens in `onCreate` in class `ChooseFileActivity`: a read operation is invoked immediately after a call to `getSharedPreferences`. When `onCreate` is invoked for the first time, its execution time is proportional to the size of the store. Any subsequent invocation of `onCreate` directly uses the in-memory hash map, and its read operation has very low $O(1)$ running time as illustrated by Figure 5a.

Figure 5c shows the cost of `onContextItemSelected` from the running example. This callback method invokes `commit` (line 24 in method `setHome`), which internally writes the hash map to the XML file. Although this writing is done in a separate thread, `commit` waits for its completion. As a result, the cost of the write operation is proportional to the number of key-value pairs.

4.1.2 Bitmap Processing

Figure 5d shows the response time for callback `onCreate` in class `BaseViewerActivity` from `VuDroid`. The format of the input image is PNG, which is the default one used by the application. The y-axis is again the average callback execution time in milliseconds and the x-axis is the size of the image in each dimension, relative to the original size. For example, 1 on the x-axis corresponds to the original size, while 2 means the image size is twice as large in each dimension. The application was executed several times, with amplification factors of 1.2, 1.4, etc., until the value on the y-axis significantly exceeded 200 ms (i.e., the threshold for user-observable slow GUI response). For this chart, there are three relevant images whose original sizes are 64×54, 216×54, and 300×15. Their final sizes are 704×594, 2376×594 and 3300×165, respectively.[1] Clearly, the execution time of the callback grows significantly as the image size increases.

Android accepts three formats of image files: PNG, JPEG, and GIF. According to the documentation, PNG is preferred, JPEG is acceptable, and GIF is discouraged. We automatically generated each of the formats for each image file used by the application. However, our experiments indicate that the choice of format does not have significant effect on the response time (less than 10% difference). We conclude that it is unnecessary to add the format choice as an additional dimension of amplification.

4.1.3 SQLite Database

Database operations can also be divided into two major categories: read operations and write operations. One example of the performance of reading is shown in Figure 5e for callback `onCreate` from class `OpenFileActivity` in `APV`. A similar example for write operations is presented in Figure 5f for callback `onPause` from the same class. The x-axis shows the number of rows we insert in each of the tables of this database before the test cases are executed. The database is empty when the original application is installed. We increase the size of each table by 5000 rows in each run, for a total of 140 runs.

For this `onCreate` callback method, the database is accessed to read the contents of one single tuple from table `bookmark`. Similarly, `onPause` updates one tuple from the same table. As the measurements show, the cost of these operations is proportional to the number of tuples in the database table. It takes a table size of over 400,000 tuples for the callback execution time to exceed 200 ms.

[1]For comparison, the built-in camera in the smartphone used for the experiments takes 3024×4032 photos.

4.1.4 Flash Storage State

Android guidelines [6] suggest that the performance of I/O operations for the flash storage depends on how full that storage is. Since all our resources are ultimately represented as files on this storage, we performed additional amplification experiments on the state of the storage, starting from about 10% full to 99.9% full. In each experiment, the test cases were executed on the original application, without any resource amplification. We did not see any significant change in callback execution time across different states of the flash storage. One possible explanation is that the applications we analyzed are not write-intensive. Significant volumes of write operations may lead to different performance profiles [6] and this dimension of amplification may still be useful for certain categories of applications.

4.2 Findings

Based on the experimental results, we can reach the following conclusions. First, some of the so-called janky operations are not expensive. For example, reading from shared preferences is not expensive even for large stores and for first-read operations. For first-read operations and for write operations, the cost grows linearly but still remains under 200 ms when the number of key-value pairs is less than 25,000. Android documentation suggests the use of shared preferences for a "relatively small collection of key-values" [20]. In our experiments, no application has more than 100 pairs in its store. Our conclusion is that in realistic scenarios, the uses of shared preference APIs can remain in the GUI thread. Although the cut-off point for store size may be somewhat different across devices and platform versions, it is highly unlikely that this overall conclusion will change for other execution environments.

Clearly, the profiling results may also help the developer to estimate whether janky operations are indeed harmful. This is illustrated by the case study of bitmap operations in `VuDroid` (Figure 5d). At amplification factor of around 4, the response time exceeds the threshold of 200 ms. However, in this particular application, when image sizes are about 3.9 times the original ones in each dimension, the layout becomes corrupted since one image will be completely covered and invisible. One can argue that in our experimental setup, the janky bitmap operations under realistic conditions are still not harmful. In contrast, consider `BarcodeScanner`. Although the profiling trends are similar to the ones for `VuDroid`, the application's layout is resilient to large images because it scales the image to fit the available display. Thus, realistic executions may include large images, and refactoring will be needed to move the API calls out of the GUI thread. In all likelihood, the developer's knowledge of the application will be enough to distinguish between such scenarios: for example, there may be application-specific constraints on the size of images that will ever be loaded in the application. With the help of profiling information, the developer can make informed decisions whether it is necessary to perform transformations for improved responsiveness.

The results for database operations (Figures 5e and 5f) indicate that the relevant API calls often do not have to be moved out of the GUI thread. It is likely that most applications do not need 400,000 rows to be stored in each table of the database. For applications that do require large databases, the profiling can provide information about size thresholds beyond which one may need to refactor the code

[12] or offload/purge the data. It is also important to note that the cost of database operations depends not only on the size of database tables but also on the inherent complexity of the SQL commands. However, as described in recent work [10], the vast majority of Android SQLite database workloads involve simple, small requests for data that touch a small number of tables. Thus, the natural characterization of database operation cost is as a function of database size. We manually analyzed the SQL operations in our benchmarks, and made similar observations: most queries involved reading or updating a single tuple in a table, and the remaining ones involved the selection of a simple subset of tuples from one database table. Our conclusion is that for many applications, database operations could likely remain in the GUI thread.

Using the profiling results gathered by the proposed approach, together with knowledge of expected operating conditions, a developer could make a decision about the potential for poor responsiveness and the need to move some operations out of the GUI thread. As illustrated by our results, in many cases such invasive changes may be unnecessary.

5. RELATED WORK

Performance analysis for mobile software. Yang et al. [28] propose a technique to detect potential "Application Not Responding" defects by inserting artificial long delays at janky operations. Nistor and Ravindranath [16] predict performance problems in smartphone applications from a given small input. They summarize a prioritized list of repetition patterns by logging callbacks at run time and checking for potential problems by adding time delays in suspicious methods which are identified from the patterns. Lin et al. [12] study performance problems in Android applications and build a tool to address poor responsiveness defects by refactoring potentially-expensive operations out from the UI thread. Their follow-up work [11] studies mechanisms for asynchronous execution in Android and proposes an approach to transform between these mechanisms. Thanaporn and Shingo [17] detect janky operations in Android by examining a static control-flow model. Their static analysis is much less general than ours and does not consider sequences of callbacks or automated test generation. Kennedy et al. [10] survey the usage of SQLite databases in Android applications, including the performance of database operations.

In all prior work on responsiveness analysis for Android, the focus is on detection and removal of potentially-expensive operations. The novel contributions of our work are twofold. First, our static analysis not only detects janky operations but also identifies which sequences of GUI events may trigger them; this is essential for subsequent test generation. Second, the presence of janky operations by itself is not sufficient to justify the maintenance effort and increased code complexity that result from moving these operations out of the GUI thread. As illustrated by our experimental results, a more nuanced analysis is needed, guided by run-time profiling and knowledge of the typical operating conditions of the application. Our proposal for profiling based on resource amplification is a first step in this direction.

Test amplification. Amplification can be used to detect potential problems during testing. Zhang and Elbaum [31] apply test amplification techniques for exception-handling code, which is necessary due to unreliable environment factors such as network connectivity. Their approach amplifies existing test cases by injecting exceptions, while in our profiling we do not change test execution code paths but rather the run-time cost of the execution. Fang et al. [5] develop a tool to quickly find memory-related performance problems in managed languages by amplifying the size of allocated objects.

Test generation for Android. A recent study by Choudhary et al. [4] summarizes the state of the art in automated test generation for Android. A few representative examples are discussed below. The standard Monkey tool [15] uses a simple random strategy for generating UI events. GUIRipper [24] generates test cases based on a dynamically built GUI model using depth-first-search exploration. ORBIT [25] uses a similar exploration strategy but combined with static code analysis to determine which UI events are relevant for a specific activity. The A^{3}E GUI exploration tool [3] employs two strategies: purely-dynamic depth-first exploration and targeted exploration based on a control-flow model from a static taint-like analysis. ACTEve [1] is a concolic testing tool which symbolically tracks events from their generation to their handling. Jensen et al. [8] use symbolic analysis to create event handler summaries and to build event sequences using the summaries and a UI model. Other examples of testing tools for Android include Dynodroid [13], SwiftHand [26], EvoDroid [14], and PUMA [22].

The WTG is a more general GUI control-flow representation than any of the static models used in prior work on test generation, because it models the complex interplay between event sequences, callbacks triggered by them, and the corresponding changes to the window stack. Although our WTG-based test generation aims to cover janky operations, the underlying approach could easily be adapted to target testing of other interesting aspects of application behavior.

6. CONCLUSIONS

Our results strongly indicate that a more nuanced approach is needed when handling potentially-expensive operations in the GUI thread. Some operations are harmless and moving them out of the GUI thread creates unnecessary code complexity without many performance benefits. Using the proposed automated test generation and resource amplification, a developer can better understand these trade-offs and can reach informed conclusions about application responsiveness and whether code transformations are necessary to improve it.

Acknowledgement

We thank the MOBILESoft reviewers for their valuable feedback. This material is based upon work supported by the U.S. National Science Foundation under CCF-1319695 and CCF-1526459, and by a Google Faculty Research Award.

7. REFERENCES

[1] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *FSE*, pages 1–11, 2012.

[2] APV PDF viewer. `code.google.com/p/apv`.

[3] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *OOPSLA*, pages 641–660, 2013.

[4] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet? In *ASE*, pages 429–440, 2015.

[5] L. Fang, L. Dou, and G. Xu. Perfblower: Quickly detecting memory-related performance problems via amplification. In *ECOOP*, pages 296–320, 2015.

[6] B. Fitzpatrick. Writing zippy Android apps. In *Google I/O Developers Conference*, 2010.

[7] Gartner, Inc. Worldwide traditional PC, tablet, ultramobile and mobile phone shipments, Mar. 2014. `www.gartner.com/newsroom/id/ 2692318`.

[8] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *ISSTA*, pages 67–77, 2013.

[9] Keeping your app responsive. `developer.android.com/training/articles/perf-anr.html`.

[10] O. Kennedy, J. A. Ajay, G. Challen, and L. Ziarek. Pocket data: The need for TPC-MOBILE. In *TCPTC*, pages 282–292, 2015.

[11] Y. Lin, S. Okur, and D. Dig. Study and refactoring of Android asynchronous programming. In *ASE*, pages 224–235, 2015.

[12] Y. Lin, C. Radoi, and D. Dig. Retrofitting concurrency for Android applications through refactoring. In *FSE*, pages 341–352, 2014.

[13] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *FSE*, pages 224–234, 2013.

[14] R. Mahmood, N. Mirzaei, and S. Malek. EvoDroid: Segmented evolutionary testing of Android apps. In *FSE*, 2014.

[15] Monkey: UI/Application exerciser for Android. `developer.android.com/ tools/help/monkey.html`.

[16] A. Nistor and L. Ravindranath. SunCat: Helping developers understand and predict performance problems in smartphone applications. In *ISSTA*, pages 282–292, 2014.

[17] T. Ongkosit and S. Takada. Responsiveness analysis tool for Android application. In *DeMobile*, pages 1–4, 2014.

[18] Robotium testing framework for Android. `code.google.com/p/robotium`.

[19] A. Rountev, D. Yan, S. Yang, H. Wu, Y. Wang, and H. Zhang. GATOR: Program analysis toolkit for Android. `web.cse.ohio-state.edu/presto/software`.

[20] Saving key-value sets. `developer.android.com/training/basics/data-storage/shared-preferences.html`.

[21] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[22] H. Shuai, L. Bin, N. Suman, G. H. William, and G. Ramesh. PUMA: Programmable UI-automation for large scale dynamic analysis of mobile apps. In *MobiSys*, pages 204–217, 2014.

[23] *Soot Analysis Framework*. `http://www.sable.mcgill.ca/soot`.

[24] P. Tramontana. Android GUI Ripper. `wpage.unina.it/ptramont/GUIRipperWiki.htm`.

[25] Y. Wei, R. P. Mukul, and X. Tao. A grey-box approach for automated GUI-model generation of mobile applications. In *FASE*, pages 250–265, 2013.

[26] C. Wontae, N. George, and S. Koushik. Guided GUI testing of Android apps with minimal restart and approximate learning. In *OOPSLA*, pages 623–640, 2013.

[27] S. Yang. *Static Analyses of GUI Behavior in Android Applications*. PhD thesis, Ohio State University, Sept. 2015.

[28] S. Yang, D. Yan, and A. Rountev. Testing for poor responsiveness in Android applications. In *MOBS*, pages 1–6, 2013.

[29] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *ICSE*, pages 89–99, 2015.

[30] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static window transition graphs for Android. In *ASE*, pages 658–668, 2015.

[31] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *ICSE*, pages 595–605, 2012.