Race Detection for Android Applications

Tian Lan
Texas A&M University
welkinlan@gmail.com

Abstract
Programming environments have been developed to exhibit a concurrency model that combines asynchronous event-based dispatch and multi-threading. This model has been widely applied in most mobile platforms such as Android, iOS to provide natural support for a rich array of sensors and user input modalities. Even though 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.

This project aims at investigating on existing tools for data race detection on Android applications and provide a comprehensive analysis of the techniques adopted and implementation detail for the tools. In this project, two latest innovative techniques, namely, DroidRacer and CAFA, are explored to investigate on the concurrency semantics and languages used. Moreover, to provide with hands-on instructions, DroidRacer and EventRacer for Android are manually tested to further provide instructions on how to use the tools, for e.g., how to instrument the Android system and build the Android source code, etc.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verifica-tion; D.2.5 [Software Engineering]: Testing and Debugg-ing; F.3.1 [Logics and Meanings of Pro-grams]: Specifying and Verifying and Reasoning about Programs.

General Terms
Algorithms, Languages, Reliability, Verification

Keywords
Race detection, Event-driven systems, Android concurrency semantics, Android mobile application, Happens-before reasoning, Causality model, Use-free race

1. Introduction
Mobile computers are increasingly important computing platforms. The percentage of people owning a smart phone achieves 56% in 2013, and there are more than 1.7 million mobile applications available from Apple App Store or Google Play. For many people, phones and tablets are the primary platform for interacting with computer systems and the data they store.

Mainstream mobile devices often have a rich array of sensors and user input modalities, which provide huge asynchronicity to the input stream of mobile applications. As a result, event-driven concurrent programming models arise naturally among popular mobile platforms such as Android and iOS. An event-driven model makes it easy to integrate input from diverse sources such as touchscreens, accelerometers, microphones, and other sensors.

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 laten- cies, enabling innovative features, programming is complex and programs can have many subtle bugs due to non-determinism. Thousands of events may get executed every second in a mobile system. Even if these events get processed sequentially in one thread, most events are logically concurrent to each other, as they may not be ordered by any programmer specified ordering operations.

Unfortunately, most existing tools such as data-race detectors for finding concurrency errors assume a thread-based model. Naively applying these tools for event-driven mobile systems works poorly, because they implicitly assume that events handled in one thread are ordered by the program order.

Thus this project aims to look into exsisting tools for detecting data races for event-based systems, especially for Android, to investigate on the effective approach for Android application race detection. Until recently, very few papers (DroidRacer and CAFA) have been published focusing on race detection for Android, this project looks into both approaches by analysing the core techniques adopted and the implementations developed.

The rest of the project report is arranged as follows: section 2 introduces the background of Android system by anylysing its specific event-driven programming model and its lifecycle scheme; happens-before rule is introduced and a motiviting example is used to illustrate the scenario where data race happens in Android asynchrnonous environment. Section 3 looks deep into DroidRacer and CAFA and explains respectively about the core techniques and algorithms adopted. Section 4 gives the

implementation framework for DroidRacer and EventRacer for Android, provided with hands-on experience about how to implement and use the tool, for e.g., how to instrument and build the Android source code, etc.

2. Background and Motivation

This section provides background knowledge of Android event-driven programming model and looks into happens-before reasoning, which has been widely used in traditional race detection. The concept of Android activity lifecycle callbacks is introduced and a motivating example regarding it is used to illustrate a common scenario of data race in Android activity.

2.1 Android Event-driven Model

An android application consists of several threads, a subset of which are looper threads, which are used to manage events from their event queue. Below are the concepts for these fundamental components for Android applications:

- *Event*: An event could be generated by an entity external to an application (e.g., sensor input, network, operating system), or internally by a thread or an event executed in the application. An event is dispatched by the looper thread and handled by invoking its event handler based on its type.
- *Event queue*: Once an event is generated, it is placed in an event queue.
- *Looper thread:* Each looper thread is associated with one event queue. The role of a looper thread is to continuously check its event queue, select and process one event at a time.

Android application could be viewed as comprising multiple asynchronous tasks that are executed on one or more threads. An asynchronous task, or an event, once started on a thread, will run to completion and is able to make both asynchronous or synchronous 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. All events executed in a looper thread are atomic with respect to each other. We can assume that each event queue is managed by only one looper thread.

2.2 Happens-before Reasoning

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, however, race detection for Android applications requires consideration of both event dispatch and thread interleavings, or it will lead to false positives. Thus there is a need to invent new happens-before rules that could generalizes the happens-before relations for both multi-threaded programs and asynchronous event-driven programs, which guarantee more precise race detection for Android applications.

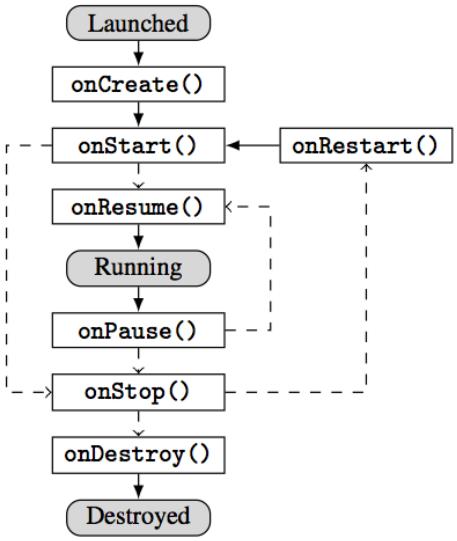

Figure 1 Android activity lifecycle

2.3 Android activity lifecycle callbacks

An activity is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI. Activity is the most used class in Android, and in the same time could be the most problematic one with concurrency issues, mainly because of the lifecycle callbacks. Figure 1 shows a life cycle state machine for Activity class. 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. For example, onStart() could be succeeded by onStop() if the app stays in the background; or by onResume() if the app appear in the foreground. This kind of non-determinism needs a precise modeling of these aspects, which is crucial to infer correct ordering among asynchronous events in order to avoid false positives.

2.4 A Motivating Example

In Android, event handlers are executed in response to messages arriving from various sources, including the framework, the network, the sensors, and the user interface. This section provides a motivating example (a GPS-based application to automatically show the trace of a user) to illustrate the scenario of two event handlers being executed in arbitrary order in the Android asynchronous environment.

Three callbacks in the app are selected to illustrate the non-determinism:

- *onResume() (*Figure 2*):* onResume() is invoked by the framework when the user starts interacting with the activity. This is a common place to register for periodic updates from sensors such as GPS and initialize a database connection.
- *onPause() (*Figure 3*) When* the application (for e.g. the map) goes into the background, the framework invokes onPause event handler. Here as a common rule, the developer removes location updates and closes the database.
- *onLocationChanged()* (Figure 4*):* After the application registers for location updates, the system periodically invokes provided callback mListener with the device's geographical location and saved into the database.

```
@Override
protected void onResume() {
  super.onResume();
  locationManager.requestLocationUpdates(GPS_PROVIDER, 0, 0, mListener);
  mDbHelper = new SQLiteOpenHelper(this, DB_NAME, DB_VERSION);
}
```

Figure 2 onResume() code

```
@Override
protected void onPause() {
  super.onPause();
  locationManager.removeUpdates(mListener);
  mDbHelper.close();
}
```

Figure 3 onPause() code

```
LocationListener mListener = new LocationListener() {
  @Override
  public void onLocationChanged(Location location) {
    //show location on map
    mDbHelper.getWritableDatabase().insert(loc);
  }
};
```

Figure 4 onLocationChanged() code

Intuitively, the onLocationChanged() callback will be executed after executing onResume() when the app comes foreground and before executing onPause when the app goes background. However, it could happen that after onPause() is called, that is when the developer has already explicitly removed location updates in callback, the framework executes onLocationChanged callback which will save data into a closed connection, which will throw the exception "java.lang.IllegalStateException: attempt to re-open an already-closed object: SQLiteDatabase" and then crash the application. The reason for this is the non determinism between the callbacks: onPause() and onLocationChanged(), sharing the variable mDBHeloper on the same memory location. With the techniques mentioned in the next section, this kind of data race will be detected after traversing through the happens-before graph.

3. Existing Techniques

After searching through the literature, only two papers (Pallavi 2014 and Chun-Hung 2014) have been published focusing on Android data race detection. Both methods follow the same pattern of race detection steps, which are:

- *Execution of application on instrumented Android system image*
- *Log the execution traces according to the operations defined in the language*
- *Apply offline algorithm on the execution traces to generate happens-before DAG according to defined happens-before rules.*
- *Reachability test to capture potential data races*

This section looks into the core techniques (language and happens-before rule) adopted by them respectively.

3.1 DroidRacer

DroidRacer is the first formulation of Android concurrency semantics. It provides 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. Language

DroidRacer uses its core language to map the operations with regard to:

- 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
- Reads from and writes to shared memory locations

Table 1 describes the operations in the language (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'
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
attachQ(t)	attach a task queue to thread t

loopOnQ(t)	begin executing procedures in t's queue
begin(t, p)	start executing the posted task p
end(t, p)	end executing the posted task p
enable(t, p)	enable posting of task p
post(t, p, t')	post task p asynchronously to thread t'

Table 1 List of operations

Among the operations, enable() is the important one which could capture the effect of ActivityManagerService (running in the system process) to help identify the ordering constraints for lifecycle callbacks made by the Android environment. With enable operations, the ordering between operations in the trace and UI callbacks can be captured, which is crucial for avoiding false positives.

b. Happens-before rule

DroidRacer provides a generalized happens-before rule which could be divided into a thread-local happens-before and inter-thread happens-before rule. Due to space constraint, the detail semantic rules can be found in the paper (Pallavi 2014).

3.2 CAFA

CAFA presents the first causality model based on happens-before rules for the event-driven Android system. CAFA implemented a race detector for finding races that lead to use-after-free violations using the causality model.

a. Language

Similar to DroidRacer, CAFA developed a language to analyse operations separately in a conventional thread-based concurrency tool (Table 2) and an event-based concurrency tool (Table 3):

begin(t); end(t)	begins and ends a task t
rd(t,x); wr(t,x)	reads and writes a value to variable x in task t
fork(t,u); join(t,u)	forks a new thread u from task t; waits in task t until thread u ends
wait(t,m); notify(t,m)	wait(t,m) stalls task t until notify(t,m), m is the monitor.

Table 2 List of operations for thread-based concurrency

send(t,e,delay)	enqueues a new event e at the end of the event queue in task t. e would be executed after delay millisec- onds has elapsed since it is enqueued
sendAtFront (t, e)	enqueues a new event e at the front of the event queue in task t. sendAtFront is used when the program- mer wants to prioritize an event over earlier events
register(t, l) and perform(t, l)	models the event listener pro- gramming construct in Android. An event listener can be per- formed as part of an event only after it has been registered with the runtime.

Table 3 List of operations for event-based concurrency

b. Causality model

CAFA presents the causality model based on happens-before rules for the event-driven Android system. The happens-before rules are divided into four sets:

- Conventional happens-before relations: *Program-order rule; Fork-join rule; Signal-and-wait rule*
- Event generation and execution: *Event listener rule; Send rule; External input rule*
- *Atomicity rule*
- *Event queue rule*

Due to space constraint, the detailed semantic rules can be found in the paper (Chun-Hung 2014).

b. False Positives elimination

CAFA uses the causality model to find use-after-free violations. A use-after-free violation arises when a pointer is dereferenced (used) after it no longer points to any object (freed). However, false alarms may happen when free is executed before the use and the programmer has taken care to either not execute the use or reallocate a new object before the use. Two simple heuristics are used to filter the false positives:

- if-guard check:
  - if-eqz (jump if a pointer is null)
  - if-nez (jump if a pointer is not null)
  - if-eq (jump if two pointers are equal)
- intra-event-allocation check:
  - if there is an allocation after a free in an event
  - if there is an allocation before a use within the same event

4. Implementation and Evaluation

To provide with hands-on experience about how to apply the techniques to real productions, DroidRacer and EventRacer for Android have been fully run and tested. The original implementation files are provided with the links in the reference.

4.2 DroidRacer Implementation

DroidRacer has three components:

- *UI Explorer*: scans the UI state of the test application during runtime, and automatically exercises UI events on the test application.
- *Trace Generator*: As the UI explorer drives the test

application; the Trace Generator logs operations performed by the application, which are relevant for race detection.

- *Race Detector*: Takes the trace generated by the race detector as input, constructs a happens-before graph and detects races

DroidRacer is a command line tool without GUI, it has been built by modifying and instrumenting Android 4.0.1_r1. It implements an offline race detection algorithm to detect data races. Below are the instructions on how to use the DroidRacer system.

a. Download and Build Android Source
Use a host machine (Ubuntu 12.04 is selected for my experiment) with the following specifications:

- A Linux or Mac system with at least 16 threads and at least 8GB RAM for faster builds (first build takes nearly 25 mins, and incremental builds between 2-6 minutes), otherwise may take hours based on resources provided.
- 64 bit system is required for Android version 2.3.x and newer.
- Python 2.6 -- 2.7; GNU Make 3.81 -- 3.82; JDK 7; Git 1.7 or newer.

Once the machine is ready, the next step is to initialize a build environment by installing required packages which are listed on Android official website. Then we are ready to download the source code by using Repo, which is a repository management tool that Google built on top of Git. Repo unifies the many Git repositories when necessary, does the uploads to the revision control system.

After performing all environment initializations, setting up the Repo repository and downloading the Android source code, the Android source code could then be instrumented and built to lunch the instrumented emulator, which could be used to log execution traces as we instructed.

b. Instrument the Android Source Code
The original ROM (Android version 4.0.1_r1 based on Android Open Source Project) is customized to instrument several key components in Android. The list of the instrumented files is attached in a note.

In order for the Trace Generator to log the operations corresponding to the core language defined for DroidRacer, Dalvik bytecode interpreter is instrumented to log read, write, acquire, release operations and track method invocations leading to the execution of application code. These DVM related files are located in the "vm" folder at the root of the Android source tree.

The core Java and Android framework libraries have been instrumented to collect execution traces for target applications and system services. For example, asynchrony related operations like attach(), 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. The corresponding files are located in "libcore" and "frameworks" folder.

c. Run DroidRacer
After successfully instrumented the system and lunched the emulator, in order to run DroidRacer, three other requirements need to be met:

- *AbcClientApp*: an Android application to be installed on the emulator. This app is needed to communicate with the ModelCheckingServer.
- *abc.txt*: a configuration file for the emulation. This file contains several lines indicating the specification for the tested application and the emulator, for e.g., process name of the app; fully qualified package name of the app; depth of UI events to be generated by UI explorer of droidRacer, etc. This file needs to be pushed into created or pushd (using adb tool) to "/mnt/sdcard/Download" on the running emulator.
- *ModelCheckingServer*: a project used to start testing apps using DroidRacer. This project needs to be imported into Eclipse. ModelCheckingServer acts as a server with which the emulator communicates after each testing run. The server performs initializations for each run as specified in abc.sh. Remember to replace the paths leading to android tools or DroidRacer files, with the path on your machine.

After all prerequisite projects have been deployed, install the app-to-test on the emulator and force stop the app. Add three parameters (app-process-name to be tested as input, port to communicate with emulator, emulator ID) to ABCServer.java and use ModelCheckingServer to start testing apps. The trace file is abc_log.txt, which could be pulled by adb tool during the execution.

4.2 EventRacer Implementation
EventRacer for Android is an online system for analyzing data races for Android system. Unlike DroidRacer, EventRacer is an interactive web system for automated testing. EventRacer has the following components:

- *Instrumented Android system image*: Android version 4.4 is instrumented to collect following information during the runtime.
- *Android UI/Application Exerciser Monkey:* To automatically explore the application, the Monkey provided by Android is used to run on the emulator and generate pseudo-random streams of user

events such as clicks, touches, or gestures, as well as a number of system-level events.

- *Happens-before graph construction*: After exploring the application, an execution trace is obtained consisting of dispatched events and operations that capture the essential features of event-driven Android applications. The happens-before edges are linked between individual operations and events in the execution trace with specific happens-before orderings.
- *Race detection:* Given the happens-before graph, the race detector applies an effective algorithm to go through all accesses of the same memory location and find all pairs such that, one of the accesses is a write and accesses were performed in an unordered events.
- *Race explorer:* All the reported data races are reported to the user in an interactive webpage.

To use EventRacer for race detection, user needs to upload the app-to-test to the website and the app will be uploaded to its testing server, where the app will then be executed on the instrumented emulator to get the execution traces. All the execution traces are analyzed offline to generate potential harmful races. The results are then displayed on the highly interactive webpage (**Figure 5** and **Figure 6**), where the users are able to navigate all the races and reveal their locations on a large-scale happens-before graph.

5. Future work

The project could be extended to:

- Integrate the idea of causality model and if-guard check and intra-event-allocation check into DroidRacer to help eliminate more false positives.
- Invent more efficient data structures and algorithm to lower the overhead of dat storage and computation
- Add some feasible manual annotations to improve the precision of the deteciton tool
- Try to convert the language and rules for Android to data race detection on iOS platform.

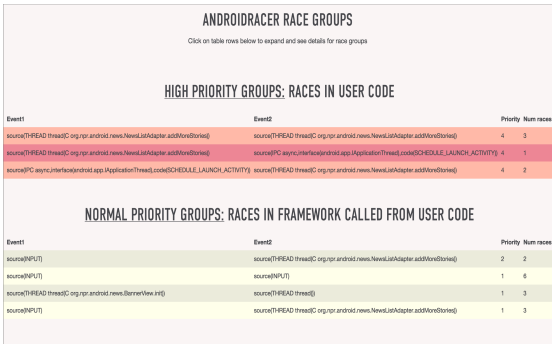

Figure 5 EventRacer race explorer

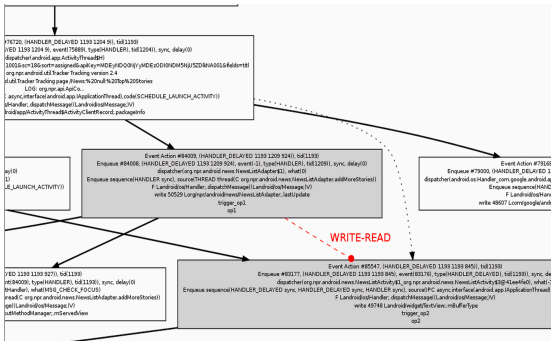

Figure 6 EventRacer happens-before graph

References

1. Maiya, Pallavi, Aditya Kanade, and Rupak Majumdar. "Race detection for Android applications." *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014.
2. Hsiao, Chun-Hung, et al. "Race detection for event-driven mobile applications."*Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014.
3. Raychev, Veselin, Martin Vechev, and Manu Sridharan. "Effective race detection for event-driven programs." *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. ACM, 2013.
4. s Petrov, Boris, et al. "Race detection for web applications." *ACM SIGPLAN Notices* 47.6 (2012): 251-262.
5. F-Droid. https://f-droid.org
6. Android open source project, https://source.android.com/
7. Android sdk, http://developer.android.com/sdk/index.html
8. DroidRacer implementation repository, https://bitbucket.org/hppulse/droidracer
9. DroidRacer related files repository, https://bitbucket.org/hppulse/droidracer-related-files
10. EventRacer for Android, eventracer.org/android/