Race Detection for Android Applications

Pallavi Maiya
Indian Institute of Science
pallavi.maiya@csa.iisc.ernet.in

Aditya Kanade
Indian Institute of Science
kanade@csa.iisc.ernet.in

Rupak Majumdar
MPI-SWS
rupak@mpi-sws.org

Abstract

Programming environments for smartphones expose a concurrency model that combines multi-threading and asynchronous event-based dispatch. While this enables the development of efficient and feature-rich applications, unforeseen thread interleavings coupled with non-deterministic reorderings of asynchronous tasks can lead to subtle concurrency errors in the applications.

In this paper, we formalize the concurrency semantics of the Android programming model. We further define the *happens-before* relation for Android applications, and develop a dynamic race detection technique based on this relation. Our relation generalizes the so far independently studied happens-before relations for multi-threaded programs and single-threaded event-driven programs. Additionally, our race detection technique uses a model of the Android runtime environment to reduce false positives.

We have implemented a tool called DroidRacer. It generates execution traces by systematically testing Android applications and detects data races by computing the happens-before relation on the traces. We analyzed 15 Android applications including popular applications such as Facebook, Twitter and K-9 Mail. Our results indicate that data races are prevalent in Android applications, and that DroidRacer is an effective tool to identify data races.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Reliability, Verification

Keywords Data races, Android concurrency semantics, Happens-before reasoning

1. INTRODUCTION

Touchscreen mobile devices such as smartphones and tablets have become an integral part of our lives. These new devices have caught the imagination of the developer community and the end-users alike. We are witnessing a significant shift of computing from desktops to mobile devices.

While their hardware is limited in many ways, smartphones have succeeded in providing programming environments that enable development of efficient and feature-rich applications. These environments expose an expressive concurrency model that combines multi-threading and asynchronous event-based dispatch. In this model, multiple threads execute concurrently and, in addition, may post asynchronous tasks to each other. Asynchronous tasks running on the same thread may themselves be reordered non-deterministically subject to certain rules. While the model can effectively hide latencies, enabling innovative features, programming is complex and programs can have many subtle bugs due to non-determinism.

In this paper, we formalize the concurrency semantics of the Android programming model. Coming up with this formalization required a thorough study of the Android framework and a careful mapping of execution scenarios in Android to more formal execution traces. We view an Android application as comprising multiple *asynchronous tasks* that are executed on one or more threads. An asynchronous task, once started on a thread, runs to completion and can make both synchronous and asynchronous procedure calls. An asynchronous procedure call results in enqueuing of an asynchronous task to the task queue associated with the thread to which it is posted and control immediately returns to the caller. While the Android runtime environment creates some threads for the application initially, the application may also spawn threads. A newly spawned thread behaves as a usual thread, but additionally, it can attach a task queue to itself and receive asynchronous tasks for execution. Often, an application works with some threads with task queues and others without. In this work, we focus on the semantics of individual applications running within their own processes, and omit formalizing inter-process communication (IPC) between different applications.

Based on the concurrency semantics, we define a happens-before relation $\preceq$ over operations in execution traces [15]. A naïve combination of rules for asynchronous procedure calls and lock-based synchronization introduces spurious happens-before orderings. Specifically, it induces an ordering between two asynchronous tasks running on the same thread if they use the same lock. This is a spurious ordering since locks cannot enforce an ordering among tasks running sequentially on the same thread. We overcome this difficulty by decomposing the relation $\preceq$ into (1) a *thread-local happens-before relation* $\preceq_{st}$ which captures the ordering constraints between asynchronous tasks posted to the same thread and (2) an *inter-thread happens-before relation* $\preceq_{mt}$ which captures the ordering constraints among multiple threads. These relations are composed in such a way that the resulting relation captures the happens-before orderings in the Android concurrency model precisely.

We develop a data race detection algorithm based on the happens-before relation. A data race occurs if there are two accesses to the same memory location, with at least one being a write, such that there is no happens-before ordering between them. Race detection for multi-threaded programs is a well-researched topic (*e.g.*, [18, 21, 22, 25, 28]). Recently, race detection for *single-threaded* event-driven programs (also called *asynchronous programs*) is studied in the context of client-side web applications (*e.g.*, [20, 24, 30]). Unfortunately, race detection for Android applications requires reasoning about both thread interleavings and event dispatch; ignoring one or the other leads to false positives. Our happens-before relation generalizes these, so far independently studied, happens-before relations for multi-threaded programs and single-threaded event-driven programs, enabling precise race detection for Android applications.

PLDI'14, June 9 – June 11, 2014, Edinburgh, United Kingdom.
Copyright © 2014 ACM 978-1-4503-2784-8/14/06…$15.00.
http://dx.doi.org/10.1145/2594291.2594311

We have implemented our race detection algorithm in a tool called DroidRacer. DroidRacer provides a framework that generates UI events to systematically test an Android application. It runs *unmodified* binaries on an instrumented Dalvik VM and instrumented Android libraries. A run of the application produces an execution trace, which is analyzed offline for data races by computing the happens-before relation. The control flow between different procedures of an Android application is managed to a large extent by the Android runtime through callbacks. DroidRacer uses a model of the Android runtime environment to reduce false positives that would be reported otherwise. Further, DroidRacer assists in debugging the data races by classifying them based on criteria such as whether one involves multiple threads posting to the same thread or two co-enabled events executing in an interleaved manner.

We analyzed 10 open-source Android applications together comprising 200K lines of code, and used them to improve the accuracy of DroidRacer. We then applied DroidRacer on 5 proprietary applications including popular and mature applications like Facebook and Twitter. Our results indicate that data races are prevalent in Android applications and that DroidRacer is an effective tool to identify data races. Of the 215 races reported by DroidRacer on 10 open source Android applications, 80 were verified to be true positives and 6 of these were found to exhibit bad behaviours.

In summary, this paper makes the following contributions:

- The first formulation of Android concurrency semantics.
- An encoding of the happens-before relation for Android which generalizes happens-before relations for multi-threaded programs and single-threaded event-driven programs.
- A tool for dynamic race detection augmented with systematic testing capabilities for Android applications, and the successful identification of data races in popular applications.

While we focus on Android, our concurrency model and the happens-before reasoning extends naturally to other environments that combine multi-threading with event-based dispatch, such as other smartphone environments, high performance servers [19], low-level kernel code [3], and embedded software [4, 14].

2. MOTIVATING EXAMPLE

We now present an Android application and explain an execution scenario to illustrate Android semantics. We model this scenario and a variant of it as execution traces and apply happens-before reasoning to them, highlighting the need for reasoning *simultaneously* about the thread-local and inter-thread happens-before constraints.

2.1 Music Player

Figure 1 shows part of the source code of a sample Android application. It downloads a music file from the network and then provides a PLAY button to play it. A progress bar continuously displays the progress in file download. The code defines two classes: `DwFileAct` and `FileDwTask`. `DwFileAct` provides the user interface; it is a subclass of `Activity` (a base class provided by Android to manage user interactions). `FileDwTask` is a subclass of `AsyncTask` (a base class provided by Android to perform background operations asynchronously) and performs file download in a background thread. The method `onPlayClick` is a handler for the `onClick` event on the PLAY button and is registered via an XML manifest file (not shown). The other methods are the callbacks used by the Android runtime to manage the application. We discuss their roles subsequently.

2.2 Execution Scenario

We start with some background on the Android runtime environment. Each application in Android runs in its own process. A *system process* runs various services to manage the lifecycle of applications and to process system and sensor (*e.g.*, GPS, battery) events. In our present discussion, only the `ActivityManagerService` component of the system process, which governs the lifecycle callbacks of various components of application, is of relevance.

```
public class DwFileAct extends Activity {
  boolean isActivityDestroyed = false;

  protected void onResume( ) {
    super.onResume( );
    new FileDwTask(this).execute("http://abc/song.mp3");
  }
  public void onPlayClick(View v) {
    Intent intent = new Intent(this, MusicPlayActivity.class);
    intent.putExtra("file", "/sdcard/song.mp3");
    startActivity(intent);
  }
  protected void onDestroy( ) {
    super.onDestroy( );
    isActivityDestroyed = true;
  }
  ...
}

public class FileDwTask extends AsyncTask<String, Integer, Void> {
  DwFileAct act;
  ProgressDialog dialog;

  public FileDwTask(DwFileAct act) {
        this.act = act;
  }
  protected void onPreExecute( ) {
    super.onPreExecute( );
    dialog = new ProgressDialog(act);
    ...
    dialog.show( );
  }
  protected Void doInBackground(String... params) {
    InputStream input = ...
    ...
    byte data[] = new byte[1024];
    long progress = 0;
    int count;
    while ((count = input.read(data)) != −1) {
      progress += count;
      assertTrue(!act.isActivityDestroyed);
      publishProgress(progress);
    }
    ...
    return null;
  }
  protected void onProgressUpdate(Integer... progress) {
    super.onProgressUpdate(progress);
    dialog.setProgress(progress[0]);
  }
  protected void onPostExecute(Void result) {
    super.onPostExecute(result);
    assertTrue(!act.isActivityDestroyed);
    dialog.dismiss( );
    Button btn = (Button) act.findViewById(R.id.playBtn);
    btn.setEnabled(true);
  }
  ...
}
```

Figure 1. Code snippet of a music player application.

Let us consider the sequence of high-level actions that take place when a user launches the application. Figure 2 shows 4 threads that are involved: (1) a thread executing `ActivityManagerService` in the system process, and the following 3 threads running in the application's process: (2) a binder thread from a thread pool that handles communication with `ActivityManagerService`, (3) the main thread which also handles the UI, and (4) a background thread created dynamically by the runtime for executing `FileDwTask`.

After initialization, the main thread attaches a task queue to itself (step 2), making it eligible to receive asynchronous call requests. `ActivityManagerService` schedules the launch of the main activity (`DwFileAct` in this case). This results in the binder thread posting an asynchronous call denoted by `LAUNCH_ACTIVITY` to the main thread on behalf of `ActivityManagerService` (steps 4 and 5). The handler for `LAUNCH_ACTIVITY` synchronously calls the lifecycle callbacks required by the Android runtime: `onCreate`, `onStart`, and `onResume` (steps 6.1–6.3). The procedure `onResume`

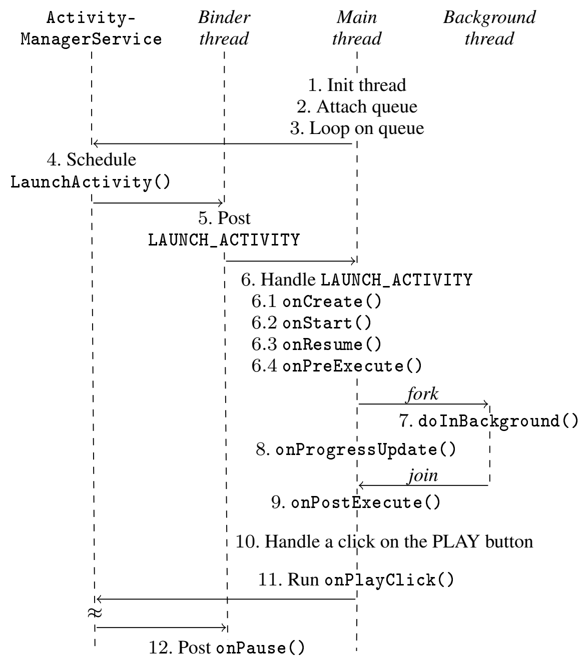


Figure 2. An execution scenario for the music player application: The solid edges indicate inter-thread communication.

starts the asynchronous task `FileDwTask` (line 6 in Figure 1). This results in execution of procedure `onPreExecute` of `FileDwTask` (step 6.4) followed by creation of a new thread on which the procedure `doInBackground` is executed (step 7). This procedure downloads the file and indicates the progress in a progress bar on the UI through the call to `publishProgress` (line 42 in Figure 1). The runtime in turn runs a procedure `onProgressUpdate` on the main thread (step 8). Once `doInBackground` finishes, the runtime calls `onPostExecute` (step 9). This procedure enables the PLAY button (lines 55–56 in Figure 1).

Now, suppose the user clicks on the PLAY button to play the downloaded file. The task scheduler of the main thread (called the "looper") processes this event (step 10), and posts and later runs the handler `onPlayClick` (step 11). The handler `onPlayClick` starts another activity (line 11 in Figure 1). This results in the component `ActivityManagerService` scheduling the callback `onPause` of the currently visible activity (the activity `DwFileAct` here). Similar to the posting of `LAUNCH_ACTIVITY` (step 5), the binder thread posts `onPause` to the main thread on behalf of `ActivityManagerService` (step 12).

Even this simple execution scenario for the music player application involves four threads running in two different processes. Moreover, much of the complex control flow and inter-thread communication in this example is managed by the Android runtime itself and is opaque to the developer. Nevertheless, the developer must understand the semantics clearly to avoid concurrency bugs.

2.3 Execution Trace

For the purposes of analysis, we model the execution scenarios of Android applications more transparently as sequences of low-level concurrency-relevant operations, called *execution traces*.

Figure 3 shows a partial execution trace corresponding to the scenario in Figure 2. The threads of control `t0`, `t1`, and `t2` correspond respectively to the binder, main, and background threads. The op-codes always take the identifier of the executing thread as their first parameter. The op-codes `threadinit` and `threadexit` mark the start and finish of the thread. `attachQ` indicates that the thread has attached a task queue to itself.

An asynchronous procedure call is indicated by `post`, whose second argument is the procedure to be run asynchronously and its

	Thread t0 (Binder)	Thread t1 (Main)	Thread t2 (Bkgnd Task)
1		threadinit(t1)	
2		attachQ(t1)	
3		loopOnQ(t1)	
4		enable(t1,LAUNCH_ACTIVITY)	
5	post(t0,LAUNCH_ACTIVITY,t1)		
6		**begin**(t1,LAUNCH_ACTIVITY)	
7		write(t1,DwFileAct-obj)	
8		fork(t1,t2)	(*a*)
9		enable(t1,onDestroy)	
10		**end**(t1,LAUNCH_ACTIVITY)	
11			threadinit(t2)
12		(*c*)	read(t2,DwFileAct-obj)
13			(*b*) post(t2,onPostExecute,t1)
14			threadexit(t2)
15		**begin**(t1,onPostExecute)	
16		read(t1,DwFileAct-obj)	
17		enable(t1,onPlayClick)	
18	(*d*)	**end**(t1,onPostExecute)	
19		post(t1,onPlayClick,t1)	
20		**begin**(t1,onPlayClick)	
21	(*e*)	enable(t1,onPause)	
22		**end**(t1,onPlayClick)	
23	post(t0,onPause,t1)		
24		...	

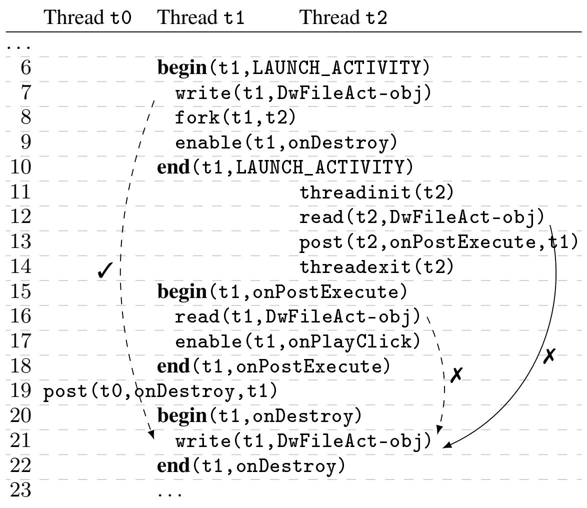
Figure 3. Execution trace corresponding to Figure 2 and some thread-local (dashed) and inter-thread (solid) happens-before edges.

third argument the target thread that will run it, that is, the task is posted to the task queue of the target thread. For simplicity, we omit the arguments to procedures, including the receiver object. The processing of the task queue of a thread begins after the thread executes `loopOnQ`. `fork` indicates spawning of a thread.

The trace in Figure 3 comprises three asynchronous tasks, each enclosed in a pair of `begin` and `end` operations. The first asynchronous task (operations 6–10) corresponds to step 6 of Figure 2. We focus on data races and abstract some aspects of the computation for readability. First, we abstract the actual computations and only track accesses (reads and writes) to memory locations. In this example, for brevity, only the accesses to the `DwFileAct` object, abbreviated as `DwFileAct-obj` in the trace, are presented. Second, we omit the call stack of the thread and synchronously executed procedures —such as the callback methods `onCreate`, `onStart`, `onResume`, and `onPreExecute` (steps 6.1–6.4 in Figure 2)— that do not have interesting concurrency behavior.

The initialization of the field `isActivityDestroyed` at line 2 in Figure 1 results in the `write` operation 7 in the trace. The thread created by the Android runtime for executing the asynchronous task (indicated by the edge labeled *fork* in Figure 2) results in the `fork` operation (*i.e.*, operation 8). Once the processing of `LAUNCH_ACTIVITY` is complete, the Android runtime semantics mandates that the activity thus created may get destroyed at any time (based on user actions or runtime indicators such as the system running out of memory). This is made explicit in the trace through the `enable` operation (*i.e.*, operation 9). The operation `enable(t,c)` indicates that the callback `c` for some environment event is enabled, that is, the application can now accept an event that causes `c` to execute. For instance, operation 4 enables the launch of the activity. The `enable` operations do not change program state, but are used in our approach to model the behavior of the Android runtime environment for more accurate reasoning.

The `read` operation at position 12 in the trace results from the assertion evaluation in the procedure `doInBackground` at line 41 in Figure 1. As mentioned earlier, the procedure indicates the progress in file download on a progress bar. Before updating the progress bar, it ensures that the corresponding activity is alive (*i.e.*, not destroyed) by checking the boolean field `isActivityDestroyed`. Once the



```
      Thread t0     Thread t1          Thread t2
...
 6                  begin(t1,LAUNCH_ACTIVITY)
 7                    write(t1,DwFileAct-obj)
 8                    fork(t1,t2)
 9                    enable(t1,onDestroy)
10                  end(t1,LAUNCH_ACTIVITY)
11                                     threadinit(t2)
12                                     read(t2,DwFileAct-obj)
13                                     post(t2,onPostExecute,t1)
14              ✓                      threadexit(t2)
15                  begin(t1,onPostExecute)
16                    read(t1,DwFileAct-obj)
17                    enable(t1,onPlayClick)
18                  end(t1,onPostExecute)   ✗              ✗
19    post(t0,onDestroy,t1)
20                  begin(t1,onDestroy)
21                    write(t1,DwFileAct-obj)
22                  end(t1,onDestroy)
23                  ...
```

Figure 4. Partial execution trace for the scenario in which the user clicks the BACK button instead of the PLAY button.

procedure `doInBackground` completes, the runtime posts a call to `onPostExecute` on the main thread as indicated by operation 13. The asynchronous task corresponding to `onPostExecute` at operations 15–18 contains a similar `read` operation and the enabling of click event to be handled by `onPlayClick`. Step 10 in Figure 2 indicates that the PLAY button is clicked, causing the main thread to post (operation 19) and then execute `onPlayClick` (operations 20–22). As a new activity is being started (line 11 in Figure 1), the callback event `onPause` of the currently visible activity object `DwFileAct-obj` is enabled at operation 21 which is posted by the runtime through binder thread subsequently (see operation 23).

2.4 Data Races and Happens-before Reasoning

We now analyze the execution trace in Figure 3 for data races. Two operations *conflict* if they refer to the same memory location and at least one of them is a write. In our trace, there are two pairs of conflicting operations: $(7, 12)$ and $(7, 16)$. Even though the operations 7 and 16 execute on the same thread `t1`, on a thread with a task queue such as `t1`, we cannot derive a happens-before ordering between two operations unless they execute in the same asynchronous task or the tasks themselves have a happens-before ordering. In the absence of an ordering, the execution order of the asynchronous tasks (and consequently, the conflicting operations) is *non-deterministic*, and we would have a data race. Data races of this form in single-threaded event-driven programs were identified for client-side web applications in [9, 20, 24, 30].

Inter-dependent reasoning. In our setting, the rules for single-threaded event-driven programs are insufficient, and we now show that our analysis must account for the combination of events (asynchronous calls) and multi-threading. The edge a in Figure 3 models the ordering between the `fork` operation and start of the forked thread, and through transitivity ensures that there is no data race between operations 7 and 12 (executing on different threads). The asynchronous procedure call semantics guarantees the ordering between a `post` operation and the corresponding `begin` operation (see edge b in Figure 3). Transitivity through edges a and b ensures that the `fork` operation happens-before `begin` (*i.e.*, operation 15). In Android, tasks run to completion and are not pre-empted. Thus, the task containing the `fork` operation executes completely before the task beginning at operation 15 starts. We can therefore construct the thread-local edge c between the two asynchronous tasks and can infer that there is no data race between operations 7 and 16.

We note that the *thread-local* edge c cannot be derived unless we reason about the happens-before relations for both the multi-threaded case and the asynchronous case (*e.g.*, no pre-emption semantics) simultaneously. Besides, as pointed above, some rules such as program order for operations running on the same thread take a different interpretation in our setting.

Table 1: List of operations (Thread `t` is currently executing.)

`threadinit(t)`	start executing current thread
`threadexit(t)`	complete executing current thread
`fork(t,t′)`	create thread `t′`
`join(t,t′)`	consume the completed thread `t′`
`attachQ(t)`	attach a task queue to thread `t`
`loopOnQ(t)`	begin executing procedures in `t`'s queue
`post(t,p,t′)`	post task `p` asynchronously to thread `t′`
`begin(t,p)`	start executing the posted task `p`
`end(t,p)`	end executing the posted task `p`
`acquire(t,l)`	`t` acquires lock `l`
`release(t,l)`	`t` releases lock `l`
`read(t,m)`	read memory location `m`
`write(t,m)`	write memory location `m`
`enable(t,p)`	enable posting of task `p`

Modeling the runtime environment. While we do not explicitly model `ActivityManagerService` (running in the system process) in the execution trace, we capture its effects through the `enable` operations on callback procedures.[1] This helps us identify the ordering constraints for lifecycle callbacks made by the Android environment (*e.g.*, see the edge e in Figure 3). Through `enable` operations, we also capture the ordering between operations in the trace and UI callbacks (*e.g.*, see the edge d in Figure 3). These edges are crucial for avoiding false positives.

As an example, let us assume that the steps 1–8 take place similar to Figure 2, but instead of clicking the PLAY button (step 9), suppose the user presses the BACK button. Figure 4 shows the resulting trace with the first 18 operations (some of them elided) same as Figure 3. The user action results in the activity being removed from the screen (but not garbage-collected), and `ActivityManagerService` posts a call to the `onDestroy` callback (operation 19 in Figure 4). This callback executes next (operations 20–22), and as seen in line 15 of Figure 1, writes into the field `isActivityDestroyed`.

Due to the happens-before ordering between `enable` and `post` (operations 9 and 19), and `post` and `begin` (operations 19 and 20), there is a happens-before edge between the `write` operations 7 and 21 and they do not constitute a data race. Without the `enable` operation, which specifies the *environment restriction* that `onDestroy` can only be called *after* `LAUNCH_ACTIVITY` finishes, we could not have derived the required happens-before ordering between operations 7 and 21, resulting in a false positive.

Two races. Again, we consider the trace in Figure 4. The callback `onDestroy` is enabled while thread `t2` is running and if fired, it executes on thread `t1`. Thus, the `read` and `write` operations 12 and 21 *may happen in parallel* giving rise to a potential data race.

In Android, two asynchronous calls, posted on the same thread, execute in the order in which they are posted. We refer to this as the *FIFO (first-in first-out) semantics*.[2] In our example, there is no happens-before ordering between the two `post` operations 13 and 19. Therefore, the asynchronous tasks executing the `read` and `write` operations 16 and 21 also are not ordered and give rise to another potential data race but this time between two operations on the *same* thread. As operation 21 (line 15 in Figure 1) sets `isActivityDestroyed` to `true`, if the operations in any of the two racey pairs $(12, 21)$ and $(16, 21)$ are reordered then the corresponding assertions at lines 41 and 53 in Figure 1 would fail.

[1] In practice, explicitly tracking the system process (which runs `ActivityManagerService` among others) is both difficult and inefficient.

[2] There are certain exceptions to the FIFO semantics which we discuss later in Section 4.2.

$$(\textsc{Start})\ \frac{}{\mathcal{C} \leftarrow \{\mathtt{t} \mid \mathtt{t} \in \mathit{Threads}\} \quad \mathcal{R} \leftarrow \emptyset \quad \mathcal{F} \leftarrow \emptyset \quad \mathcal{B} \leftarrow \emptyset \quad \mathcal{E} \leftarrow \mathit{Init} \quad \mathcal{Q}(\mathtt{t}) \leftarrow \epsilon, \mathcal{L}(\mathtt{t}) \leftarrow \emptyset \text{ for all } \mathtt{t} \in \mathcal{C}}$$

$$(\textsc{Fork})\ \frac{\alpha = \mathtt{fork}(\mathtt{t},\mathtt{t}') \quad \mathtt{t} \in \mathcal{R} \quad \mathtt{t}' \text{ is a fresh thread-id}}{\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathtt{t}'\} \quad \mathcal{E}(\mathtt{t}') \leftarrow \mathtt{main} \quad \mathcal{Q}(\mathtt{t}') \leftarrow \epsilon \quad \mathcal{L}(\mathtt{t}') \leftarrow \emptyset} \qquad (\textsc{Join})\ \frac{\alpha = \mathtt{join}(\mathtt{t},\mathtt{t}') \quad \mathtt{t} \in \mathcal{R} \quad \mathtt{t}' \in \mathcal{F}}{}$$

$$(\textsc{Init})\ \frac{\alpha = \mathtt{threadinit}(\mathtt{t}) \quad \mathtt{t} \in \mathcal{C}}{\mathcal{C} \leftarrow \mathcal{C} \setminus \{\mathtt{t}\} \quad \mathcal{R} \leftarrow \mathcal{R} \cup \{\mathtt{t}\}} \qquad (\textsc{Exit})\ \frac{\alpha = \mathtt{threadexit}(\mathtt{t}) \quad \mathtt{t} \in \mathcal{R}}{\mathcal{R} \leftarrow \mathcal{R} \setminus \{\mathtt{t}\} \quad \mathcal{F} \leftarrow \mathcal{F} \cup \{\mathtt{t}\}}$$

$$(\textsc{Acquire})\ \frac{\alpha = \mathtt{acquire}(\mathtt{t},\mathtt{l}) \quad \mathtt{t} \in \mathcal{R} \quad \mathtt{l} \notin \mathcal{L}(\mathtt{t}') \text{ for any } \mathtt{t}' \neq \mathtt{t}}{\mathcal{L}(\mathtt{t}) \leftarrow \mathcal{L}(\mathtt{t}) \cup \{\mathtt{l}\}} \qquad (\textsc{Release})\ \frac{\alpha = \mathtt{release}(\mathtt{t},\mathtt{l}) \quad \mathtt{t} \in \mathcal{R} \quad \mathtt{l} \in \mathcal{L}(\mathtt{t})}{\mathcal{L}(\mathtt{t}) \leftarrow \mathcal{L}(\mathtt{t}) \setminus \{\mathtt{l}\}}$$

$$(\textsc{AttachQ})\ \frac{\alpha = \mathtt{attachQ}(\mathtt{t}) \quad \mathtt{t} \in \mathcal{R} \quad \mathcal{Q}(\mathtt{t}) = \epsilon}{q \text{ is a fresh task queue} \quad \mathcal{Q}(\mathtt{t}) \leftarrow q} \qquad (\textsc{Post})\ \frac{\alpha = \mathtt{post}(\mathtt{t},\mathtt{p},\mathtt{t}') \quad \mathtt{t},\mathtt{t}' \in \mathcal{R} \quad \mathcal{Q}(\mathtt{t}') \neq \epsilon}{\mathcal{Q}(\mathtt{t}') \leftarrow \mathcal{Q}(\mathtt{t}') \oplus \mathtt{p}}$$

$$(\textsc{LoopOnQ})\ \frac{\alpha = \mathtt{loopOnQ}(\mathtt{t}) \quad \mathtt{t} \in \mathcal{R} \setminus \mathcal{B} \quad \mathcal{Q}(\mathtt{t}) \neq \epsilon}{\mathcal{B} \leftarrow \mathcal{B} \cup \{\mathtt{t}\} \quad \mathcal{E}(\mathtt{t}) \leftarrow \perp} \qquad (\textsc{Sequencing})\ \frac{\sigma \xrightarrow{\alpha} \sigma' \quad \sigma' \xrightarrow{\beta} \sigma''}{\sigma \xrightarrow{\alpha;\beta} \sigma''}$$

$$(\textsc{Begin})\ \frac{\alpha = \mathtt{begin}(\mathtt{t},\mathtt{p}) \quad \mathtt{t} \in \mathcal{R} \cap \mathcal{B} \quad \mathcal{E}(\mathtt{t}) = \perp \quad \mathtt{p} = \mathsf{Front}(\mathcal{Q}(\mathtt{t}))}{\mathcal{Q}(\mathtt{t}) \leftarrow \mathcal{Q}(\mathtt{t}) \ominus \mathtt{p} \quad \mathcal{E}(\mathtt{t}) \leftarrow \mathtt{p}} \qquad (\textsc{End})\ \frac{\alpha = \mathtt{end}(\mathtt{t},\mathtt{p}) \quad \mathtt{t} \in \mathcal{R} \cap \mathcal{B} \quad \mathcal{E}(\mathtt{t}) = \mathtt{p}}{\mathcal{E}(\mathtt{t}) \leftarrow \perp}$$

Figure 5. Semantic rules.

3. ANDROID CONCURRENCY SEMANTICS

In this section, we formalize the concurrency semantics of Android applications. An *Android application* $\mathcal{A}$ can be seen as a triple $(\mathit{Threads}, \mathit{Procs}, \mathit{Init})$ where $\mathit{Threads}$ is a finite set of threads created by the framework to run the application (*e.g.*, the main thread and the binder threads), Procs is a set of procedures, and Init is a mapping from $\mathit{Threads}$ to Procs indicating the procedure that is to be scheduled on each thread initially. The application can create new threads dynamically. A procedure p consists of its signature, variable declarations, and an ordered list of statements.

We only present the essential low-level operations relating to concurrency and event-handling, and omit the (standard) sequential programming features of the Android programming model. The sequential part of Android is an object-oriented language based on Java, and its object-based semantics can be defined similar to that of Java [1, 10]. Since we are interested in identifying data races, instead of modeling the effect of a computation on an object, we merely note whether there is a read or write access to it. We refer to heap-allocated objects as *memory locations*.

Our core language has dynamically allocated *threads*, *task queues* associated with threads, *asynchronous calls* made by posting a task to the task queue of a thread, *synchronization* via locks, and reads from and writes to *shared memory* locations. Table 1 decribes the operations in our language. The environment can trigger any of the enabled events. The `enable` operation is used to indicate the handlers of the events being enabled.

In order to formally specify the semantics of these operations, we define the notion of the state of an application. The *state* $\sigma = (\mathcal{C}, \mathcal{R}, \mathcal{F}, \mathcal{B}, \mathcal{E}, \mathcal{Q}, \mathcal{L})$ of an application $\mathcal{A} = (\mathit{Threads}, \mathit{Procs}, \mathit{Init})$ consists of (1) a set $\mathcal{C}$ (of thread-ids) of threads that are created but not scheduled for execution so far, (2) a set $\mathcal{R}$ of threads that are running, (3) a set $\mathcal{F}$ of threads that have completed execution, (4) a set $\mathcal{B}$ of threads which have begun processing their task queues, (5) a mapping $\mathcal{E} : \mathcal{C} \cup \mathcal{R} \rightarrow \mathit{Procs} \cup \{\perp\}$ indicating which procedure in Procs is executing (or shall execute) on a thread where $\perp$ indicates that the thread is idle, (6) a mapping $\mathcal{Q} : \mathcal{C} \cup \mathcal{R} \rightarrow Q \cup \{\epsilon\}$ associating a task queue with each thread where Q is a set of task queue objects which support enqueue and dequeue operations (denoted as $\oplus$ and $\ominus$ respectively) with FIFO semantics and ϵ is a queue of capacity zero, and finally, (7) a mapping $\mathcal{L} : \mathcal{C} \cup \mathcal{R} \rightarrow 2^L$ giving the locks held by a thread with L being the set of locks. For brevity, we do not explicitly model the program counter or the control stack (for synchronous procedure calls) of a thread. These details can be added in a straightforward manner.

Let $S_{\mathcal{A}} = (\Sigma, \rightarrow, \sigma_0)$ be the *transition system* for an application $\mathcal{A}$ where Σ is the set of states, $\rightarrow \subseteq \Sigma \times \mathit{Ops}(\mathcal{A}) \times \Sigma$ is the set of transitions between states according to the operations in the application $\mathcal{A}$ (denoted as $\mathit{Ops}(\mathcal{A})$), and σ_0 is the initial state. Figure 5 gives the semantic rules for different types of operations. A triple $(\sigma, \alpha, \sigma')$ belongs to $\rightarrow$ (written $\sigma \xrightarrow{\alpha} \sigma'$) iff the antecedent conditions of a rule for α hold in σ, and σ' is obtained through updates to σ as described in the consequent of the rule. Any component not updated explicitly in the consequent remains unchanged. In these rules, we use t and t$'$ to indicate thread-ids, p to indicate a procedure, and l to denote a lock. The operations `read`, `write`, and `enable` do not affect the state of the application (as defined above by us) and hence we do not give specific semantic rules for them.

The rule START defines the initial state σ_0 of the transition system $S_{\mathcal{A}}$. It adds the threads in $\mathit{Threads}$ to $\mathcal{C}$. Recall that Init gives the mapping from threads to procedures to be run initially. $\mathcal{E}$ is therefore initialized to Init. In the beginning, no thread is associated with a task queue or owns a lock. This is modeled by setting $\mathcal{Q}(\mathtt{t})$ to ϵ and $\mathcal{L}(\mathtt{t})$ to $\emptyset$ for each thread $\mathtt{t} \in \mathcal{C}$. A `fork` operation creates a new thread. The default procedure to execute on the thread is denoted by `main` in the rule FORK. The `join` operation requires its second argument to have finished execution (see the rule JOIN). The operation `threadinit` starts executing a thread and `threadexit` moves the thread to the set $\mathcal{F}$. Note that the maps $\mathcal{E}$, $\mathcal{Q}$, and $\mathcal{L}$ are defined only for threads that are either created or running. A thread t may acquire or release locks which we track in the lock set $\mathcal{L}(\mathtt{t})$ (see the rules ACQUIRE and RELEASE).

A thread without a task queue can attach a task queue to itself through the `attachQ` operation (see the rule ATTACHQ). Even though the task queue can start receiving the asynchronous call requests immediately after `attachQ` (see the rule POST), the thread begins processing them only after executing `loopOnQ` (see the rule LOOPONQ). The operation `loopOnQ` adds the thread to the set $\mathcal{B}$ and sets $\mathcal{E}(\mathtt{t})$ to $\perp$. A thread t is *idle* if $\mathcal{E}(\mathtt{t}) = \perp$. When a thread with a task queue becomes idle, it dequeues the task at the front of its queue (denoted by the function `Front` in the rule BEGIN) and executes it. If the task queue is empty, it waits. When a thread finishes executing an asynchronous task, it becomes idle (see the rule END) before

$$(\textsc{No-q-po})\ \frac{\begin{array}{c}\texttt{loopOnQ(t)} \notin \{\alpha_1,\ldots,\alpha_{i-1}\}\\ \mathit{thread}(\alpha_i)=\mathit{thread}(\alpha_j)=\texttt{t}\end{array}}{\alpha_i \preceq_{st} \alpha_j}$$

$$(\textsc{Async-po})\ \frac{\begin{array}{c}\texttt{loopOnQ(t)} \in \{\alpha_1,\ldots,\alpha_{i-1}\}\\ \mathit{task}(\alpha_i)=\mathit{task}(\alpha_j)=(\texttt{t},\_)\end{array}}{\alpha_i \preceq_{st} \alpha_j}$$

$$(\textsc{Enable-st})\ \frac{\alpha_i=\texttt{enable(t,p)} \qquad \alpha_j=\texttt{post(t,p,\_)}}{\alpha_i \preceq_{st} \alpha_j}$$

$$(\textsc{Post-st})\ \frac{\alpha_i=\texttt{post(t,p,t)} \qquad \alpha_j=\texttt{begin(t,p)}}{\alpha_i \preceq_{st} \alpha_j}$$

$$(\textsc{Fifo})\ \frac{\begin{array}{c}\alpha_i=\texttt{end(t,p1)}\\ \alpha_j=\texttt{begin(t,p2)} \qquad \texttt{post(\_,p1,t)} \preceq \texttt{post(\_,p2,t)}\end{array}}{\alpha_i \preceq_{st} \alpha_j}$$

$$(\textsc{Nopre})\ \frac{\begin{array}{c}\alpha_i=\texttt{end(t,p1)} \qquad \alpha_j=\texttt{begin(t,p2)}\\ \exists\alpha_k.\ \mathit{task}(\alpha_i)=\mathit{task}(\alpha_k) \wedge \alpha_k \preceq \texttt{post(\_,p2,t)}\end{array}}{\alpha_i \preceq_{st} \alpha_j}$$

$$(\textsc{Trans-st})\ \frac{\alpha_i \preceq_{st} \alpha_k \qquad \alpha_k \preceq_{st} \alpha_j}{\alpha_i \preceq_{st} \alpha_j}$$

Figure 6. Thread-local happens-before rules.

dequeuing the next available task. The rule SEQUENCING gives the combined state change due to two consecutive operations.

A sequence $\rho = \langle\alpha_1,\ldots,\alpha_n\rangle$ of operations is an *execution trace* of $\mathcal{A}$ if there exist states $\sigma_1,\ldots,\sigma_n$ such that $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \ldots \sigma_{n-1} \xrightarrow{\alpha_n} \sigma_n$ according to the rules in Figure 5.

4. RACE DETECTION

In this section, we define the happens-before relation for concurrent Android applications. We also present a precise modeling of the Android runtime environment to track happens-before relations more accurately, and an algorithm which detects and classifies data races by analyzing execution traces.

4.1 Happens-before Relation

Let $\rho = \langle\alpha_1,\ldots,\alpha_n\rangle$ be an execution trace of an application $\mathcal{A}$. Without loss of generality, we assume that a procedure is executed at most once in an execution trace. This assumption is met by uniquely renaming distinct occurrences of a procedure name in the trace.

For an execution trace ρ, we define a binary relation $\preceq$, called *happens-before*, over operations in $\{\alpha_1,\ldots,\alpha_n\}$. This relation must capture effects of multi-threading as well as asynchronous procedure calls. As noted in the Introduction, a naïve combination of rules for asynchronous procedure calls and lock-based synchronization introduces spurious happens-before orderings. We overcome this difficulty by decomposing the relation $\preceq$ into two relations: (1) a *thread-local happens-before relation* $\preceq_{st}$ and (2) an *inter-thread happens-before relation* $\preceq_{mt}$. Here, the subscripts *st* and *mt* respectively abbreviate “single-threaded” and “multi-threaded”. These are the smallest relations closed under the rules in Figure 6 and Figure 7 respectively. These relations are mutually recursive but impose the following restrictions: (1) for threads running asynchronous tasks, program order is restricted to individual asynchronous tasks, (2) an ordering between `acquire` and `release` operations on a lock is derived only if they run on two different threads, and (3) transitivity is defined in such a way that an ordering between asynchronous tasks running on the same thread and utilizing the same lock cannot be derived transitively through another thread utilizing that lock. Finally, the happens-before relation $\preceq$ is equal to $\preceq_{st} \cup \preceq_{mt}$.

Our rules use two helper functions *thread* and *task* to respectively obtain the thread that executes an operation α_i and, in the case of a thread with a task queue, to obtain the pair comprising the thread and the asynchronously called procedure to which α_i belongs. From now on, let α_i and α_j be operations in ρ such that $i < j$.

$$(\textsc{Attach-q-mt})\ \frac{\alpha_i=\texttt{attachQ(t)} \qquad \alpha_j=\texttt{post(t}',\texttt{\_,t)}}{\alpha_i \preceq_{mt} \alpha_j}$$

$$(\textsc{Enable-mt})\ \frac{\alpha_i=\texttt{enable(t,p)} \qquad \alpha_j=\texttt{post(t}',\texttt{p,\_)}}{\alpha_i \preceq_{mt} \alpha_j}$$

$$(\textsc{Post-mt})\ \frac{\alpha_i=\texttt{post(t}',\texttt{p,t)} \qquad \alpha_j=\texttt{begin(t,p)}}{\alpha_i \preceq_{mt} \alpha_j}$$

$$(\textsc{Fork})\ \frac{\alpha_i=\texttt{fork(t,t}'\texttt{)} \qquad \alpha_j=\texttt{threadinit(t}'\texttt{)}}{\alpha_i \preceq_{mt} \alpha_j}$$

$$(\textsc{Join})\ \frac{\alpha_i=\texttt{threadexit(t}'\texttt{)} \qquad \alpha_j=\texttt{join(t,t}'\texttt{)}}{\alpha_i \preceq_{mt} \alpha_j}$$

$$(\textsc{Lock})\ \frac{\alpha_i=\texttt{release(t,l)} \qquad \alpha_j=\texttt{acquire(t}',\texttt{l)}}{\alpha_i \preceq_{mt} \alpha_j}$$

$$(\textsc{Trans-mt})\ \frac{\alpha_i \preceq \alpha_k \qquad \alpha_k \preceq \alpha_j}{\alpha_i \preceq_{mt} \alpha_j}$$

Figure 7. Inter-thread happens-before rules.

Thread-local rules. Consider the thread-local happens-before rules given in Figure 6. In these rules, the two operations α_i and α_j are executed on the same thread, *i.e.*, $\mathit{thread}(\alpha_i) = \mathit{thread}(\alpha_j)$. If no task queue is attached to the thread or the processing of the task queue has not begun (by executing `loopOnQ`) until α_i is executed then the two operations have a happens-before relation $\alpha_i \preceq_{st} \alpha_j$ due to the *program order* (see the rule NO-Q-PO). This rule however is not applicable for the operations executed on the thread after the `loopOnQ` operation. In that case, they have a happens-before ordering (see the rule ASYNC-PO) if they execute in the same asynchronous task, *i.e.*, if $\mathit{task}(\alpha_i) = \mathit{task}(\alpha_j)$. If the trace contains an `enable` operation for a procedure then the subsequent `post` of the procedure, if any, happens after the `enable` operation as stated by ENABLE-ST. Further, the `begin` of an asynchronous task happens after the corresponding `post` as per the rule POST-ST.

The next two rules determine the happens-before ordering between a pair of *asynchronous tasks* running on the same thread. More specifically, the rules determine whether `end` of one task happens before `begin` of the other or not. The ordering between these operations, combined with program order (the rule ASYNC-PO) and transitivity (discussed shortly), implies that all operations in the first task happen before any operation in the second task.

The rule FIFO imposes a happens-before ordering between two operations `end(t,p1)` and `begin(t,p2)` if the corresponding `post` operations have a happens-before ordering *irrespective* of whether the `post` operations belong to the same thread or not. Note that the ordering between the `post` operations is in terms of the combined happens-before relation $\preceq$. This rule encodes the FIFO semantics of Android for asynchronous calls executed on the same thread.

In Android, the asynchronously called procedures are run to completion, *i.e.*, there is no pre-emption. The rule NOPRE captures this constraint. The operation `end(t,p1)` happens before `begin(t,p2)` if there is some operation α_k in the task `(t,p1)` which happens-before the `post` operation of `p2`. Here again, the `post` operation could take place from another thread and hence, the ordering between α_k and the `post` operation is over $\preceq$.

Finally, the rule TRANS-ST states that the thread-local relation $\preceq_{st}$ is transitive. It is also reflexive and anti-symmetric. These rules are not typeset in Figure 6 due to limitations of space.

Inter-thread rules. Consider the happens-before rules shown in Figure 7 that are applicable to operations α_i and α_j executed on different threads, *i.e.*, $\mathit{thread}(\alpha_i) \neq \mathit{thread}(\alpha_j)$. In these rules, we refer to two distinct threads by `t` and `t'`. A thread may `post` an asynchronous procedure call to another thread. The rule ATTACH-Q-MT states that any `post` to a thread happens only after the thread

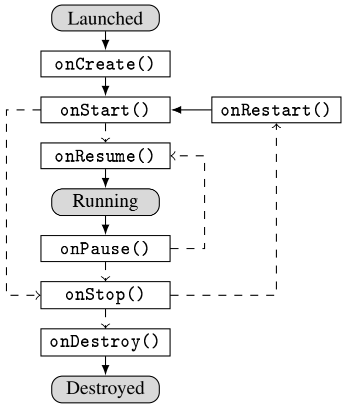


Figure 8. Partial lifecycle of an `Activity` component.

has been attached a task queue. The next two rules, ENABLE-MT and POST-MT are analogues of the corresponding thread-local rules. The rules for `fork`, `join`, and lock-based synchronization are as usual. Even if there is a happens-before ordering between two `post` operations but if they post the tasks to distinct threads, the executions of the tasks may interleave arbitrarily. Hence, there is no analogue of the rule FIFO for $\preceq_{mt}$. Similar is the case for the NOPRE rule.

The transitivity rule TRANS-MT permits transitive closure of the happens-before rule $\preceq$ but only if the operations α_i and α_j execute on distinct threads. Note that in the rule, the ordering between α_i and α_k or α_k and α_j may involve the thread-local ordering $\preceq_{st}$.

Specializations. The rules for *single-threaded event-driven programs* (*e.g.*, [24]) can be obtained by specializing the thread-local rules. Dropping the FIFO rule gives the non-deterministic scheduling semantics of asynchronous programs (*e.g.*, [8]). The happens-before rules for *multi-threaded programs* (where the threads do *not* have task queues) can be obtained by discarding all rules (both thread-local and inter-thread) for asynchronous procedure calls.

4.2 Precise Modeling of the Android Runtime Environment

The happens-before rules in Figures 6 and 7 do not capture all the constraints of the Android runtime environment. We now discuss modeling of two key aspects of the runtime environment, namely, (1) lifecycle callbacks and (2) management of asynchronous tasks. A precise modeling of these aspects is crucial to infer ordering among asynchronous tasks, thereby, avoiding false positives.

Lifecycle callbacks. As discussed in Section 2, the lifecycle of a component of an application is managed by the runtime environment by invoking callbacks in a specific order. Figure 8 shows a partial lifecycle state machine for the `Activity` class.[3] The gray nodes indicate the states of an activity and the other nodes are callback procedures. The solid edges indicate *must happen-after* ordering whereas the dashed edges indicate *may happen-after* ordering. If β may happen after α then in some (but not necessarily all) executions, we should see β after α, and there is no trace in which β happens before α. For example, `onStart` has two may-successors `onResume` and `onStop`. The former is called by the runtime environment if the activity comes to the foreground immediately after `onStart` finishes. The latter is called if the activity stays in the background. Similar lifecycles exist for other types of components in the Android programming model such as `Services` and `Broadcast Receivers`. Our implementation handles them but we omit the discussion on them due to space constraints. We refer the reader to the Android documentation for more details.

The operation `enable` in our core language is exploited to model the lifecycle constraints. Our implementation instruments the runtime in such a way that if a callback `C2` is expected to happen after a callback `C1` according to their lifecycle then the trace of `C1` contains an operation `enable(_,C2)`. The rules ENABLE-ST and ENABLE-MT establish the connections between the `enable` operations and subsequent `post` operations.

Task management. Apart from the `post` operation considered earlier, the Android runtime supports operations (1) to perform delayed posts in which a timeout is associated with an asynchronous task and the task is executed when the timeout occurs, (2) to cancel posted tasks, and (3) to post asynchronous tasks to the front of the queue overriding the FIFO semantics. Handling the first case requires only a slight modification to the FIFO rule. Let α_i = `end(t,p1)` and α_j = `begin(t,p2)` where `t` is a thread and `p1`, `p2` are two asynchronously called procedures. Let β_i and β_j be the respective post operations such that $\beta_i \preceq \beta_j$. We derive $\alpha_i \preceq_{st} \alpha_j$ if (a) β_j is a delayed post but β_i is not or (b) both are delayed posts and $\delta_i \leq \delta_j$ where δ_i and δ_j are respectively the timeouts used in β_i and β_j. The cancellation of posted tasks (the second case) is handled by removing the corresponding `post` operations from the trace. We defer the handling of posting-to-the-front (the third case) to future work.

4.3 Algorithm

Consider two operations α_i and α_j from an execution trace ρ such that $i < j$. We say that a *data race* exists between α_i and α_j if (1) they conflict and (2) $\alpha_i \not\preceq \alpha_j$ with respect to the trace ρ. Recall that two operations *conflict* if they access the same memory location and at least one of them is a `write` operation.

We implement a simple graph-based algorithm to detect data races. Given a trace $\rho = \langle \alpha_1, \ldots, \alpha_n \rangle$, it constructs a directed graph $G = (V, E)$ where $V = \{\alpha_1, \ldots, \alpha_n\}$ is the set of nodes and $(\alpha_i, \alpha_j) \in E$ is an edge iff $\alpha_i \preceq \alpha_j$. The edges are computed by performing a transitive closure which runs in time cubic in the length of the trace. Once all the edges are added to G, for each memory location `l` in the trace, our algorithm checks whether there is an edge between every pair of conflicting operations on `l`. If there is no edge then it reports a warning about a data race between the two operations. Note that our algorithm performs an offline analysis, and detects all races witnessed in the trace.

Classification of races. In order to assist the developer in understanding the root cause of a data race, our algorithm classifies the races in several categories by analyzing the trace.

For a `post` operation α_i, $callee(\alpha_i)$ is the task that α_i posts. Let $chain(\alpha_i) = \langle \beta_1, \ldots, \beta_m \rangle$ be the maximal sub-sequence of `post` operations in the trace ρ such that $callee(\beta_j) = task(\beta_{j+1})$ for $1 \leq j < m$ and $callee(\beta_m) = task(\alpha_i)$. Between β_j and β_k with $j < k$, β_k is called the *most recent* `post` operation.

Let α_i and α_j be two operations from the trace ρ involved in a data race where $i < j$. If $thread(\alpha_i) \neq thread(\alpha_j)$ then it is a *multi-threaded data race*. Otherwise, it is a *single-threaded data race*. Single-threaded data races are further categorized as:

- *Co-enabled*: Let β_i and β_j be the most recent `post` operations for environmental events, say e_i and e_j, from $chain(\alpha_i)$ and $chain(\alpha_j)$ respectively. If $\beta_i \not\preceq \beta_j$ then the race is called *co-enabled*. Debugging it would involve checking whether the events e_i and e_j are indeed co-enabled, that is, can they happen in parallel. Two UI events on the same screen or lifecycle callbacks of two distinct objects are examples of co-enabled events.
- *Delayed*: Let β_i and β_j be the most recent delayed posts in $chain(\alpha_i)$ and $chain(\alpha_j)$ respectively. The data race is called *delayed* if either (1) only β_i or β_j is defined (*i.e.*, there is a delayed post in only one of $chain(\alpha_i)$ or $chain(\alpha_j)$), or (2) $\beta_i \neq \beta_j$. This race would require inspecting timing constraints used in the delayed posts β_i and β_j for ruling it out.
- *Cross-posted*: Let β_i and β_j be the most recent `post` operations in $chain(\alpha_i)$ and $chain(\alpha_j)$ respectively such that β_i executes on a thread other than $thread(\alpha_i)$ and similarly, for β_j. If only one of them is defined or they are distinct then the race is called

[3] https://developer.android.com/guide/components/index.html

cross-posted. Resolving this race would require both thread-local and inter-thread reasoning.

The classification is performed by checking the criteria for the categories in the order in which they are presented above. We note that these criteria are not necessarily exhaustive but were found to describe root causes of most of the data races we observed. If none of the criteria is met then the data race is classifed as *unknown*.

5. IMPLEMENTATION

We have implemented our race detection technique in a tool called DROIDRACER that instruments and modifies Android 4.0. DROIDRACER has three components: (1) UI Explorer, (2) Trace Generator, and (3) Race Detector. DROIDRACER runs on an Android emulator along with the unmodified binary of the test application.

UI Explorer. The UI Explorer inspects UI related classes like `WindowManagerImpl` at runtime and obtains the events enabled on a screen for all widgets. It can generate several types of events, such as click, long-click, text input, screen rotation, and the BACK button press. For text fields, it can determine the required format of the input (*e.g.*, an email address) by inspecting flags associated with text fields. It supplies text of appropriate format from a manually constructed set of data inputs. DROIDRACER takes a bound k to limit the length of UI event sequences. The UI Explorer systematically generates event sequences of length k in a depth-first manner. The event sequences generated are stored in a database and used for backtracking and replay. In order to trigger an event only after the previous event is consumed and replay events consistently across testing runs, we have implemented checks that involved instrumenting various classes (*e.g.*, `Activity`, `View`, and `MessageQueue`).

Trace Generator. As the UI Explorer drives the test application, the Trace Generator logs the operations corresponding to our core language. We instrument the Dalvik bytecode interpreter to log `read`, `write`, `acquire`, `release` operations and track method invocations leading to the execution of application code. We only log object field accesses by application code. Whenever library methods are invoked from application code the accesses to receiver object of the method are also logged. While asynchrony related operations like `attachQ`, `loopOnQ`, `post`, `begin`, and `end` are emitted by instrumenting `MessageQueue` and `Looper` classes, the rest of the core operations except `enable` are tracked in Android's native code.

In order to establish ordering between lifecycle callbacks of application components such as `Activity`, one may need to track IPC calls, code being executed in the system process, or any other application process with which the application communicates. This however makes trace generation difficult and less efficient. We instead exploit `enable` operations to capture happens-before relation between such callbacks. We have extensively studied Android developer documentation, Android library codebase, and various execution scenarios to identify instrumentation sites to emit `enable` operations. Thus, our Trace Generator executes entirely within the test application's process and is efficient. The `enable` operations also help us establish logical ordering between UI event callbacks, capture relations between registering for a callback and execution of a callback (as in case of `BroadcastReceiver`, `IdleHandler`), or connect periodic execution of Java's `TimerTask` objects.

Race Detector. The Race Detector takes a trace generated by Trace Generator as input and constructs a happens-before graph. It detects and classifies races as described in Section 4.3.

6. EVALUATION

We applied DROIDRACER on 10 open-source applications and 5 proprietary applications from different categories like entertainment, communication, personalization, and social networking (see Table 2). These include popular, feature-rich applications like Twitter and Facebook with more than hundred million downloads each.

Table 2: Statistics about applications and traces.

Application (LOC)	*Trace length*	*Fields*	*Threads (w/o Qs)*	*Threads (w/ Qs)*	*Async. tasks*
Aard Dictionary (4044)	1355	189	2	1	58
Music Player (11012)	5532	521	3	2	62
My Tracks (26146)	7305	573	11	7	164
Messenger (27593)	10106	845	11	4	99
Tomdroid Notes (3215)	10120	413	3	1	348
FBReader (50042)	10723	322	14	1	119
Browser (30874)	19062	963	13	4	103
OpenSudoku (6151)	24901	334	5	1	45
K-9 Mail (54119)	29662	1296	7	2	689
SGTPuzzles (2368)	38864	566	4	1	80
Remind Me	10348	348	3	1	176
Twitter	16975	1362	21	5	97
Adobe Reader	33866	1267	17	4	226
Facebook	52146	801	16	3	16
Flipkart	157539	2065	36	3	105

Together, the open source applications comprise 200K lines of code (comments and blank lines omitted). The open-source applications are separated from proprietary ones by a horizontal rule in Table 2.

In our experiments, these applications were systematically tested with UI event sequences between 1 and 7 events. Some of the applications exhibited complex concurrent behavior by spawning many threads, starting `Service` components, and triggering `Broadcast Intents`, even before DROIDRACER triggered the first UI event on those applications. For such applications, we triggered sequences of 1–3 events only. For each application, DROIDRACER found tests which manifested one or more races. This shows that data races are prevalent in Android applications.

Table 2 gives statistics over one such representative test for each of these applications. The rows of Table 2 have been arranged in ascending order of trace length. The traces resulting from these tests comprise 1K to 157K operations (in our core language). The applications accessed thousands of memory locations in each run. Even if a field of a particular class is accessed through multiple objects in a trace, we report it only once under the column "Fields."

Table 2 shows that these applications make heavy use of multithreading and asynchrony. The highest number of threads were created by the Flipkart application (36 of them without task queues and 3 with queues). These numbers do *not* include the count of binder threads and other system threads created by the Android runtime for the applications (usually about 10–15). The applications also made many asynchronous calls (see the column "Async. tasks"), further adding to the non-determinism.

Performance. Trace generation causes a slowdown up to 5x due to instrumentation overhead. Race Detector constructs a graph representation of the trace with operations as nodes and edges added due to happens-before rules. As an optimization, contiguous memory accesses without any intervening synchronization operation are modeled by a single node in the graph. This reduced the number of nodes to 1.4% to 24.8% of the original trace length (with avg. 11.1%) without sacrificing on the precision. For example, after the optimization, the number of nodes in the graph representation of Flipkart application's trace was only 2.2K (compared to the trace length of 157K). Race Detector took a few seconds to a few hours to analyze traces and flag races while using up to 20 MB of memory.

The experiments were performed on an Intel Xeon E5-2450 2.10 GHz machine with 8 cores, 20 MB cache, and 250 GB SSD. DROIDRACER runs in Android emulator on a single core.

Data races. Table 3 gives the number of data races reported by DROIDRACER on the same traces as described in Table 2. It classifies the races into different categories as discussed in Section 4.3. If there are multiple races belonging to the same category on the same memory location, DROIDRACER reports any one of them randomly. Races for different objects of the same class are considered separately. In addition to the races presented in Table 3 under different

Table 3: Data races reported by DROIDRACER: The entries of the form "$X(Y)$" indicate the number of reports X generated by DROIDRACER and the number of true positives Y among them.

Application	*Multi-threaded*	*Single-threaded*		
		Cross-posted	*Co-enabled*	*Delayed*
Aard Dictionary	1 (1)	0	0	0
Music Player	0	17 (4)	11 (10)	4 (0)
My Tracks	1 (0)	2 (1)	1 (0)	0
Messenger	1 (1)	15 (5)	4 (3)	2 (2)
Tomdroid Notes	0	5 (2)	1 (0)	0
FBReader	1 (0)	22 (22)	14 (4)	0
Browser	2 (1)	64 (2)	0	0
OpenSudoku	1 (0)	1 (0)	0	0
K-9 Mail	9 (2)	0	1 (0)	0
SGTPuzzles	11 (10)	21 (8)	0	0
Total	27 (15)	147 (44)	32 (17)	6 (2)
Remind Me	0	21	33	0
Twitter	0	20	7	4
Adobe Reader	34	73	0	9
Facebook	12	10	0	0
Flipkart	12	152	84	30
Total	58	276	124	43

categories, DROIDRACER reported 3 races for Music Player (with 2 being true positive), 9 races for Adobe Reader, and 36 races for Flipkart in the *unknown* category (see Section 4.3).

We performed manual inspection to distinguish between true and false positives. Apart from core operations, DROIDRACER logs procedure calls made on each thread to help identify source code locations and call stacks leading to racey accesses. We classify only those reported races as *true positives* for which we could produce alternate ordering of racey memory accesses than the reported order in the trace. We used the `DDMS` debugger[4] of Android for this purpose as follows: (1) For *multi-threaded* and *cross-posted* races, stall certain threads using breakpoints, giving others the opportunity to progress or to enforce a different ordering of asynchronous procedure calls. (2) For *co-enabled* races, change the order of triggering events. (3) For *delayed* races, alter delay associated with asynchronous posts. Table 3 gives the number of reports generated by DROIDRACER and the number of true positives under each category for the open-source applications.

Open-source applications. We used the open-source applications to thoroughly evaluate DROIDRACER. Out of the total 215 reports (including the 3 *unknown* category races of Music Player) generated by DROIDRACER, 80 (37%) were confirmed to be true positives (with 2 of these from Music Player's *unknown* category). Thus, even in the challenging setting of Android's programming model, DROIDRACER could effectively find real races. Since this is the first work on race detection for Android, it is not possible to compare the precision of DROIDRACER with another tool. Below we present the 6 cases for which we observed *bad behavior* by exercising the races.

A multi-threaded race. In Aard Dictionary, DROIDRACER reported a race on an object of type `Service` which was responsible for loading dictionaries. The race involved two threads with one writing to the object (the main thread) and the other reading from it (a background thread) without synchronization. We could produce another trace in which the write causes a state change for the `Service` object. This temporarily permitted the background thread to access the (empty) dictionaries even before they were loaded. As a consequence, the word the user wanted to lookup could not be retrieved.

A single-threaded race. In the Messenger application, DROIDRACER reported a race on an object of type `Cursor` which holds a list obtained from a database. The race involved two asynchronous tasks running on the main thread with one of them being posted by another thread. These tasks did not have a happens-before relation between them. We could reorder the asynchronous tasks causing an "index of out of bounds" runtime exception on the `Cursor` object due to access to a list element deleted by the other task. This race belongs to the *cross-posted* category.

DROIDRACER also reported state altering races on `mDataValid` and `mRowIDColumn` fields of `CursorAdapter` class accessed during `Activity` launch and `Activity` exit in the Messenger application. `mDataValid` and `mRowIDColumn` were expected to be $false$ and -1 respectively after execution of `Activity`'s `onStop` lifecycle callback. Racey accessses (of the type *cross-posted*) set `mDataValid` to $true$ and `mRowIDColumn` to a row ID greater than -1.

In addition, when the asynchronous tasks containing racey access were reordered for validating the reported data races, the applications FBReader and Tomdroid Notes crashed (respectively with exceptions `BadTokenException` and `NullPointerException`) in one execution each.

Proprietary applications. On the proprietary applications, we found a total of 546 races (including 45 in the *unknown* category). Since we depend on manual inspection and a debugger to validate races, for the proprietary applications, in the absence of enough information (and the source code), we could not distinguish between true/false positives. From our experience with the open-source applications, we believe that the developers of these applications could utilize DROIDRACER's output to identify problematic cases.

False positives and negatives. DROIDRACER only tracks operations due to Java code, whereas some applications perform operartions using C/C++ code too. The high number of false positives reported for Browser is due to asynchronous posts by untracked natively-created (non-binder) threads. Applications like Messenger and FBReader use custom task queues implemented as list of `Runnables`. DROIDRACER would require a mapping of the high-level constructs (*e.g.*, adding and removing from the list) to the operations in our core language to apply happens-before reasoning. We did not modify Trace Generator to address these application-specific issues. Custom task queues can also cause false negatives, as DROIDRACER treats threads with custom queues as usual threads and applies the NO-Q-PO rule deriving spurious happens-before relations. The identification of instrumentation points to emit `enable` operations for lifecycle callbacks is challenging due to the lack of documentation. Missing `enable` operations might result in false positives whereas adding spurious ones will cause false negatives. Another cause of false positives is ad hoc synchronization, which can potentially be addressed using the notion of *race coverage* [24].

7. RELATED WORK

To the best of our knowledge, no previous paper presents (1) a formal concurrency semantics, (2) a happens-before relation, and (3) a race detection tool for the multi-threaded event-driven programming model of Android.

There are some tools that target specific types of concurrency bugs in Android. Dimmunix [12] is a tool to detect and recover from deadlocks. There are tools that check Android applications against specific thread usage policies; *e.g.*, Android's `StrictMode` tool dynamically checks that the UI thread does not perform I/O or other time-consuming operations, Zhang et al. [29] statically check that only the UI thread accesses UI objects.

The concurrency model exposed by Android is different compared to the models explored in the literature in the context of race detection. There is of course a large body of work on static and dynamic race detection techniques for multi-threaded programs, *e.g.*, based on locksets [5, 6, 17, 23, 25] or happens-before relations [7, 11], or their effective combinations [18, 22, 26–28]. However, these algorithms do not consider asynchronous calls, and either do not scale or produce many false positives, if asynchronous calls are simulated through additional threads [24]. Further, analyses based on locksets produce false positives because there may be no explicit locks and instead, the synchronization could be through ordering

[4] `http://developer.android.com/tools/debugging/ddms.html`

of events. For multi-threaded C programs, Kahlon et al. [13] statically infer the targets of asynchronous calls and callbacks through a pointer analysis. They however filter away races among procedures running on the same thread, and thereby, miss single-threaded races.

Recent work on race detection for client-side web applications [20, 24, 30] considers the happens-before relation for single-threaded event-driven programs and framework-specific rules to capture the semantics of browsers and JavaScript. However, their analysis is not immediately applicable to Android because there is additional interference through multi-threading. As discussed in Section 4.1, our definitions generalize the happens-before relations for both multi-threaded programs and single-threaded event-driven programs (modulo framework-specific rules).

Safety verification is undecidable for multi-threaded programs communicating via FIFO queues, and there are no software model checkers that understand this concurrency model. In general, static analysis is also challenging for Android due to the heavy use of reflection, native code, asynchrony, databases, and inter-process communication. Proprietary applications can be even more difficult to analyze statically not only because the source code is unavailable but also because the bytecode could be obfuscated. Most of these issues however are handled well by our technique since it works directly on unmodified binaries.

Finally, we compare our UI Explorer with related testing approaches. Our UI Explorer systematically explores and stores the UI event sequences by performing depth-first exploration of the UI widgets (in the style of stateless model checking). Dynodroid [16] is a testing tool for Android which randomly explores the UI events and unlike ours, does not provide easy replay. However, compared to our current implementation, Dynodroid can simulate intents (building blocks of IPC). Android `Monkey`[5] is a random event generator and lacks the ability to systematically explore the UI. AndroidRipper [2] performs a systematic UI exploration but requires an external framework to generate events. The Trace Generator and Race Detector components of DROIDRACER are independent of UI Explorer. We can potentially combine them with any available testing tools.

8. CONCLUSIONS

We presented a formal tool for concurrency analysis of Android applications, focusing on data race detection. The concurrency model of Android is more general than most models considered in the literature, and poses unique challenges for analysis. Our tool effectively found many races even in popular and mature applications.

Android is an expressive programming environment, and our formalization does not capture all its features. For example, we have not formalized its handling of inter-process communication (except IPCs relating to lifecycle events). DROIDRACER only generates UI events but not intents in the testing phase. Modeling and implementing these additional features are left for future work. We also wish to investigate how to provide better debugging support, *e.g.*, by analyzing the races that fall in the *unknown* category.

Acknowledgments. We thank Thejas Ramashekar and Maisagoni Chaithanya for helping us with the implementation and race analysis, as well as the anonymous reviewers for their useful feedback. This work is partially supported by the Indo-German Max Planck Center for Computer Science funded by DST, Government of India, BMBF, Government of Germany, and Max Planck Society.

References

[1] J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, 1999.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *ASE*, pages 258–261, 2012.

[3] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: safe asynchronous event-driven programming. In *PLDI*, pages 321–332. ACM, 2013.

[4] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *LCN*, pages 455–462, 2004.

[5] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[6] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI*, pages 219–232, 2000.

[7] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[8] P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *TOPLAS*, 34(1):6:1–6:48, 2012.

[9] J. Ide, R. Bodik, and D. Kimelman. Concurrency concerns in rich internet applications. In *EC(2), CAV Workshop*, 2009.

[10] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, pages 396–450, 2001.

[11] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Toward integration of data race detection in DSM systems. *J. Parallel Distrib. Comput.*, pages 180–203, 1999.

[12] H. Jula, T. Rensch, and G. Candea. Platform-wide deadlock immunity for mobile phones. In *DSN-W*, pages 205–210, 2011.

[13] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *ESEC/FSE*, pages 13–22, 2009.

[14] K. Klues, C. M. Liang, J. Paek, R. Musaloiu-Elefteri, P. Levis, A. Terzis, and R. Govindan. TOSThreads: Thread-safe and non-invasive preemption in TinyOS. In *SenSys*, pages 127–140, 2009.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, pages 558–565, 1978.

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

[17] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, pages 308–319, 2006.

[18] R. O'Callahan and J. Choi. Hybrid dynamic data race detection. In *PPOPP*, pages 167–178, 2003.

[19] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX*, pages 199–212, 1999.

[20] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *PLDI*, pages 251–262, 2012.

[21] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPOPP*, pages 179–190, 2003.

[22] E. Pozniansky and A. Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurr. Comput. : Pract. Exper.*, pages 327–340, 2007.

[23] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Practical static race detection for C. *TOPLAS*, 33(1):3:1–3:55, 2011.

[24] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *OOPSLA*, pages 151–166, 2013.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *TOCS*, pages 391–411, 1997.

[26] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *SOSP*, pages 369–384, 2011.

[27] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *FSE*, pages 205–214, 2007.

[28] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234, 2005.

[29] S. Zhang, H. Lü, and M. D. Ernst. Finding errors in multithreaded GUI applications. In *ISSTA*, pages 243–253, 2012.

[30] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *WWW*, pages 805–814, 2011.

[5] `http://developer.android.com/tools/help/monkey.html`