Static Detection of Event-based Races in Android Apps

YONGJIAN HU, University of California, Riverside
IULIAN NEAMTIU, New Jersey Institute of Technology

Event-based races are the main source of concurrency errors in Android apps. All prior approaches to detecting event-based races have been dynamic. Due to their dynamic nature, these approaches suffer from coverage and false negative issues. We introduce a static approach and tool, named SIERRA, for detecting Android event-based races centered around a new concept of "concurrency action" (that reifies threads, events/messages, system and user actions) and statically-derived order (happens-before relation) between actions. Establishing action order is complicated in Android, and event-based systems in general, because of externally-orchestrated control flow, use of callbacks, asynchronous tasks, and ad-hoc synchronization. We introduce several novel approaches that enable us to infer order relations statically: auto-generated code models which impose order among lifecycle and GUI events; a novel context abstraction for event-driven programs named *action-sensitivity*; and finally, on-demand path sensitivity via backward symbolic execution to further rule out false positives. We have evaluated SIERRA on 194 Android apps. Of these, we chose 20 apps for manual analysis and comparison with the state-of-the-art dynamic race detector. Experimental results show that SIERRA is effective and efficient, typically taking 960 seconds to analyze an app and revealing 43 potential races. Compared with the dynamic race detector, SIERRA discovered an average 29.5 true races with 3.5 false positives, where the dynamic detector only discovered 4 races (hence missing 25.5 races per app) – this demonstrates the advantage of a sound static approach. We believe that our approach opens the way for precise analysis and static event race detection in other event-driven systems beyond Android.

ACM Reference format:
Yongjian Hu and Iulian Neamtiu. 2016. Static Detection of Event-based Races in Android Apps. 1, 1, Article 1 (January 2016), 18 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Android is the dominant software platform for smartphones and tablets [1]. Ever since the platform's inception, however, Android has been plagued by concurrency errors, with concurrency being one of the top-5 most common bug causes every year starting in 2008 [27].

Android apps use a concurrent event-driven model and generally revolve around a GUI. To keep the GUI responsive, only the main (UI) thread has access to GUI objects. Other, non-main threads, are used for long-running computation or I/O tasks, e.g., file download; when the task is finished, it sends a message to the main thread to perform a GUI update. Event handlers (callbacks) are written by the developers while a system component named Android Framework (AF) orchestrates control flow by invoking these event handlers in response to GUI actions or hardware notifications. This event-based model can lead to concurrency errors because the order in which events are posted is nondeterministic, e.g., an app with two asynchronous tasks T_1 and T_2 where the developer assumes that T_1 always executes first, and T_2 relies on some initialization performed by

T_1. However, if T_2 executes first, the result can be a crash or error due to uninitialized data—this is called an *event-based race.*

Such errors are pervasive and pernicious: a study of 18,000 fixed bugs in the Android platform and apps has revealed that 66% of the high-severity bugs are due to concurrency [28]. Android concurrency research has shown that the majority of Android race bugs are event-driven races [7, 16, 18]; per Maiya et al. [18], in Android apps, event-driven races are 4–7 times more frequent than data races.

Hence there is a strong impetus for constructing tools that help find event-driven races in Android apps. To find such races, several *dynamic* detectors have been proposed, e.g., DroidRacer [18], CAFA [16], and EventRacer Android [7]. However, dynamic detectors have two main issues. First, due to their dynamic approach, they are prone to false negatives, i.e., miss actual bugs (in our experiments, EventRacer missed 25.5 out of 29.5 true races on average). Second, their effectiveness hinges on high-quality inputs that can ensure good coverage [6], as well as efficient ways to explore schedules.

To address these issues, we propose a static approach to event race detection. Android's concurrency model makes static event-based race detection challenging – it is difficult to establish happens-before relations – for several reasons. First, unlike traditional (desktop/server) Java applications, Android apps do not have a `main` method but rather rely on callbacks being invoked by the AF. Second, apps consist of activities (separate screens) that can be navigated back and forth [6]; further, each activity comprises GUI objects which can be accessed in relatively unconstrained order [23]. Third, asynchronous/long-running operations (e.g., network and I/O) are run in separate threads and their results posted back via messages, in nondeterministic order. Fourth, ad-hoc synchronization eludes standard control- and data-flow analyses.

To overcome these challenges, we introduce several novel approaches. First, we reify Android concurrency primitives and their processing as context-sensitive *actions* (event processors) that can model threads, messages, lifecycle activities and GUI events. Second, we use static analysis refinements to significantly improve precision, e.g., automatically-constructed harnesses to kickstart the static analysis, and a novel *action-sensitive* context abstraction for pointer analysis (Section 3). Third, we introduce Happens-before rules which order actions, from a harness-based model for lifecycle and GUI events to inter- and intra-procedural domination; the result is a Static Happens-before Graph (Section 4). Fourth, for those actions and memory accesses that have not been orderable yet, we use symbolic analysis, i.e., goal-directed (refutation-based) symbolic execution, to see if indeed independent path conditions allow events to execute in any order (Section 5).

We have implemented our approach in a tool named SIERRA (StatIc Event-based Race detectoR for Android). Given an app, SIERRA analyzes the bytecode (hence the app source code is not required, and apps can be readily analyzed in the APK format there are distributed in) and produces a ranked list of potential races.

Section 6 presents the experimental results. We evaluated SIERRA on 194 apps, of which 20 were chosen for further manual analysis. Experiments show that SIERRA is effective, discovering about 1,223 happens-before edges and 68 racy pairs per app. Refutation reduces these substantially, to just 43 race reports per app. SIERRA is efficient: it typically takes 960 seconds to analyze an app, which is acceptable for a static analysis. For the 20 manually-analyzed apps, we ran EventRacer Android [7], the most advanced dynamic race detector to date. We found that SIERRA reports 38 potential races on average, of which 29.5 are true races, whereas EventRacer Android reports 4 races, missing 25.5 true races. Moreover, SIERRA can also filter out some false positives reported by EventRacer.

In summary, our main contributions are:

(1) A definition of actions as Android concurrency units.
(2) An approach for defining static happens-before relationships in Android apps.
(3) A suite of refinements and precision enablers, based on static and symbolic analysis, that substantially increase the precision of ordering.

```
class NewsActivity extends Activity
    implements onClickListener {
  RecycleView rv;
  NewsAdapter adapter;

  void onCreate() {
    rv = (RecycleView) findViewById
          (...) ;
    adapter = new NewsAdapter(...);
    rvNews.setAdapter(adapter);
  }

  void onClick(View v) {
    new LoaderTask(adapter).execute() ;
  }
}
```

```
class LoaderTask extends AsyncTask {
  final NewsAdapter adapter;
  LoaderTask(NewsAdapter adapter) {
    this.adapter = adapter;
  }

  void doInBackground() {
    News[] newslist = download();
    adapter.add(newslist);
  }

  void onPostExecute(News news) {
    adapter.notifyDataSetChanged();
  }
}
```

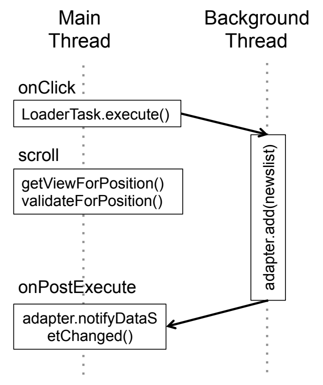

Fig. 1. Intra-component race.

(4) A tool, SIERRA, which implements our approach and works on off-the-shelf Android apps without requiring app source code.
(5) An evaluation of SIERRA on 194 Android apps.

2 BACKGROUND AND MOTIVATION

We provide a brief background of the Android platform and the app construction model, then motivate our approach with two concrete examples of races.

2.1 Android Background

Android platform. The Android software stack consists of apps running on top of the AF, which orchestrates app control flow and mediates intra-app and inter-app communication, as well as the communication between apps and hardware. Apps are typically written in Java (though certain parts can be written in C or C++ for efficiency) and compiled into either Dalvik bytecode that executes on top a Dalvik virtual machine (Android version < 5.0) or directly to native code (Android version ≥ 5.0). The Dalvik VM or native code run on top of an Android-specific Linux kernel.

Android app construction. An app consists of *components*; there are four kinds of components: (1) Services, used for background operations, (2) Content Providers, which manage access to data, (3) Activities, i.e., user visible screens, and (4) Broadcast receivers, used for system or application events [4].

Activities are the most popular components—apps usually consist of a suite of Activities. The app transitions among activities in response to user input, e.g., in the Amazon app, the "Home" screen is named MainActivity; when the user clicks on the "Search" box, the app transitions to a SearchActivity; upon selecting from the list of result items, the app transitions to the SearchListActivity . Within one activity, various GUI objects are placed in a View hierarchy. Activities follow a state machine, where the states have associated callbacks that can be filled out by the programmer, e.g., upon activity creation, the onCreate() method is called, upon destruction the onDestroy() callback is invoked, while in-between the activity can cycle between visible and invisible states that have associated onStop()/onRestart () callbacks. GUI objects, e.g., menus, buttons, have callbacks as well. The AF automatically invokes callbacks in response to user input (e.g., click 'Back') or hardware events.

While components are strongly isolated — e.g., the only way for one Activity to share information with another activity is through message passing (called Intent s) — inter-component races are possible (Section 2.3).

Threads. Android has three kinds of threads: looper, background, and binder. Looper threads have an associated Looper object that implements message processing: the thread blocks waiting for messages and when a message comes, it is processed atomically; the importance of this *looper atomicity guarantee* will become clear later on (Section 4.3 §6). Each Android app has a "main" thread, also known as the UI thread, responsible for updating the GUI (GUI objects are only accessible to this thread); the main thread is a looper thread. Background threads are akin to traditional threads, created via fork (). Binder threads are used in thread pools to process inter-process communication. Apps typically perform actual work in background threads, which notify the main thread when a GUI update is needed, by posting a message to the main thread's processing queue.

2.2 Intra-component Race

Figure 1 shows an actual, harmful, event-based race in the Android platform (AOSP)[1] – more precisely, an *intra-component race*, as it happens within one activity. The NewsActivity, shown on the left, has a RecycleView to display the news items. RecycleView is an advanced widget, designed to display large data sets that can be scrolled very efficiently by maintaining a limited number of views. In NewsActivity's onCreate method, the RecycleView is initialized and the corresponding adapter is set (lines 7–9). The activity registers an onClickListener that creates a LoaderTask (a subclass of AsyncTask) to update the news list; this is shown in the center of the figure. The time-consuming download operation is in the doInBackground method which runs in a separate thread. This practice is strongly suggested in Android to make the app more responsive. When the AsyncTask is done, it posts an onPostExecute callback to the main thread and notifies the adapter to refresh the RecycleView with the latest data.

The race manifests when the user scrolls the view before downloading has finished — a runtime exception will then crash the app. This exception occurs only in the specific event schedule (as shown in Figure 1 on the right) where the onScroll callback is executed before onPostExecute on the main thread, and the adapter's internal data is just updated in the background thread. The root cause of the bug is that when the user scrolls down, the RecycleView will decide which view to show according to the last-scrolled position. If the view position does not match the previously-cached result because the adapter has not had a chance to execute notifyDataSetChanged to update the cache, the exception is thrown. The fix for this bug is to invoke notifyDataSetChanged right after the adapter's add method, or move the add method to the onPostExecute callback in AsyncTask. Note that this race bug is very hard to reproduce using dynamic analysis as it manifests only in specific schedules.

2.3 Inter-component Race

The previous example has shown a harmful race within one Android component (Activity). In Figure 2, we show an *inter-component* "Activity vs Broadcast Receiver" race that occurs across Android components. In the onCreate callback of the MainActivity, a DataBase object is created and a Broadcast Receiver is registered. Accordingly, the receiver is unregistered and the DataBase object is freed in the onDestroy callback where the activity is no longer usable. The program opens the database in the onStart method when the activity is becoming visible to the user, and closes it in onStop when it is no longer visible — the rationale is, the app should consume fewer resources when the activity is pushed into the background. The Broadcast Receiver is designed to be invoked from the background service when new data is available and communicate with the foreground activity to update the data.

The event-based race occurs if the broadcast message is delivered at the time when the activity is pushed into the background. Since the database is closed in the onStop callback, an update operation at this time in the onReceive callback would cause exceptions. There are multiple solutions to fix this event race bug. For example,

[1]https://code.google.com/p/android/issues/detail?id=77846

```
class MainActivity {
  DataBase mDB;
  BroadcastReceiver recv = new
       BroadcastReceiver() {
    void onReceive(Context ctx,
         Intent i) {
      Bundle b = i.getExtras();
      mDB.update(b);
    }
  }

  void onCreate(...) {
    mDB = new DataBase();
    registerReceiver(recv, ...);
  }
  void onStart() {
    mDB.open();
  }

  void onStop() {
    mDB.close();
  }

  void onDestroy() {
    unregisterReceiver(
         recv);
    mDB = null;
  }
}
```

Fig. 2. Inter-component race.

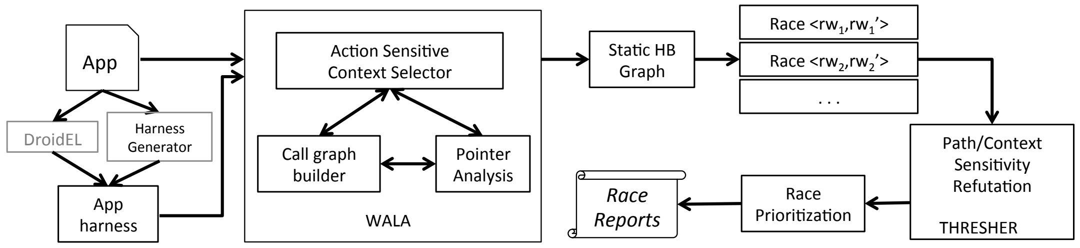

Fig. 3. Overview of SIERRA.

registering and unregistering the broadcast receiver during onStart and onStop, or adding a flag to indicate the status of the activity and checking it before database updates. Again, this race requires a specific event ordering and is likely to be missed by dynamic analysis if that specific schedule order is not exercised.

3 APPROACH

In this section, we first describe SIERRA's architecture (Section 3.1) and harness generation (Section 3.2). Section 3.3 describes action sensitivity, a novel approach that enables precise static analysis for event-driven programs.

3.1 Architecture

Figure 3 shows the architecture of SIERRA. First, we leverage DroidEL [10], a static AF modeling tool, to handle view inflation and reflection. The AF relies on reflection to load the APK. For example, the GUI layouts, written in XML, are accessed via the findViewById(**int** id) API to access the specific view. However, static analysis cannot resolve such objects created via reflection. DroidEL can resolve these objects and creates bindings between layout structure and view objects. The models generated by DroidEL are then integrated into our harness generator (described later) that will drive the analysis.

Second, we leverage WALA [17] to perform whole-program (application and framework) analysis. WALA is a mature, industrial-level program analysis tool for object-oriented languages like Java. It provides versatile features for program analysis such as pointer analysis, call graph generation and control/data flow analysis.

Action	Creation (SHBG node)	Happens-before introduction (SHBG edge)
Thread		
Asynchronous thread	**new** AsyncTask	Thread. start ()
Background thread	**new** Thread	AsyncTask.execute ()
Runnable	**new** <...> **implements** Runnable	Executor . execute ()
Message	Message.obtain ()	sendMessage*(Message msg)/post *(Runnable r)
		Execution: Runnable.run()
Lifecycle event	onCreate(), onDestroy()	According to the activity lifecycle, e.g.,
	onStart () , onStop(), onRestart ()	onCreate→ <created,onStart>
	onPause(), onResume()	onStop→ <stopped,onStart>
GUI event	onClick *()	According to the GUI model, e.g., onClick1→ onClick2
System event	BroadcastReceiver . onReceive()	registerReceiver
	onServiceConnected	bindService
	onServiceDisconnected	startService

Table 1. Actions and HB introduction.

Selecting the appropriate context in pointer analysis is key to achieving scalability and precision. Prior research has shown that a mix of object sensitivity and call-site sensitivity is an effective abstraction for object-oriented languages. However, in event-driven systems like Android, neither object sensitivity nor call-site sensitivity is precise enough due to over-approximation (merging) when the context length exceeds the threshold k. SIERRA introduces a novel abstraction called *action sensitivity* which adds action as part of the context abstraction, and combines object sensitivity and call-site sensitivity within an action (Section 3.3).

Different action execution orders on the looper thread lead to a non-deterministic schedule; an event-based race can manifest if two actions access the same memory, and at least one access is a write. However, naively considering that each pair of memory actions is a potential race will produce an overwhelming number of false positives. SIERRA defines a set of static happens-before rules between actions to rule out infeasible racy action pairs, e.g., onCreate always happens-before onDestroy (only actions that do not have strict happens-before relation could be involved in races). This stage, described in Section 4.3, yields a Static Happens-before Graph (SHBG).

Next, SIERRA generates candidate races by intersecting the points-to sets between actions that are not ordered by happens-before. However, these pairs (named racy pairs) are not necessarily races since in asynchronous programming ad-hoc synchronizations are widespread. So, in the next step, we attempt to refute (rule out) false positives by a path-sensitive, backward symbolic execution; for this we extended the Thresher tool [8] to verify path feasibility between two actions (Section 5).

Race prioritization. Finally, to help developers fix likely-harmful races, SIERRA prioritizes race reports using several heuristics: 1) races in application code have higher priority than those in framework code; 2) framework races directly invoked from app code have higher priority than those invoked from the library; 3) races involved in pointer reference reads/writes are more likely to be dangerous as they can result in NullPointerException.

3.2 Harness Creation

We now describe SIERRA's automatic harness creation approach. As SIERRA performs whole-program analysis, we need to find the app's entrypoints. While in traditional Java programs we would start at main(), Android apps have no main. Rather, in Android, app control flow is orchestrated by the AF, which invokes lifecycle callbacks, such as onCreate when the app is created, or onDestroy when the app is destroyed. Besides these lifecycle callbacks,

```
1  class Harness {
2    public static void main() {
3      NewsActivity a = new NewsActivity();
4      a.onCreate();
5      a.onStart();
6      a.onResume();
7      while (*) {
8        switch(*) {
9        case 1: a.invokeOnClick(); break;
10       case 2: a.invokeOnScroll(); break;
11         ......
12       }
13     }
14     a.onPause();
15     a.onStop();
16     a.onDestroy();
17 } }
```

Fig. 4. Harness example.

an app can implement view event handlers (e.g., onClick and onScroll) that can be registered either statically in the layout XML or dynamically in code. Figure 4 illustrates a harness generated for the example in Figure 1.

First, we create a Harness activity with a main method which serves as the entrypoint. Second, we instantiate the NewsActivity and invoke its Activity lifecycle callbacks (lines 4–6 and 14–16). Third, starting from these lifecycle callbacks, a call graph is built by WALA to compute the reachable methods. Within the reachable methods, the analysis might discover new callbacks. For example, an onClickListener may be created and registered via setOnClickListener. At this time, the harness generator creates synthetic invocation sites (lines 9–11) and builds the call graph again. This process iterates until a fix-point is reached, i.e., no new callbacks found. Finally, the callbacks registered in XML files are added to the harness since they are unique. We borrow FlowDroid [5]'s predefined callback list to find callbacks.

3.3 Action Sensitivity

Context sensitivity plays a key role for scalability and precision in static analysis. Two main kinds of context sensitivity have been proposed for object-oriented languages: *object-sensitivity* (k-obj) [19] and *call-site-sensitivity* (k-cfa) [24].

Prior research [20, 25] has shown that object-sensitivity increases precision; however, we have found that it is still not precise enough for our Android setting. *K-obj* sensitivity merges the last k object allocation sites, thus precision is lost for contexts longer than k. The same loss occurs for *k-cfa* sensitivity which merges the last k call sites. Incorrect aliasing may occur when two different actions call a method foo() which contains j call sites to method bar() and allocates an object. If $j > k$, both k-obj and k-cfa fail to distinguish that the objects are allocated in two different actions and incorrectly consider them as aliased because their last k allocation sites (or k call sites, respectively) are the same. While precision could be improved by increasing the value of k, this greatly decreases performance, as analysis complexity is exponential in k.

Based on the insight that objects should be associated with their corresponding actions, SIERRA introduces a new context abstraction named *action-sensitivity* which greatly improves precision. When building the call graph for an action, we add the action's id as part of its context, and leverage *hybrid-context-sensitivity* which consists of *object-sensitivity* and *call-site-sensitivity*. More specifically, the *hybrid-context-sensitivity* uses k-obj for normal dispatch calls and k-cfa for static invocations within one action. Each object's abstract context has the action id where this object is allocated in. In the previous foo() example, the objects created by foo() have different action ids in their contexts, and are not aliased (conflated) anymore. Note that, although within one action the objects may still lose precision due to last k merges, across actions objects are still separate. Since SIERRA focuses on analyzing objects accessed by different actions, we find that action-sensitivity is particular useful for our race detection. While action-sensitivity is effective at distinguishing abstract objects, a class of objects named "views" need to be handled specially, as explained next.

Inflated view context. Apps can define views using layout XML files, and then inflate the views at runtime. Android provides the findViewById(**int** id) API to access the inflated view, given the constant view id; findViewById can be invoked in different actions, but the object is aliased when using the same id. SIERRA uses a special context named *InflatedViewContext* that contains view ids and their type. During APK parsing, for each view defined in the layout, SIERRA saves its view id into a map. When findViewById(id) is called, SIERRA uses this constant id to retrieve the view object from the map; two inflated view objects are considered aliased when they have the same ids.

4 HAPPENS-BEFORE RELATIONSHIP

Prior event-driven race detectors for Android have defined *dynamic* happens-before rules [7, 16, 18]. Those definitions do not easily translate here, as our approach is static and uses symbolic path condition information, hence we define our own static happens-before (HB) rules. HB orders *actions*, described shortly, and the order relations are captured in a *Static Happens-Before Graph* (SHBG).

4.1 Definitions

We first define the concepts and notations used in our approach. We use A, B, A_1, etc. as action names. The happens-before relation, denoted $A \prec B$, indicates that we can statically prove that action A is completed before action B starts.

Races. We define races as unordered memory accesses, at least one of which is a write. Our points-to sets map variables x to memory locations ρ, i.e., $\pi(x) = \rho$. Memory accesses α are $\langle x, \tau, A \rangle$ bundles, indicating that variable x is accessed using access type τ (*read* or *write*) in action A.

Racy pairs. We define *racy pairs* as follows: accesses α_1 and α_2 form a racy pair if they come from different actions A_1 and A_2, operate on at least one shared location (i.e., their points-to sets' intersection is non-empty, $\pi(\alpha_1.x) \cap \pi(\alpha_1.x) \neq \emptyset$) and at least one of the accesses $\alpha_1.\tau$ or $\alpha_2.\tau$ is a write.

Race-finding strategy. Our approach proceeds by constructing an HB graph, then finding all candidate racy pairs, and finally using symbolic analysis to refute those racy pairs that are actually ordered.

4.2 Actions: SHBG Nodes

Actions are the building blocks of our approach. An action represents *context-sensitive event handling*. Table 1 shows how HB nodes and edges are identified, so they can be added to the SHBG. When the analysis reaches an action creation (column 2) it creates the appropriate HB node, as described next.

There are four classes of actions. *Threads* can be created as asynchronous tasks, background threads, or runnables. *Messages:* in Android, messages are sent using either the send* or post* API; in either case, the message has an associated Runnable which will execute in the recipient thread. *Lifecycle events:* Android activities are controlled by the Android Framework and have a well-defined lifecycle, described as activity states, which form HB nodes, while activity state transitions form HB edges (Section 4.3 §2). *GUI events:* our harness is a GUI model where GUI callbacks are HB nodes, while the GUI callback order introduces HB edges (Section 4.3 §3).

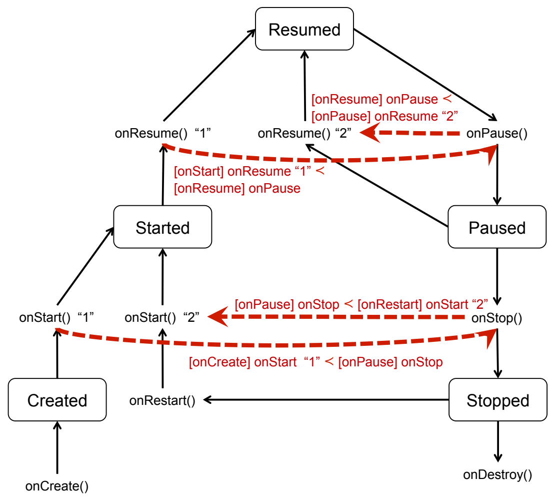

Fig. 5. HB edges among Activity lifecycle callbacks (dashed red arrows) induced by CFG dominance in our harness model. Black edges represent control flow.

4.3 HB Rules: SHBG Edges

We now define the HB rules, i.e., rules for adding edges between actions in the SHBG.

1. Action invocation rule: when an action is invoked, the *sender* action happens before the *recipient*. For example, as is standard in race detection, we add an HB edge between the action in which a thread is created and the new action (that thread's body). Similarly, we add an HB edge from the message sender's action to the message's Runnable.

2. Android component lifecycle rule: in Android, activities follow a lifecycle described as a state machine where state transitions invoke callbacks [3]. The Android Framework will invoke these callbacks in predefined order, e.g., upon activity creation, onCreate is invoked first, then onStart, then onResume.[2] Our key insight is to use (pre) dominator information to distinguish between different instances of callbacks that appear in cycles so we can order them.

We illustrate this rule in Figure 5 on the actual Android activity lifecycle. According to the lifecycle rules, onCreate is the first method to be invoked after an Activity has been created, while onDestroy is the last method to be invoked before the Activity is disposed of. However, in the call graph, all these callbacks are disconnected. The harness, described in Section 3.2, mirrors the Activity lifecycle and invokes the callbacks in the required order. In the harness, as onCreate dominates any other node, we know that any shared memory access in onCreate will precede accesses in subsequent actions, e.g., onStart, hence we can add an HB edge onCreate→ onStart.

We now show how we deal with cycles. As seen in Figure 5, onResume/onPause and onStart/onStop form cycles. At first sight, these callbacks do not appear orderable by HB. For example, since onResume is invoked after either Started or Resumed states, onResume appears not to be orderable with onPause — onPause can come either before or after onResume.

Our insight is that we can distinguish between the two onResume's if we take into account dominator information. For simplicity let us name onResume 1'' the callback pre-dominated by onStart and onResume 2'' the callback pre-dominated by onPause. Now it can be easily seen that

[2]While this lifecycle state machine has been unchanged since Android's inception, it would be trivial to change our model to accommodate potential future changes in the state machine, should they occur in subsequent Android versions.

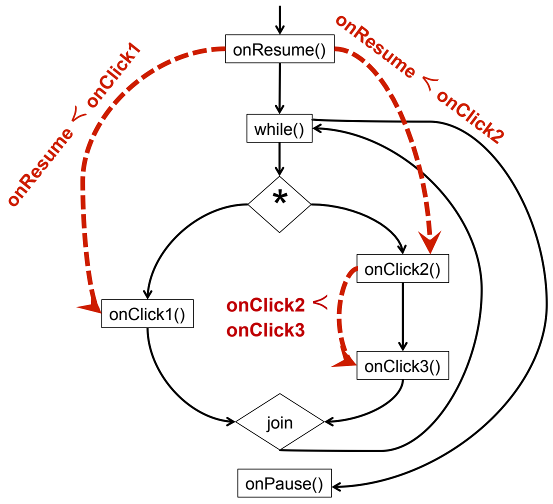

Fig. 6. HB edges (dashed red arrows) induced by CFG dominance in our GUI model, e.g., onResume→ onClick1, or onClick2→ onClick3. Black edges represent control flow.

$$\text{onResume } 1'' \prec \text{onPause},$$

and

$$\text{onPause} \prec \text{onResume } 2'',$$

hence the previously-unorderable callbacks can actually be ordered. Similarly, we have:

$$\text{onStart } 1'' \prec \text{onStop}$$

$$\text{onStop} \prec \text{onStart } 2''.$$

3. GUI layout/object order: similar to the Android lifecycle, the GUI layout captured by the harness (Section 3.2) is used as a basis for HB. We illustrate this rule in Figure 6 on a simplified GUI layout, where an app cycles and nondeterministically chooses between onClick1() or onClick2 (); onClick3(). Since onResume pre-dominates onClick1 we have:

$$\text{onResume} \prec \text{onClick1}$$

$$\text{onResume} \prec \text{onClick2}$$

$$\text{onClick2} \prec \text{onClick3}.$$

4. Intra-procedural domination. Assume that a method M in activity A has two outgoing calls e_1 and e_2 that post actions A_1 and A_2, respectively. If e_1 dominates e_2 then $A_1 \prec A_2$; this is intuitive because e_1 will always be invoked before e_2 and by the time e_2 executes (and gets a chance to post A_2), A_1 has already been posted, so A_2 can only be posted after A_1.

5. Inter-procedural, intra-action domination. This is similar to rule 4, but the difference is that e_1 and e_2 can be in two separate methods M_1 and M_2 of the same activity A. Note that e_1 cannot straight-up dominate e_2 because e_2 might be invoked from a context that does not involve e_1. We leverage WALA's interprocedural CFG (ICFG) to address this issue as follows: we remove e_1 from the ICFG and check whether e_2 is still reachable; if it is not reachable, then de facto e_1 dominates e_2 and we add $A_1 \prec A_2$. If, on the other hand, e_2 is still reachable when e_1 is absent, we do not add any HB edges.

6. Inter-action transitivity: If $A_1 \prec A_2$, A_1 posts an action A_3, and A_2 posts an action A_4, then $A_3 \prec A_4$. Figure 7 illustrates this. On top (Figure 7 (a)) we show the order relation. On the bottom we show the two possible execution schedules for this order. A_1 executes first, and during its execution, it posts A_3. Importantly, by the time A_1 finishes, A_3 is already posted. We have two cases: Figure 7(b) when A_3 executes before A_2 does, hence

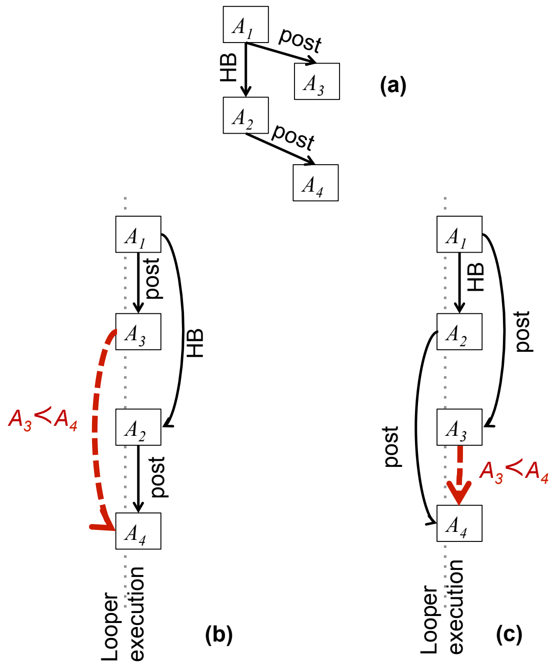

Fig. 7. Adding intra-action transitive HB edges: (a) is the action order, while (b) and (c) are possible schedules.

$A_3 \prec A_4$ because A_4 has not even been posted when A_2 finishes; and Figure 7(c) when A_2 executes first, but because A_3 has already been posted when A_2 starts executing, A_4 can only be posted after A_3 hence $A_3 \prec A_4$. We can infer these orderings thanks to the looper atomicity guarantee.

7. Transitivity: HB is transitive, i.e.,

$$A_1 \prec A_2 \land A_2 \prec A_3 \implies A_1 \prec A_3$$

We repeatedly invoke transitive closure together with rule 6, as rule 6 can discover new HB edges in ways other than control- or data- flow (which rules 1–5 are limited to).

Note that after applying these HB rules we still have an *under-approximation* of all HB relations, which preserves soundness at the expense of having potential false positives. We now describe how we further introduce ordering to refine our HB relations hence reduces false positives.

4.4 Accesses and Races

Handlers and threads. A thread can register a Looper object to receive asynchronous messages. Each Looper object is associated with one thread and each thread can register at most one Looper. In Handler's constructor, a Looper object must be specified so that the messages sent via this Handler will be delivered to the corresponding thread. Two actions are considered to be potentially racy, iff the corresponding Handler objects refer to the same Looper. SIERRA pre-processes all the creation sites of Loopers and Handlers to learn which thread is associated with the Handler by traversing the call graph from the entry of each thread and performing an in-thread reachability analysis.

Ruling out ordered accesses. Racy pairs (e.g., accesses α_A and α_B in actions A and B, respectively) form the starting point for detecting races—these accesses are candidate races unless we can refute that assumption, i.e., prove that α_A and α_B are ordered (we do so via symbolic execution, described next).

```
 1 Timer.Runnable runner = {
 2 void run() { // action A
 3  if (mIsRunning) {
 4   mAccumTime=... // αA
 5    if (*) {
 6     ...
 7     postDelayed(runner ,...) ;
 8    }
 9   else
10    mIsRunning=false; }
11 }}

12 void stop() { // action B
13  if (mIsRunning) {
14    mIsRunning = false;
15    mAccumTime=... // αB
16  }
17 }
```

Fig. 8. Refutation helps eliminate this false positive in the OpenSudoku app.

5 SYMBOLIC EXECUTION-BASED REFUTATION

A candidate race, e.g., accesses α_A and α_B in two unordered actions A and B, is not necessarily a true positive since accesses could be protected by ad-hoc synchronization [22]; such synchronization idioms are prevalent in event-based systems to protect the event handler from executing unsafe paths.

Example. We show how SIERRA refutes a candidate race in the OpenSudoku app (Figure 8). The run method on the left is from a Runnable object that is posted from the onResume callback. The stop method on the right is invoked from the onPause callback to stop the Runnable object.

These two actions do not have an HB edge and they both write to a shared field mAccumTime (lines 4 and 15). SIERRA starts by considering both orderings possible. Let us assume that action B occurs before action A. SIERRA performs backward symbolic analysis starting from α_A (line 4 in action A). When the analysis reaches the **if** conditional on line 3, it adds a path constraint {mIsRunning = **true**}, i.e., a precondition to reach α_A. The backward analysis continues until reaching the boundary of the run method and proceeds (assuming there are no conflicting constraints). Then SIERRA traces the path back to the exit block of the stop method in action B, and continues backward. When the path reaches line 12, SIERRA chooses to enter the block guarded by line 13 because the guard condition is consistent with the path constraint {mIsRunning = **true**}. Finally, a conflicting constraint is found when the path reaches line 14 which performs a strong update to mIsRunning. This strong update means the path constraint after this statement must be mIsRunning = **false**, which conflicts with our current path constraints. After searching all the possible paths, SIERRA cannot find a feasible way to witness the backward path from α_A to α_B, thus the candidate race is refuted.

The backward analysis framework is based on Thresher [8], which we adapted to fit our event-based race detection scenario. Thresher is designed to perform precise heap refutation by traversing all the paths related to the candidate query back to the program's entrypoint. SIERRA changes the refutation process to be witnessing a feasible path between a source and a sink. The candidate race is a true positive, *iff* in both orderings of actions A and B, there does exist a feasible path from α_A to α_B, and vice versa.

On-demand constant propagation. When the action is Handler.handleMessage(Message m), program behavior depends on the values of Message's fields, e.g., the what field is an integer indicating the type of the message. To increase precision, we introduce constraints to check if any of these fields are constant integers and used as guard conditions. SIERRA does on-demand constant propagation from the creation site of the action (i.e., handler.sendMessage) and checks if any of the message's fields are constant. If yes, the constraints are added to the query of the backward symbolic executor.

App	**Installs**	**Bytecode size** (KB)
APV	500,000–1,000,000	736
Astrid	100,000–500,000	5,400
Barcode Scanner	100,000,000–500,000,000	808
Beem	50,000–100,000	1,700
ConnectBot	1,000,000–5,000,000	700
FBReader	10,000,000–50,000,000	1,013
K-9 Mail	5,000,000–10,000,000	2,800
KeePassDroid	1,000,000–5,000,000	489
Mileage	500,000–1,000,000	641
MyTracks	500,000–1,000,000	5,300
NPR News	1,000,000–5,000,000	1,500
NotePad	10,000,000–50,000,000	228
OpenManager	N/A	77
OpenSudoku	1,000,000–5,000,000	170
SipDroid	1,000,000–5,000,000	539
SuperGenPass	10,000–50,000	137
TippyTipper	100,000–500,000	79
VLC	100,000,000–500,000,000	1,100
VuDroid	100,000–500,000	63
XBMC remote	100,000–500,000	1,100

Table 2. App popularity and size for the 20-app dataset.

Caching. Refutation's running time varies depending on app complexity. A refutation could be terminated by the executor if the system runs out of memory or exceeds the maximum number of paths (we set this to 5,000 paths in SIERRA). In either case, SIERRA soundly reports the race, though it might be a false positive. To prevent redundant computation, SIERRA memoizes (caches) the call graph nodes visited in a refuted query. Later queries first check the cache. If the current node in a path exists in the cache, then the query stops immediately as the path is infeasible. This caching mechanism is particularly useful where many race candidates are within the same call graph node or dominated by that node in a refuted query.

6 EVALUATION

We have evaluated SIERRA in terms of *effectiveness*, i.e., how many potential races it can find, and *efficiency*, i.e., how long it takes to analyze an app.

App datasets. We chose apps from various categories (news apps, video players, email clients, etc.) and of various sizes. First, we reuse Gator [23]'s benchmark which contains 20 apps as listed in Table 2. We chose this dataset because all the apps are open-source so that we can manually check SIERRA's correctness. The center column of Table 2 shows app popularity, retrieved from Google Play in August 2017. As we can see, 17 of these apps have in excess of 100,000 installs; the number of installs was not available for Open Manager as it was retrieved from the alternative F-Droid market. The right column of Table 2 shows the bytecode size (`.dex`) for each app; this ranges between 63 KB and 5.4 MB. The experimental results on these 20 apps are discussed in Sections 6.1– 6.5.

Next, we chose an additional 174 apps with a median size of 1.1MB from F-droid [2] (an online open source repository for Android) for automatic testing. These results are discussed in Section 6.6.

App	Harne-sses	Actions	HB Edges	Ordered (%)	Racy Pairs w/o AS	Racy Pairs with AS	After refutation	Manual Inspection		EventRacer
								True Races	FP	App Races
APV	4	84	1,648	47	75	25	10	8	2	3
Astrid	6	147	2,755	26	319	83	54	37	17	-
Barcode Scanner	9	136	2,756	30	64	24	15	11	4	7
Beem	12	169	3,724	26	467	73	13	10	0	0
ConnectBot	11	171	4,829	33	567	96	58	43	15	16
FBReader	27	259	4,710	14	836	285	106	93	13	5
K-9 Mail	29	312	5,725	12	1,347	370	89	72	17	1
KeePassDroid	15	216	4,076	18	266	61	27	16	1	0
Mileage	50	331	8,498	16	496	195	36	33	3	1
MyTracks	8	198	6,826	35	634	174	80	75	5	34
NPR News	13	490	10,673	9	607	132	21	21	0	3
NotePad	9	72	609	24	436	65	31	27	4	9
OpenManager	6	92	1,036	25	532	113	55	51	4	5
OpenSudoku	10	141	1,425	14	426	158	110	83	27	72
SipDroid	11	206	2,386	11	321	94	27	17	10	-
SuperGenPass	2	43	343	38	82	16	6	6	0	3
TippyTipper	5	100	1,864	38	93	21	9	7	2	1
VLC	13	151	2,349	20	202	78	35	32	3	0
VuDroid	3	45	150	15	62	27	10	10	0	5
XBMC	13	330	4,218	8	445	137	63	48	15	17
Median	*10.5*	*160*	*2,755*	*22*	*431*	*80.5*	*33*	*29.5*	*8.5*	*4*

Table 3. SIERRA effectiveness on the 20-app dataset.

Experimental setup. We ran our experiments on an 8-core hyper-threaded (hence 16 threads) Intel Xeon E5-2687W CPU 3.4GHz, with 64GB memory. The server was running Ubuntu 14.04.1 LTS. We use DroidEL as a pre-processor to handle reflection and extract app layout, and automatically create harnesses via our harness generator. WALA has provided points-to information and call graph construction. The action-sensitive context selector is implemented as a WALA plugin. SIERRA modifies Thresher to run goal-directed path-sensitive race refutation. Thresher in turn uses the Z3 SMT solver [12].

6.1 Effectiveness

We present the results in Table 3. Per Section 3.2, SIERRA creates a harness method for each app activity which serves as the entrypoint of the static analysis (on average 10.5 harnesses per app). Next, we show the number of *actions*, i.e., SHBG nodes. The number sums all the actions found in each harness—typically about 160 actions per app. Column 4 shows the total number of HB edges found by SIERRA, and column 5 shows the fraction of HB edges compared with the total number of edges (e.g., if the app has N actions, and all actions are in a happens-before relation, the transitively-closed graph would have $\frac{N*(N-1)}{2}$ edges); the higher this percentage, the less work later stages have to do at refuting potential races, and the lower the chance of false positives. Note how SIERRA manages to find 22% of the theoretically maximum number of edges.

Columns 6 and 7 show the number of racy pairs without and with action-sensitive contexts. The results demonstrate the effectiveness of action sensitivity, as action-sensitive contexts reduce racy pairs by a factor of 5, from 431 to 80.5, which then greatly reduce the number of races to be refuted by the backward symbolic executor.

App	CG+PA	HBG	Refutation	Total
APV	182	18	83	283
Astrid	325	24	938	1,287
Barcode Scanner	173	29	247	449
Beem	397	36	1,664	2,097
ConnectBot	241	54	2,128	2,423
FBReader	1,058	85	1,687	2,830
K-9 Mail	2,936	113	2,759	5,808
KeePassDroid	136	33	288	457
Mileage	1,927	41	3,361	5,329
MyTracks	2,711	52	2,170	4,933
NPR News	562	46	1,546	2,153
NotePad	148	78	702	928
OpenManager	275	53	715	1,043
OpenSudoku	253	36	612	901
SipDroid	278	71	488	837
SuperGenPass	87	16	419	522
Tippytipper	133	32	285	450
VLC	738	30	793	1,561
VuDroid	67	29	405	501
XBMC	2,438	39	1,038	3,515
Median	*1,310*	*28.5*	*560.5*	*1,899*

Table 4. SIERRA efficiency on the 20-app dataset: running time for each stage and total, in seconds.

After refutation (column 8) the median number of races is reduced substantially, to just 33, which we believe is very effective for developers. Section 6.4 compares SIERRA's results with EventRacer's (last column).

We have manually inspected the races reported by SIERRA and classify them into true races (median = 29.5) and false positives (median = 8.5) in columns 8 and 9. Section 6.5 contains a detailed analysis of true/false positives.

6.2 Efficiency

Table 4 shows the results of efficiency experiments. For each app, we show the time, in seconds, it took to run each analysis stage. The front-end analysis with WALA typically takes 1,310 seconds per app (CG column). SHBG construction took 28.5 seconds which is quite efficient. Unsurprisingly, refutation takes about 560.5 seconds per app due to symbolic execution. In total, SIERRA takes about 1,899 seconds per app, which is acceptable for a static analysis.

6.3 Harmful Race Example

The NPR News app contains a harmful event race that may result in incorrect view states – such a race is hard to detect dynamically. The NewsListActivity contains a ListView to show the news list. When new data must be loaded, the app creates background threads, via ImageLoaderTask, to load a list of news items – each item bundles images and text from a certain URL. Similar with the example in Section 2.2, the program does not take scroll events into consideration. If a scroll event occurs before the background ImageLoaderTask posts back data, the ListView will create another ImageLoaderTask to load the new image. If the new image comes before the old one, the old image will replace the new one hence displaying the incorrect image to the user. Triggering this race requires a specific event order – this order can easily elude dynamic race detectors. There are multiple ways to fix this bug. The key

App	Bytecode size (KB)	Effectiveness						Efficiency (time in seconds)			
		Harnesses	**Actions**	**HB edges**	**Ordered** (%)	**Racy pairs**	**After refutation**	**CG**	**HBG**	**Refutation**	**Total**
Median	*1,114*	*4.5*	*67.5*	*1,223*	*17.3*	*68*	*43.5*	*139*	*27*	*648*	*960*

Table 5. SIERRA effectiveness and efficiency on the 174-app dataset.

is to associate the background ImageLoaderTask with the URL for each news item. If the downloaded image does not match the item's URL, then the image should not update the view.

6.4 Comparison with Dynamic Race Detection

We also ran EventRacer Android [7], the most advanced dynamic race detector to date, on our test apps. We show the dynamic detection results in the last column of Table 3. Out of the 20 apps, we could not run EventRacer Android on Astrid and SipDroid. We then considered the high priority races that occur in app code. After analyzing the 182 races reported by EventRacer Android in 18 apps, we found that 102 of them are false positives because they are protected by guard conditions. EventRacer Android uses a concept called "Race coverage" to filter ad-hoc sychronization races, but it only reasons about primitive type variables. Most of the 102 false positive races are protected by pointer checking condition (e.g., `var != NULL` or `var == NULL`). SIERRA can successfully filter out these false positives because it uses combined path and points-to queries. For such cases, SIERRA has the ability to provide *more accurate results* than EventRacer.

There were also 15 races reported by EventRacer that SIERRA did not report because they could be ruled out; the races fall into two categories. First, EventRacer considers that UI actions can occur after Activity lifecycle callbacks (e.g., onClick after onStop). However, SIERRA rules this out because when an Android Activity goes to an invisible state (i.e., is stopped), an UI callback cannot be executed. Second, EventRacer considers UI and UI action as racy, but SIERRA can order UI events (Section 4.3). The remaining races missed by EventRacer are in actions the dynamic detector does not cover. This demonstrates the soundness advantage of a static approach compared with a dynamic approach.

6.5 Discussion

False positives. Thanks to action-sensitivity, SHBG and symbolic execution, SIERRA is able to filter a great amount of false positives. However, we found some cases where false positives may happen. For example, in OpenManager, SIERRA reports a race as follows: both onCreate and onClick create a thread that fetches some data from disk and posts callbacks to update the ListView items. SIERRA considers thread callbacks as non-deterministic. But there is an implicit dependency in the app: onClick can only be triggered after the ListView is filled with data by the thread from onCreate. Such implicit dependencies are beyond the current capabilities of SIERRA. Another type of false positives comes from over-approximate merging in arrays or containers. SIERRA uses index-insensitive analysis to handle instances stored into an array or list. Finally, a symbolic executor timeout may also produce false positives – if we cannot refute within the time budget, we report a potential race to maintain soundness.

Benign races. Symbolic execution is instrumental in filtering out the vast majority of candidate races. The reported races are true positives because SIERRA witnesses a feasible path. However, a true race does not mean it is harmful. Actually, the majority of the true races are due to guard variables in the control flow graph. For example, in Figure 8, SIERRA reports a true race of reading mIsRunning on line 3 of action A and writing it on line 14 of action B. Note that mIsRunning is a guard variable to protect mAccumTime from incorrect access. If action A happens first, the read value is $true$ in action A, and in the alternative order, the read value is $false$. Although this race is a true race, it is arguably benign. We have examined the race reports ("After Refutation" column) and found that 74.8% fit this pattern. For the remaining race reports, SIERRA witnesses different values of an instance variable being set in alternative order of the actions. To conclude, all the true races reported by SIERRA are due to bad programming practices and should be fixed, but the precise extent of the harm inflicted upon the app varies from race to race.

False negatives. SIERRA is sound up to reflection and native code, which is standard practice in Java static analysis. Reflection use beyond the capabilities of DroidEL and unsafe concurrent use of native code use might result in false negatives, i.e., missing races.

6.6 Results on the 174 App Dataset

SIERRA's results (medians) on the additional set of 174 apps which were not subject to manual analysis are shown in Table 5. SIERRA typically reports 43.5 potential races per app, and the analysis takes 960 seconds. These results are mostly in line with the 20-app dataset but we believe the results are more indicative due to the larger set size (174 vs. 20).

7 RELATED WORK

Hong and Kim [15] have surveyed race detection techniques for traditional programs. Out of the 43 tools/approaches surveyed, only 7 were static since, as they noted, "the accuracy of [static] execution models is often low because of the imprecision inherent to static analysis methods." Hence there is a clear need for accuracy in static race detection.

Event-driven race detection. Recent works have looked at detecting event-driven races. EventRacer [21, 22] detects event-driven races in web applications while EventRacer Android [7], CAFA [16] and DroidRacer [18] focus on Android apps. These approaches are all dynamic, hence prone to false negatives and dependent on high-quality inputs; these drawbacks are the main impetus for our work.

Race detection for traditional apps. Race detection has been widely studied; proposed approaches were either static [13, 26] or dynamic [11, 14]. However, these efforts have mainly focused on detecting multi-threaded data races in applications running on desktop or server platforms. In Android, event-driven races are 4x–7x more numerous than data races [16, 18]. Moreover, techniques geared at desktop/server programs can be ineffective for detecting event-based races. For example, traditional dynamic race detectors assume that instructions executed on the same thread have program order. However, this is not true for Android due to asynchronous programming model and Looper events arriving in non-deterministic order.

Static analysis for Android. Many static analysis approaches for Android have been proposed, with specific purposes such as constructing GUI models [10, 23], or information flow [5]. Hopper [9] also uses backward symbolic execution but with a different goal, finding null pointer dereferences. We employ an array of techniques, that while geared at finding races, we believe can also be used as a general, precise static analysis framework for Android apps.

8 CONCLUSIONS

We have presented SIERRA, the first (to our knowledge) approach for static event-based race detection in Android apps. Existing Android race detectors are dynamic, as are most race detectors for traditional programs, due to the difficulty of ordering memory accesses statically. We show that, by employing precise, automatically-constructed harnesses and a static happens-before graph, we can order actions quite effectively. Further, by employing action-sensitivity as well as symbolic execution we can eliminate a large percentage of false positives. Experiments reveal that our approach is effective at finding true races without a large number of false positives, yet has acceptable performance. We believe that SIERRA opens the way for precise analysis of, and race detection in, event-based systems in general.

REFERENCES

[1] Mobile/Tablet Operating System Market Share, Oct 2015. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1.

[2] F-Droid, 2017. https://f-droid.org/.

[3] Android Developers. Activity Lifecycle, 2017. http://developer.android.com/reference/android/app/Activity.html.

[4] Android Developers. App Components, 2017. https://developer.android.com/guide/components/index.html.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[6] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 641–660, New York, NY, USA, 2013. ACM.

[7] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable race detection for android applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 332–348, New York, NY, USA, 2015. ACM.

[8] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: Precise refutations for heap reachability. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 275–286, New York, NY, USA, 2013. ACM.

[9] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective control-flow abstraction via jumping. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 163–182, New York, NY, USA, 2015. ACM.

[10] Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. Droidel: A general approach to android framework modeling. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 19–25, New York, NY, USA, 2015. ACM.

[11] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: Proportional detection of data races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 255–268, New York, NY, USA, 2010. ACM.

[12] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[13] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[14] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, 2009.

[15] Shin Hong and Moonzoo Kim. A survey of race bug detection techniques for multithreaded programmes. *Softw. Test. Verif. Reliab.*, 25(3):191–217, May 2015.

[16] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, New York, NY, USA, 2014. ACM.

[17] IBM T.J Watson. WALA, 2017. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[18] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 316–325, New York, NY, USA, 2014. ACM.

[19] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005.

[20] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[21] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 251–262, New York, NY, USA, 2012. ACM.

[22] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 151–166, New York, NY, USA, 2013. ACM.

[23] Atanas Rountev and Dacong Yan. Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, 2014.

[24] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *in Steven S. Muchnick and Neil D. Jones (eds.), Program Flow Analysis: Theory and Applications, Prentice-Hall, Englewood Cliffs, New Jersey*, pages 189–234, 1981.

[25] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM.

[26] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 205–214, New York, NY, USA, 2007. ACM.

[27] B. Zhou, I. Neamtiu, and R. Gupta. Experience report: How do bug characteristics differ across severity classes: A multi-platform study. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 507–517, Nov 2015.

[28] Bo Zhou, Iulian Neamtiu, and Rajiv Gupta. A cross-platform analysis of bugs and bug-fixing in open source projects: Desktop vs. android vs. ios. In *19th International Conference on Evaluation and Assessment in Software Engineering, EASE 2015*, page 10, April 2015.