Static Detection of Event-based Races in Android Apps

Abstract
Event-based races are the predominant source of concurrency errors in Android apps. So far all the approaches for detecting event-based races have been dynamic. Due to their dynamic nature, these approaches suffer from coverage and false negative issues, and despite being dynamic they still have a high rate of false positives. We introduce a static approach and tool, named SIERRA, for detecting Android event-based races centered around a new concept of “concurrency action” (that refines 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. Our approach introduces several key enablers for inferring order relations statically: models which impose order among lifecycle and GUI events; static analysis refinements, e.g., path-sensitive pointer analysis; and finally, on-demand path sensitivity via backward symbolic execution to further rule out false positives. We evaluate SIERRA on 194 Android apps. Of these, we chose 20 apps for manual analysis and comparison with a dynamic race detector. Experimental results show that our approach is effective and efficient, typically taking 1,230 seconds to analyze an app and revealing 33 potential races. Moreover, SIERRA can refute false positive results reported by the dynamic detector. Our approach opens the way for precise analysis and static event race detection in other event-driven systems beyond mobile platforms.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification—Reliability, Validation; D.2.5 [Software Engineering]: Testing and Debugging—testing tools

General Terms Reliability, Verification

Keywords Mobile applications, Concurrency, Event-based race, Happens-before, Static analysis, Google Android

1. Introduction
Android is the dominant software platform for smartphones and tablets [2]. 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 [30]. In fact, 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 [29]. Android concurrency research has shown that the majority of Android race bugs are event-driven races [7, 17, 20]; per Maiya et al. [20], 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 [20], CAFA [17], and EventRacer Android [7]. However, dynamic detectors have several issues. First, due to their dynamic approach, they are prone to false negatives, i.e., miss actual bugs. Second, only 3% of the races they report are actually harmful [18]. Third, their effectiveness hinges on high-quality inputs that can ensure good coverage [6] as well as efficient ways to explore schedules.

Android apps employ 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 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.

To address these issues, we propose a static event race detection approach. 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 Android Framework. Second, apps consist of activities (separate screens) that can be navigated back and forth [6]; further, each activity comprises of GUI objects which can be accessed in relatively unconstrained order [25]. 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, the use of 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 refinement to significantly improve precision, e.g., automatically-constructed harnesses to kickstart the static analysis, and a novel path-sensitive pointer analysis (Section 3). Third, we introduce 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.

We provide an evaluation of our approach in Section 6. We evaluated SIERRA on 194 apps, and chose 20 of them for further manual analysis. Experiments show that our techniques are effective, discovering about 1,223 happens-before edges and 176 racy pairs per app. Refutation reduces this substantially, to just 34 race reports per app. SIERRA is efficient: it typically requires 1,230 seconds to analyze an app (of which 843 seconds are spent in the symbolic execution phase) which is acceptable for a static analysis. Path-sensitive pointer analysis is very effective, too, eliminating 58.5% of spurious nodes and 43% of spurious edges in the call graph. For the 20 manually-analyzed apps, we ran EventRacer Android [7], the most advanced dynamic race detector to date. We found that SIERRA compares quite well: while it reports more potential races, it 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.
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 an Android Framework (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 in turn 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 a search 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 Intents) — inter-component races are possible (Section 2.3).

Threads. Android has three main 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 apparent later on. Each Android app has a “main” thread, also known as the UI thread, that has the privilege and responsibility of 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 IPC communication. Apps are typically performing the actual work in background threads, which notify the main thread when a GUI update must be performed by posting a message to the main thread’s processing queue.

2.2 Intra-component Race

Figure 1 shows a harmful event-based race — more precisely, an intra-component race, as it happens within one activity. The NewsActivity, shown on the left, has a RecyclerView to display the news items. RecyclerView 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 RecyclerView 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 put 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 RecyclerView 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 RecyclerView 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 that case). 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 BroadcastReceiver 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 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, 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). Then we describe how to design a best-fit context abstraction (Section 3.3) for statically analyzing event-driven programs, and an effective refinement approach (Section 3.4) to make pointer analysis precise.

3.1 Architecture

Figure 3 shows the architecture of SIERRA. First, we leverage DroidEL [10], a static Android framework 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(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 intergrated into our harness generator (described later) that will drive the static analysis.

Second, we leverage WALA [19] 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. Selecting the appropriate context in pointer analysis is key to achieving scalability and precision. Prior research has shown that object sensitivity is an effective abstraction for object oriented languages. However, in event-driven systems like Android, object sensitivity is not precise enough. SIERRA introduces a novel abstraction called hybrid context sensitivity which selectively combines object sensitivity, call-site sensitivity, and thread sensitivity (Section 3.3).

In addition, SIERRA introduces an on-demand pointer analysis refiner. To keep the analysis scalable, all main-stream pointer analysis tools are path-insensitive. However, this may lead to severe loss of precision in some scenarios (example provided in Section 3.4). Thus SIERRA employs a staged refinement process: applying path-insensitive pointer analysis and call-graph generation first, which produce over-approximate results; then finding control dependencies intra-procedurally between input argument and return references, and finally re-propagating the points-to constraints to achieve more precise results (Section 3.4).

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
class Harness {
    public static void main() {
        NewsActivity a = new NewsActivity();
        a.onCreate();
        a.onStart();
        a.onResume();
        while (true) {
            switch (*) {
                case 1: a.onClick(); break;
                case 2: a.onScroll(); break;
                .......
            }
        }
    }
}

Figure 4: Harness example.

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 the app harness creation process. As SIERRA performs whole-program analysis, we need to find the program’s entrypoints. While in traditional Java programs we would start at main(), Android apps have no main. Rather, in Android, the control flow of an app is orchestrated by the AE, which invokes lifecycle callbacks, such as onCreate when the app is created, or onDestroy when the app is destroyed. Besides these lifecycle callbacks, 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 how to write a harness 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 Hybrid Context 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: call-site-sensitivity (k-cfa) [26] and object-sensitivity (k-obj) [21].

Prior research [22, 27] has shown that object-sensitivity increases precision; however, we have found that it is not precise enough for our event-driven programs (Android apps). Object-sensitivity selects receiver object’s allocation site as context if the caller method is a dispatch and merge the callee by this context. This over-approximate merge may reduce precision. Take GestureDetector’s onTouch(MotionEvent e) function as an example. The onTouch distinguishes user’s action by comparing the current MotionEvent with the previous one, and decide whether is action is fling or long press. It invokes handler.sendMessage(FLING, ...) to notify the handler to handle fling action, and invokes handler.sendMessage(LONG_PRESS, ...) for long press action. If we use object-sensitivity alone, the two callees will be merged because their context is the same handler object, thus precision is lost. Note that this pattern is very common in Android apps.

Based on the insight that we should distinguish as many actions as possible, SIERRA employs a new context sensitivity named hybrid-context-sensitivity which consists of object-sensitivity, call-site-sensitivity and thread-sensitivity. Whenever an action creation site is reached (e.g., thread creation, message sent, runnable posted, per Table 1), both the object context and call site context are selected. In our experiments, we found that this context abstraction is the most effective one in the trade-off between scalability and precision: all the actions are found with no incorrect merge. Thread-sensitivity is used to record which thread the allocated object belongs to. Note that event-driven races, occur due to concurrent unordered accesses on the same looper thread. Each Handler object contains a Looper which maintains the underlying MessageQueue. Thus, when checking the potential racy pair of actions, we must check whether the action handlers’ Looper objects belong to the same thread.

3.4 Path-sensitive Pointer Analysis Refinement

We introduce a novel path-sensitive pointer analysis which increases precision while relying on on-demand refinement for efficiency. State-of-the-art pointer analysis frameworks, e.g., WALA, Soot, Doop and Chord, are path-insensitive to keep the analysis scalable. Path sensitivity substantially increases the precision of pointer analysis, at the expense of analysis time — the analysis has to solve the complex path condition constraints which are exponential, hence problematic for large programs.

However, the imprecise nature of path-insensitive pointer analysis can lead to a large number of false positives. For ex-
### Table 1: Actions and HB introduction.

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

![Figure 5: Imprecision of path-insensitive pointer analysis.](image)

```java
1 static {
2     static ListView listView = new ListView();
3     static TextView textView = new TextView();
4     static Button button = new Button();
5     ......
6 }
7 View inflateViewById(int id) {
8     switch (id) {
9         case R.id.listView: return listView ;
10         case R.id.textView: return textView;
11         case R.id.button: return button;  
12         ......
13     }
14 }
```

An example, SIERRA pre-processes each app with DroidEL, a tool that automatically maps the view id to its data type. Figure 5 shows an example of how DroidEL handles layout inflation. Android apps can define views and layouts statically, in XML files. At runtime, the Android Framework dynamically inflates the layout and provides `findViewById` API to allow the app get the view by its id. DroidEL pre-processes the app’s XML layout files and generates stubs as shown in Figure 5. Due to path insensitivity, our pointer analysis tool (WALA) does not model, nor solve, any path/branch constraint during points-to set propagation and call graph generation. Thus, suppose there is a call site that invokes `findViewById(id)` to get the `ListView` object. WALA would return a myriad of views, i.e., `listView`, `textView`, `button`, etc., because the points-to set only contains the `may` information. The imprecision would be further compounded if WALA tried to use fix-point computation for points-to sets from the spurious call graph edges.

SIERRA solves the path insensitivity problem via on-demand refinement. Note that it is very expensive to solve the path constraints together with first-round pointer analysis and call graph construction. SIERRA uses WALA to do path-insensitive analysis in the first round. The result is an over-approximated call graph and points-to set. Then for each callsite in the app that invokes `findViewById(id)`, SIERRA finds the view that matches the given view id (a constant integer in general), and updates the returned points-to set. So SIERRA refines the returned points-to set with the exact `View` object associated with the given id. Starting with this refined point-to set as the seed, SIERRA repropagates the pointer constraints in WALA that can be reached transitively from the seed including the local pointers, method dispatch calls, return pointers, and heap data. The repropagation is fixed-point-based, that is, we iterate until the all the points-to set reach the fixed point. From our experiments, the on-demand pointer analysis refinement plays a crucial role in improving precision. The nodes in call graph are reduced by 75% and spurious edges are reduced by 56% (Section 6.1).

### 4. Happens-before Relationship

Prior event-driven race detectors for Android have defined dynamic happens-before rules [7, 17, 20]. Those definitions do not easily translate here, as our approach is static and uses 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 now 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 \( \rho \), i.e., \( \pi(x) = \rho \). Memory accesses \( \alpha \) are \( \langle x, \tau, A \rangle \) bundles, indicating that variable \( x \) is accessed using access type \( \tau \) (read or write) in activity \( A \).

**Racy pairs.** We define racy pairs as follows: accesses \( \alpha_1 \) and \( \alpha_2 \) form a racy pair if they come from different activities \( A_1 \) 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 block of our approach. An action represents *context-sensitive event handling*. In Table 1 we describe how we identify HB nodes and edges so they can be added to our SHBG. When our 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 either using 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).

### 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.

\(^2\)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.
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 7 on a simplified GUI layout, where an app cycles and nondeterministically chooses between onClick1() or onClick2(); onClick3(). Since onResume pre-dominates onClick1 we have:

onResume ≺ onClick1
onResume ≺ onClick2
onClick2 ≺ onClick3.

4. Intra-procedural domination. Let us assume that a method M in activity A has two outgoing calls e₁ and e₂ that posts actions A₁ and A₂, respectively. If e₁ dominates e₂ then A₁ ≺ A₂; this is intuitive because e₁ will always be invoked before e₂ and by the time e₂ executes (and gets a chance to post A₂), A₁ has already been posted, so A₂ can only be posted after A₁.

5. Inter-procedural, intra-action domination. This is similar to rule 4 above, but the difference is that e₁ and e₂ can be in two separate methods M₁ and M₂ of the same activity A. Note that e₁ cannot straight-up dominate e₂ because e₂ might be invoked from a context that does not involve e₁. We leverage WALA’s interprocedural CFG (ICFG) to address this issue as follows: we remove e₁ from the ICFG and check whether e₂ is still reachable; if it is not reachable, then de facto e₁ dominates e₂ and we add A₁ ≺ A₂. If, on the other hand, e₂ is still reachable when e₁ is absent, we do not add any HB edges.

6. Inter-action transitivity: If A₁ ≺ A₂ and A₂ posts an action A₃, and A₂ posts an action A₄, then A₃ ≺ A₄. We illustrate this in Figure 8. On top (Figure 8 (a)) we show the order relation. On the bottom we show the two possible execution schedules for this order. A₁ executes first, and during its execution, it posts A₃. Importantly, by the time A₁ finishes, A₃ is already posted. We have two cases: Figure 8(b) when A₂ executes before A₂ does, hence A₃ ≺ A₄ because A₄ has not even been posted when A₂ finishes; and Figure 8(c) when A₂ executes first, but because A₃ has already been posted when A₂ starts executing, A₄ can only be posted after A₃ hence A₃ ≺ A₄. Note that we can infer these orderings thanks to the looper atomicity guarantee.

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

A₁ ≺ A₂ ∧ A₂ ≺ A₃ ⇒ A₁ ≺ A₃

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 Looper and Handler 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.


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

Figure 9: Refutation helps eliminate this false positive in the OpenSudoku app.

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 the fact that they are racing, i.e., we can prove that they are ordered (we do so via symbolic execution, described next).

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 [24]; 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 9). 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 on both left and right). 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 conditional branch statement on line 3 (left), it adds a path constraint \( \{ \text{mIsRunning} = \text{true} \} \), i.e., a precondition to reach \( α_A \). The backward analysis continues until reaching the boundary of 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 (right) SIERRA chooses to enter the block guarded by line 2 (right) because the guard condition is consistent with the path constraint \( \{ \text{mIsRunning} = \text{true} \} \). Finally, a conflicting constraint is found when the path reaches line 3 (right) which performs a strong update to the mIsRunning. This strong update means the path constraint after this statement must be \( \{ \text{mIsRunning} = \text{false} \} \) which conflict 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], but modified 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 source and sink. The candidate race is a true positive, \( \text{if} \) in both orderings of actions \( A \) and \( B \), there does exist a feasible path from \( α_A \) to \( α_B \), and vice versa. Note that the path constraint can be a primitive constraint, or a points-to constraint (e.g., a pointer equal null or not). In our experiments, we found that points-to queries are particular useful to refute false positives due to the singleton pattern.

On-demand constant propagation. When the action is Handler.handleMessage(Message m), program behavior depends on the values of Message’s what, arg1 and arg2 fields. To increase precision, we introduce constraints to check if any of these fields are constant integers (e.g., the FLING and LONG_PRESS in Section 3.3’s example) 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.

Caching. The running time of refutation varies, depending on the complexity of the program. A refutation could be terminated by the executor if the system runs out-of-memory (OOM) or exceeds maximum number of paths (in our setting 5,000 paths) allowed. In either case, SIERRA soundly reports the race though it might be a false positive. To prevent redundant computation, SIERRA memoizes (caches) all the call graph nodes visited so far in a refuted query. Later queries first check the cache. If the current node in a path already 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 evaluate our approach 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 a wide range of categories: news apps, video players, email clients, etc.. First, 174 apps with a median size of 1.1MB from F-droid [1], an online open source repository for Android, were selected for automatic testing. Next, we reuse Gator [25]’s benchmark which contains 20 apps as listed in Table 2. We chose these datasets 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 the Google
### Table 2: App popularity and size for the 20-app dataset.

<table>
<thead>
<tr>
<th>App</th>
<th>Installs</th>
<th>Bytecode size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>APV</td>
<td>500,000–1,000,000</td>
<td>736</td>
</tr>
<tr>
<td>Astrid</td>
<td>100,000–500,000</td>
<td>5,400</td>
</tr>
<tr>
<td>Barcode Scanner</td>
<td>100,000,000–500,000,000</td>
<td>808</td>
</tr>
<tr>
<td>Beem</td>
<td>50,000–100,000</td>
<td>1,700</td>
</tr>
<tr>
<td>ConnectBot</td>
<td>1,000,000–5,000,000</td>
<td>700</td>
</tr>
<tr>
<td>FBReader</td>
<td>10,000,000–50,000,000</td>
<td>2,800</td>
</tr>
<tr>
<td>K-9 Mail</td>
<td>5,000,000–10,000,000</td>
<td>489</td>
</tr>
<tr>
<td>KeePassDroid</td>
<td>1,000,000–5,000,000</td>
<td>228</td>
</tr>
<tr>
<td>Mileage</td>
<td>500,000–1,000,000</td>
<td>641</td>
</tr>
<tr>
<td>MyTracks</td>
<td>100,000–500,000</td>
<td>5,300</td>
</tr>
<tr>
<td>NPR News</td>
<td>1,000,000–5,000,000</td>
<td>1,500</td>
</tr>
<tr>
<td>NotePad</td>
<td>5,000,000–10,000,000</td>
<td>228</td>
</tr>
<tr>
<td>OpenManager</td>
<td>NA</td>
<td>77</td>
</tr>
<tr>
<td>OpenSudoku</td>
<td>1,000,000–5,000,000</td>
<td>170</td>
</tr>
<tr>
<td>SipDroid</td>
<td>1,000,000–5,000,000</td>
<td>539</td>
</tr>
<tr>
<td>SuperGenPass</td>
<td>10,000–50,000</td>
<td>137</td>
</tr>
<tr>
<td>TippyTpper</td>
<td>100,000–50,000</td>
<td>79</td>
</tr>
<tr>
<td>VLC</td>
<td>10,000,000–50,000,000</td>
<td>1,100</td>
</tr>
<tr>
<td>VuBMC</td>
<td>100,000–50,000</td>
<td>63</td>
</tr>
<tr>
<td>XBMC, remote</td>
<td>100,000–500,000</td>
<td>1,100</td>
</tr>
</tbody>
</table>

### 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 harness via harness generator. WALA has provided points-to information and call graph construction. The hybrid context selector and path sensitive refinement were implemented as plugins to WALA. SIERRA modifies Thresher to run goal-directed path-sensitive race refutation. Thresher in turn uses the Z3 SMT solver [13] to solve constraints.

### 6.1 Effectiveness

We present the results in Table 3. For each activity of an app, SIERRA creates a harness function 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 up 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 $N(N-1)/2$ 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.

Column 6 shows the number of rcay pairs (median = 431). These are potential races in the app and the AF; we ignore potential races in the Java library since developers have little control over those. After refutation (column 7) the median number of races is reduced substantially, to just 34, which we believe is very effective for developers.

To quantify the effectiveness of path-sensitive refinement, columns 8 and 9 show the percentage reduction in the call graph after our analysis; this reduction is substantial, 75% of nodes and 56% of edges.

### 6.2 Efficiency

Table 4 shows the results. For each app, we show the time, in seconds, it took to run each analysis stage. The front-end analysis with WALA typically takes 157 seconds per app (CG column); our pointer analysis refinement took 88 seconds which is fairly efficient. SHBG construction took 41 seconds which is quite efficient as well. Unsurprisingly, refutation takes about 1,243 seconds per app due to symbolic execution. In total, SIERRA takes about 1,628 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. 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 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 columns of Table 3. Out of the 20 apps, we could not run EventRacer Android on Astrid and SipDroid. After analyzing totally 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
### 6.5 Discussion

**False positives.** False positives (FP) are inevitable in static analysis because of the $k$-limit nature of object/callsite sensitivity which merge the abstraction object with same $k$-obj/cfa into same abstraction. In our experiments, we examine 250 races reported by SIERRA and found that 61 of them are FPs due to imprecise alias information.

**6.5 Discussion**

**False positives.** False positives (FP) are inevitable in static analysis because of the $k$-limit nature of object/callsite sensitivity which merge the abstraction object with same $k$-obj/cfa into same abstraction. In our experiments, we examine 250 races reported by SIERRA and found that 61 of them are FPs due to imprecise alias information.
**Benign races.** Thanks to symbolic execution, SIERRA is able to filter 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 feasible races are used as guard in a control flow graph. For example, in Figure 9, SIERRA reports a true race of reading mIsRunning on line 3 of action A and writing it on line 3 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 become false. Although this race is a true race, we consider it as benign. We have examined 250 races after refutation phase and found that 187 races match this pattern. For other races, SIERRA witnesses different values of an instance variable being set in alternative order of the actions. However, it is difficult to discern whether the race is benign or harmful without being closely familiar with the app.

**False negatives.** SIERRA may have false negatives, i.e., missing races. Besides incomplete modeling of AF callbacks just mentioned, the refutation phase may miss races. For experiments we set SIERRA’s maximum number of exploring paths to 5,000, i.e., around 5-minute time limit per race on our setup. If the symbolic executor runs out of memory or exceeds the max path number, SIERRA regards the race to be unlikely to occur. Note that this could miss a race if the feasible path is not visited before reaching the limit. However, the design goal of SIERRA is effectiveness, an an acceptable trade-off to protect the user from an overwhelming number of races reported.

### 6.6 Results on the 174 App Dataset

Besides the 20 apps, we also select 174 open source apps from F-droid, an online repository for open source Android apps. For each app, we use DroidEL to pre-process the apk and use our harness generator to generate the harness. The median data is shown in Table 5; the data is mostly in line with the results on the 20-app dataset but we believe the results are more indicative due to the larger set size.

### 7. Related Work

Hong and Kim [16] 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 for Web and Android apps.** Recent works have looked at detecting event-driven races. EventRacer [23, 24] detects event-driven races in web applications while EventRacer Android [7], CAFA [17] and DroidRacer [20] 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. Prior efforts have used either static [14, 28] or dynamic [11, 15] analysis to detect races. 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 [17, 20]. 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, 25], or information flow [5]. Hopper [9] also use 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 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 hybrid context-sensitivity as well as symbolic execution we can discover further memory access orderings hence eliminating a large percentages 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 our approach opens the way for precise analysis of, and race detection in, event-based systems in general.

<table>
<thead>
<tr>
<th>App</th>
<th>Bytecode size (KB)</th>
<th>Harnesses</th>
<th>Actions</th>
<th>HB edges</th>
<th>Ordered (%)</th>
<th>Racy pairs</th>
<th>After refutation</th>
<th>CG</th>
<th>PA refine</th>
<th>HBG</th>
<th>Refutation</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median</td>
<td>1,114</td>
<td>4.5</td>
<td>67.5</td>
<td>1,223</td>
<td>17.3</td>
<td>176</td>
<td>33.5</td>
<td>96</td>
<td>76.5</td>
<td>32</td>
<td>843</td>
<td>1,230</td>
</tr>
</tbody>
</table>

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


