Scalable Race Detection for Android Applications

Pavol Bielik
Department of Computer Science
ETH Zürich, Switzerland

Veselin Raychev
Department of Computer Science
ETH Zürich, Switzerland

Martin Vechev
Department of Computer Science
ETH Zürich, Switzerland

firstname.lastname@inf.ethz.ch

Abstract

We present a complete end-to-end dynamic analysis system for finding data races in mobile Android applications. The capabilities of our system significantly exceed the state of the art: our system can analyze real-world application interactions in minutes rather than hours, finds errors inherently beyond the reach of existing approaches, while still (critically) reporting very few false positives.

Our system is based on three key concepts: (i) a thorough happens-before model of Android-specific concurrency, (ii) a scalable analysis algorithm for efficiently building and querying the happens-before graph, and (iii) an effective set of domain-specific filters that reduce the number of reported data races by several orders of magnitude.

We evaluated the usability and performance of our system on 354 real-world Android applications (e.g., Facebook). Our system analyzes a minute of end-user interaction with the application in about 24 seconds, while current approaches take hours to complete. Inspecting the results for 8 large open-source applications revealed 15 harmful bugs of diverse kinds. Some of the bugs we reported were confirmed and fixed by developers.

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

Keywords Data Races, Happens-before, Android, Nondeterminism

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA

http://dx.doi.org/10.1145/2814270.2814303

1. Introduction

Modern smartphones are powerful computing platforms able to run complex applications, thus end users increasingly rely on them for various computing needs. A distinguishing feature of mobile applications is their event-driven nature: the application must handle asynchronously generated events from a diverse set of sources including the user interface, the network, the sensors and the framework. Unfortunately, this asynchrony can cause data races that potentially corrupting the overall application behavior. As a result, researchers have recently devised program analyses aiming to automatically discover such harmful behaviors (e.g., for Android applications [6, 9]).

While these works represent a promising step forward, they still suffer from several drawbacks, including: (i) inability to analyze realistic user interactions and applications due to poor analysis scalability requiring hours to handle even short interactions, (ii) reduced testing coverage missing important data races, and (iii) incomplete happens-before model of Android concurrency also affecting (ii). As a result, these analyzers cannot handle real-world applications and user interactions in any reasonable time. A more detailed discussion of prior work is provided in Section 2.

This Work In this work we present the first scalable dynamic analysis system for finding data races in Android applications and demonstrate the scalability and precision of our approach by applying it to 354 real-world applications. Our system is based on several key ingredients: new algorithms for building and querying the happens-before (HB) graph (the algorithms are of interest beyond Android), a thorough and formal definition of the HB model capturing Android concurrency, handling of potential sources of concurrent interference beyond user provided source code, and a detailed set of filters for reporting fewer false positives.

Challenges Arriving at a complete solution was challenging for three reasons. First, formally capturing how concurrency arises in the entire Android platform is difficult as many subtle features of Android concurrency are not well documented. Second, and perhaps most unexpected: existing state-of-the-art analysis algorithms [5], even those targeting event-driven applications [12], do not scale to han-

dling real world Android programs. The reasons are fundamental: either, their vector clock data structures quickly run out of memory [5] when dealing with thousands of events, or as in [12], they focus on simple fork-join models of concurrency which are not suitable when analyzing platforms that exhibit complex happens-before rules (as Android does). Finally, handling the full complexity of realistic applications while reporting few false positives comes with its own set of difficult problems.

Main Contributions The contributions of our work are:

- A thorough and precise *happens-before* model which captures Android-specific concurrency (Section 5). This model is a useful basis for any Android program analyzer.
- A new scalable algorithm for building and querying the happens-before graph (Section 6). This algorithm can deal with complex happens-before rules and is key to enabling analysis of real-world applications and user interactions in seconds.
- An effective set of filters allowing the analysis to report very few false positives (Section 7). These reduce the number of false positives by an order of magnitude.
- A complete implementation of our approach including both, a standalone analyzer as well as an online service for analyzing Android binaries[1].
- An comprehensive evaluation on 354 real-world applications (Section 8). Inspecting the results for 8 open-source apps revealed 15 harmful bugs including displaying old results, crashes after configuration changes or when the user navigates away from the application. Some of the bugs we reported were confirmed and fixed by developers, including errors manifesting *inside* the framework.

2. Related work

Before we proceed further, we compare our work with those that are most closely related. These include CAFA [6] and DROIDRACER [9], both investigating the problem of finding data races in Android, as well as EVENTRACER [11, 12] and SDNRACER [10] which focuses on web pages and software defined networks respectively.

2.1 Comparison with CAFA and DROIDRACER

Our work differs from these approaches in several major ways.

Difference 1: Handling Realistic Interactions Prior work cannot analyze realistic interactions in reasonable time, taking hours (as reported by the authors) with trivial interactions. In contrast, our system analyzes realistic interactions in seconds.

The reason is that our algorithms have better asymptotic time complexity for constructing the HB graph: $\mathcal{O}(N^2)$ vs. $\mathcal{O}(N^3)$ for DROIDRACER. The complexity analysis of CAFA is unavailable, but the poor runtimes suggest it is similar to DROIDRACER's. Note that disabling some features of our algorithm (described in Section 6) leads to *massive* analysis slowdowns: from 11 seconds to 254 minutes.

Difference 2: Happens-Before Model The HB rules in CAFA are inconsistent (e.g., by not considering the effect of barriers) and introduce orderings *contradictory* to those observed in the execution. Further, our HB model is more precise than both prior works as we introduce more true edges (which reduce the number of reported false positives) and do not introduce false edges (which enables us to find more real errors). A detailed summary is found in Section 5.3.

Difference 3: Error Coverage We provide better error coverage than prior work. To report fewer false positives, CAFA only detects errors that lead to null pointer exceptions, while DROIDRACER focuses on interference occurring in user code. In contrast, we detect data races originating from *both* user code *and* between user code and the framework (shown in Section 3.2). Existing approaches cannot detect these races, yet they are important – developers fixed several such errors that we reported.

Difference 4: Tool Availability and Robustness Experimental comparison with these tools was not possible as CAFA is not publicly available while for DROIDRACER, even after discussion with its developers, the tool could not process traces larger than few seconds and kept crashing. In contrast, our complete system is fully available both as a push-button online service where one can simply upload an Android binary and receive the analysis results, as well as a standalone downloadable analyzer available at: `http://www.eventracer.org/android`.

Summary Because of the above differences and the capabilities of our system, we believe that this work presents the first scalable analysis system for finding harmful data races in real-world Android applications.

2.2 Comparison with EVENTRACER and SDNRACER

The EVENTRACER work [11, 12] focuses on detecting data races in another application domain (web pages) but has developed advanced general-purpose offline analysis algorithms [12]. While these algorithms are useful, we find that in our application domain, the main analysis bottleneck is not the processing of the trace, but the act of building the HB graph. Thus, to handle realistic programs in reasonable time, we developed new algorithms presented in Section 6.

Another recent work, SDNRACER [10], defines happens-before rules and a commutativity specification for the domain of software defined networks and demonstrates that the commutativity specification can significantly reduce the number of reported races. The happens-before rules and the commutativity specification are specific to the domain of software defined networks and are not applicable to Android.

[1] `http://www.eventracer.org/android`

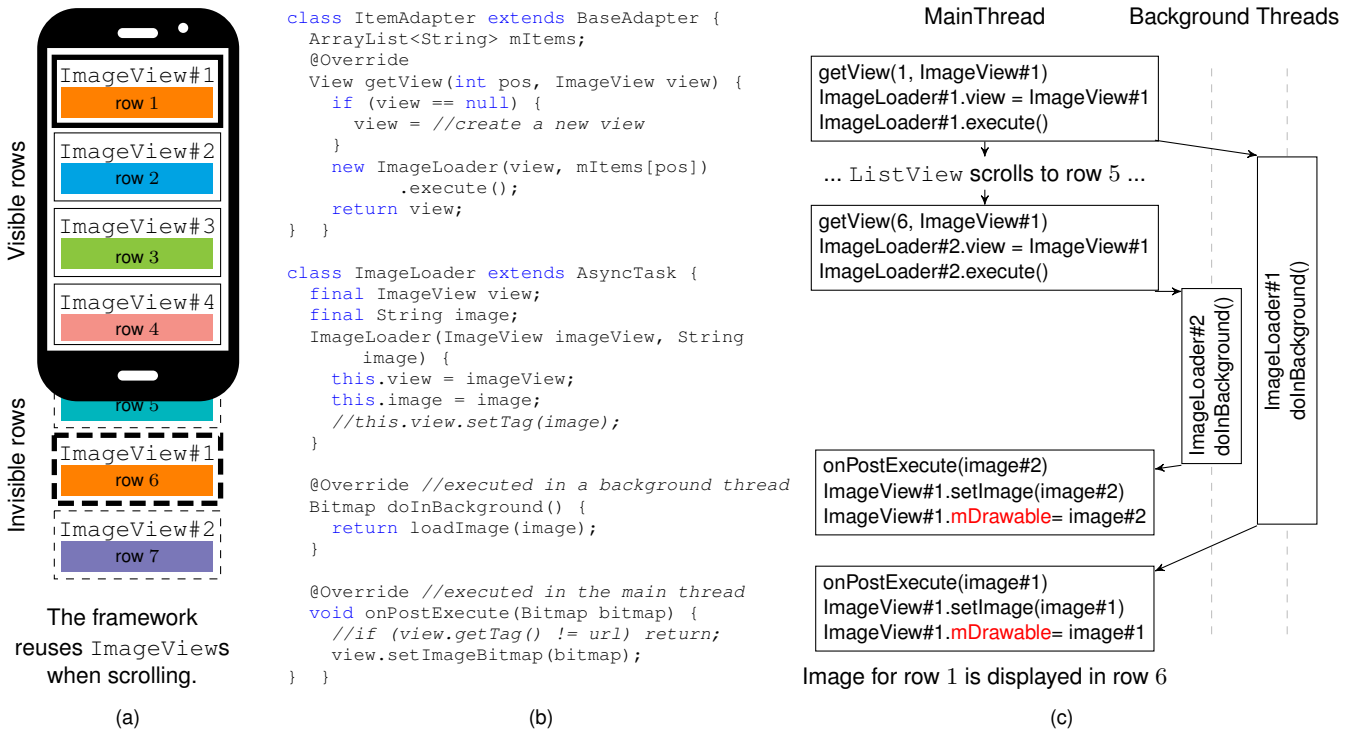


Figure 1. Illustration of a `ListView` with images as rows (row color denotes which image was loaded in the bad execution trace) where: (a) the framework reuses `ImageView` components when scrolling, (b) the relevant application code which starts image download when the row becomes visible, and (c) a bad execution trace displays the incorrect image for row 6.

2.3 Comparison with Testing Tools

A number of Android testing tools were developed in recent years [1–3, 7, 8, 13] that are complementary and focus on input generation while we focus on analysis of a given execution. These approaches often rely on assertions or exceptions to report issues and may need to run an application multiple times and explore a large number of traces before finding harmful behaviors. In contrast, our approach finds violations which do not exist in the given execution but do exist in a permutation of that execution. Thus, it provides more powerful coverage guarantees per explored execution.

3. Overview

In this section we provide three illustrative examples of the different kinds of concurrency errors that can arise in Android applications and show how they manifest as data races. These examples show harmful behaviors detected by our analysis system.

Fundamentally, data races can arise when two events (generated from various sources including the framework, the network, the sensors, and the user interface) can be executed in arbitrary order and they access overlapping memory locations where one of the accesses is a write. In our work we focus on data races between events handled by a single *main thread* (also referred to as a *UI thread*) – a designated thread that facilitates queuing and dispatching of messages allowed to manipulate user interface components.

We note that these data races denote unordered conflicting accesses to the same memory location and even though these accesses have well-defined semantics (events are often executed by the same thread), the resulting conflict can lead to unexpected application behaviors. This is different from traditional data races for which programming languages often give very weak semantics (e.g., incrementing non-atomic variable in a C program). However, for completeness, we also report traditional data races between different threads of the application.

3.1 Data Races Caused by Object Reuse

Consider a simplified version of a harmful bug found by our tool in the `OI File Manager` application which contains a `ListView` user interface component displaying one image per row. Without loss of generality we assume that all rows are supposed to display different images denoted by color as shown in Fig. 1 (a). Each row is identified by an index and contains an `ImageView` in which an underlying adapter loads the respective images from the file system or network as shown in Fig. 1 (b). Each time a new row becomes visible on the screen, either by initially showing the list view or when the user scrolls, the framework implementation of `ListView` invokes the `getView` method of its

adapter to populate the contents of that row. Even though the ListView can contain hundreds of rows, at any given time, only a couple of them fit the screen and are shown to the user. Because only few rows can be shown on the screen at a time, for performance reasons, the components representing the rows (i.e., ImageViews) are reused internally by the framework and are supplied to the application as a parameter of the getView method.

To ensure responsiveness, in its getView method, the application creates an ImageLoader object which is a subtype of the framework class AsyncTask (in Android, AsyncTask is one of the key mechanisms for creating asynchronous behaviors). The application then fires off an asynchronous task while supplying description of an image to be loaded. While the application proceeds, a background thread invokes the procedure doInBackground which asynchronously loads the image and stores it in a suitable Bitmap representation. On completion of method doInBackground, the background thread notifies the *main* thread which in turn invokes the method onPostExecute with the resulting image representation.

Unfortunately, because the ImageView components are reused internally by the framework, it is possible that two asynchronous tasks with the same associated ImageView are loading different images in parallel. One possible execution of the code that illustrates this scenario is shown in Fig. 1 (c). Initially, the framework calls method getView to populate row 1. Let the concrete view object of row 1 be ImageView#1. As a result, the application starts an ImageLoader#1 for loading the image at row 1. However, as the user scrolls the ListView, row 1 may become invisible and instead row 6 is about to be shown. The framework calls the getView method to populate row 6, but reuses the view ImageView#1. In this case, ImageLoader#2 is started to load the image at row 6.

Harmful Behavior In the bad execution trace shown in Fig. 1 (c), both ImageLoaders are executed concurrently in separate threads. Because the framework does not guarantee the order in which ImageLoader tasks will complete, the user may not see the correct image displayed for row 6, but instead sees the image for row 1.

Data Race The harmful behavior described above is a result of an unordered execution of two onPostExecute events updating the same ImageView and manifests itself as a data race on mDrawable field of the ImageView#1 object. The field mDrawable is written inside setImage method defined in the framework which is called from the user code.

Repair To synchronize this example we could ignore any ImageView objects supplied to the getView method and create a new one for each invocation. Although this solution prevents the aforementioned bug, it results in degraded performance. Another, better solution is shown as commented code in Fig. 1 (b). Here, the onPostExecute method checks whether the associated ImageView is still assigned to the same position in the ListView.

```
class MainActivity extends Activity {
  void onCreate() {
    ...
    new Downloader(this).execute(URL); // downloads data
  }

  class Downloader extends AsyncTask {
    ProgressDialog mDialog;

    Downloader(Activity context) {
      mDialog = new Dialog.Builder(context).create(...);
      mDialog.show();
    }

    ArrayList<Data> doInBackground(URL url) {
      return downloadData(url);
    }

    void onPostExecute(ArrayList<Data> data) {
      // if (mDialog.getContext().isDestroyed()) return;
      mDialog.dismiss(); // accesses activity state
      ...//display the data to the user
} } }
```

Figure 2. Example of using Activity in invalid state if the user navigates away while data is being downloaded.

3.2 Data Races Caused by Invalidation

An invalidation error designates that an operation may use an object no longer in a valid state. The example in Fig. 2 shows a simplified version of a data race our tool found in the AnyMemo application. The MainActivity class extends the Activity framework class to display a single full-screen window to the user. When MainActivity is shown to the user, the framework calls the onCreate method which creates an AsyncTask that downloads data in a background thread and upon finishing displays the data to the user. While the data is downloaded in the background, the user is notified about the progress using a ProgressDialog component.

Harmful Behavior When the AsyncTask finishes, the MainActivity object may no longer be in a valid state. This happens for example when the user navigates away by pressing the back button while the data is still downloaded. In this case, invoking mDialog.dismiss() results in an exception that crashes the application. The exception is thrown, because the dialog internally uses reference to the invalid MainActivity object supplied upon its instantiation.

Data Race In this example, the event that destroys object MainActivity when the user navigates away (this event is inside the framework) and the event that runs onPostExecute are unordered. A data race is reported on the mPanels field *inside* the framework class PhoneWindow. This example shows that it is critical to handle not only user level memory locations but also memory locations inside the framework (handling these is beyond the reach of any existing work).

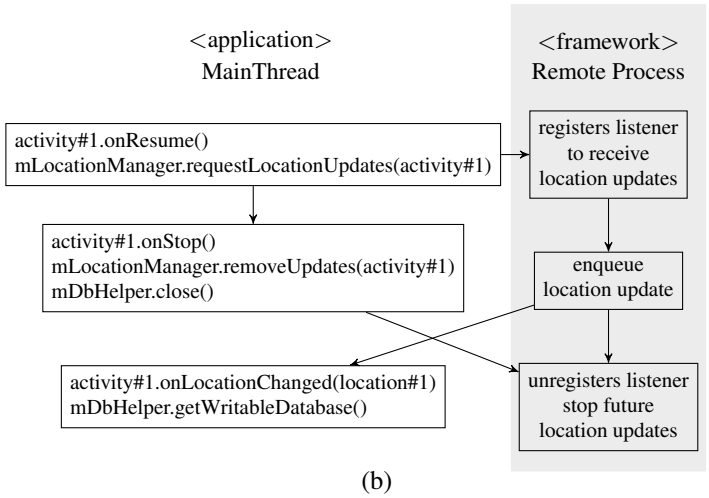
```
public class MainActivity extends Activity
    implements LocationListener {
  SQLiteOpenHelper mDbHelper = new
       SQLiteOpenHelper(this, DB_NAME, DB_VERSION);
  // boolean isActive = false;

  protected void onResume() {
    // isActive = true;
    mLocationManager.requestLocationUpdates(this);
  }

  protected void onStop() {
    // isActive = false;
    mLocationManager.removeUpdates(this);
    mDbHelper.close();
  }

  public void onLocationChanged(Location loc) {
    // if (!isActive) return;
    mDbHelper.getWritableDatabase().insert(loc);
  }
}
```

(a)



(b)

Figure 3. An example application that may unintentionally keep a database connection open if a location update is enqueued while the application's activity stops (a) and its bad execution trace (b).

Repair To synchronize this example we can modify the `onPostExecute` method to check whether the `Activity` is still in a valid state before dismissing the `ProgressDialog` as illustrated by the comments in Fig. 2. This synchronization relies on the fact that when the framework destroys an `Activity`, it sets a flag to reflect this state change in a field in the `Activity` object.

3.3 Callback Races

The Android framework contains hundreds of different types of events that can be delivered to a running application in non-deterministic order. As a result, even developers knowledgeable with the Android event system may not know all of the ordering constraints.

Consider the example shown in Fig. 3 (a). This example registers a listener to receive location updates from the GPS sensor and writes them into a database. The method `onResume` is used to start location updates when the application becomes visible, and `onStop` is used to remove location updates when the user navigates away. Database manipulation is facilitated by a `SQLiteOpenHelper` class provided by the Android framework. The `onStop` method closes the database and the semantics of `getWritableDatabase` are such, that it either returns an already open database instance or creates a new one.

Intuitively, the `onLocationChanged` callback should be executed only after executing `onResume` and before executing `onStop`, yet this is not always the case. The reason why location updates may arrive after they are stopped is illustrated in Fig. 3 (b). If a sensor enqueues a location update while the `onStop` method is being executed, then after completing the `onStop` method, the application will process the enqueued location update.

Harmful Behavior If location updates come to a stopped activity, the database instance will be reopened and will consume resources even if the user is not interacting with the application.

Data Race Here, a race is reported on the internal field `mConnectionPoolLocked` of the class `SQLiteDatabase`.

Repair To ensure that no location updates are processed after the application is stopped, we suggest adding a guard `isActive` as shown in the comments in Fig. 3 (a).

4. Android Platform Overview

We provide a brief overview of the Android framework focusing on the parts which affect concurrency.

4.1 Event Dispatching

Event dispatching in Android is facilitated by the `Looper` class. While `Looper` objects may be created by the developer, their common use is in the *main* thread of every Android application. As the application starts, the frameworks launches the *main* thread and runs a `Looper` that continuously dispatches events.

Events dispatched by `Looper` are called messages and each message defines a routine to be executed. Once a message routine is started, no other message routines can start until the first one completes. Additionally, the `Looper` class allows for a message to be enqueued both by the application and by the framework, and supports removal of messages that may become obsolete before they are dispatched.

Messages The timing and order of message dispatching is controlled by the message type and parameters as follows:

- $Delayed(delay)$ denotes a message to be dispatched after a specified time elapses. If the time is set to 0 it denotes

that the message should be dispatched directly after the current event finishes (assuming there are no other messages already scheduled before).

- $AtTime(when)$ denotes a message to be dispatched at a specific time.
- $Front$ denotes a message to be dispatched next. For messages of type $Front$ even if the message queue already contains other messages scheduled for execution, the message will be scheduled before them.
- $Idle$ denotes a message to be dispatched when the `Looper` has no other messages to dispatch.

Barriers Messages marked as barriers by the framework are allowed in time critical contexts to enforce additional ordering constraints by delaying dispatch of standard messages enqueued by the application. Examples of messages marked as barriers are animations and user interface manipulations that must be processed at a steady high frame rate. We note that the name *barrier* is adopted from the terminology used inside the Android framework and is unrelated to the barrier semantics typically used in concurrent programming.

Native Messages The `Looper` contains additional lower level mechanism for enqueuing and dispatching system messages from native code. The native messages are used exclusively by the framework for efficient delivery of user input and graphics subsystem messages. We denote these two types of messages as *Input* and *Display* respectively.

4.2 Inter-process Communication (IPC)

The IPC to system services such as `AudioManager` or `WifiManager` is facilitated by a mechanism called `Binder`. Similarly to `Looper`, we can view `Binder` as a special type of message dispatcher, where message enqueueing and dispatching is performed by different processes. The IPC messages can be enqueued in one of two modes – synchronous and asynchronous. Synchronous mode guarantees that the thread performing enqueue blocks until the message is fully processed by the receiver, whereas with asynchronous mode, thread performing enqueue returns immediately.

Message Dispatching Each application processes IPC messages using a designated pool of up to 16 threads, referred to as `Binder` threads. The choice of the `Binder` thread processing an incoming message is non-deterministic and applications should be prepared to handle multiple message dispatches happening at the same time.

5. Capturing Android Asynchrony

We present a thorough formal model which captures the asynchrony arising in the Android platform. The formal model is described in the form of a happens-before (HB) relation between operations – a core building block for many concurrency analyzers (e.g., static may-happen analysis, atomicity analysis). We begin by defining operations necessary to capture the HB where we aim to cleanly capture key details of the platform. We then formalize the notion of an Event and finally define the HB rules over the operations. The definition of HB rules is based on studying framework source code and careful empirical experimentation with the goal to model the ordering of the events as precisely as possible. Our experience across several Android versions is that the HB remains quite stable (i.e., only one change in three years).

Message Type(s)	Field	Description
IPC	$sync$	execution mode
$Delayed$	$delay$	time before dispatch
$Front, AtTime, Delayed$	$barrier$	execution mode

Table 1. Description of *enqueue* operations.

5.1 Defining Operations and Events

Operations defined here are an abstraction of the Android platform where different Android APIs can be mapped to the same operation.

Each operation $op \in Op$ is a structure with the following fields:

$$\langle event, pid, tid, mid, type, dispatcher, \\ dispatcher_{type}, delay, sync, barrier \rangle$$

Here, $event$ denotes the unique identifier of the event as part of which the operation executes, pid is the process identifier, tid is the thread identifier, mid is an unique identifier assigned to each message, $type$ ranges over the values $\{Delayed, Front, Idle, AtTime, Input, Display, IPC\}$, $dispatcher$ denotes the unique identifier of the message dispatcher, $dispatcher_{type}$ ranges over $\{$`Looper`, `Binder`$\}$, $delay$ is a natural number and $sync$ and $barrier$ are booleans.

Operations typically make use of only a subset of fields (discussed below). The operations are divided into three kinds, discussed next.

Message Dispatch The following set of operations provide an abstraction over the various types of message dispatching mechanisms found in Android including both `Looper` and `Binder` described in Section 4.

- *enqueue(mid, type)* denotes a message with $type$ and a unique identifier mid scheduled for execution. For all messages the unique message dispatcher is denoted as $enqueue(mid).dispatcher$. The meaning of the other fields used by this operation is summarized in Table 1: one can see that different message types typically go in pair with different fields.
- *blocking_enqueue_end(mid)* denotes that the blocking *enqueue* operation has finished the message dispatch.

  The only message dispatcher that currently supports blocking *enqueue* operations is `Binder` used for IPC.

- *remove(mid)* denotes that a previously enqueued message is removed and will not be dispatched.
- *begin(mid)* and *end(mid)* denotes the beginning and ending of a dispatching message mid.

Thread Specific

- *fork(tid, tid′)* and *join(tid, tid′)* denote that thread tid created a new thread tid', and respectively tid waits until thread tid' finishes execution.
- *thread_init(tid)* and *thread_exit(tid)* denote the first and last operation performed by a thread tid.
- *wait(id)* and *notify(id)* denote thread synchronization using `wait` and `notify` operations with an unique id.

Explicit Synchronization To model explicit synchronization we define three callback operations. Here, *register(c)*, *invoke(c, sync)* and *unregister(c)* denote that a callback c was registered and might be invoked later using an *invoke*. Boolean field $sync$ denotes the execution mode of the invocation which can be either synchronous or asynchronous. The *unregister* operation denotes removal of the callback c.

The happens-before is defined over API calls at the level of dispatcher mechanisms such as `Looper` and `Binder`. An operation α not performed as a result of dispatching a message is not part of an event and thus $\alpha.event = \perp$.

Definition 5.1. Event. An event is a finite sequence of operations: *begin(mid)* $\cdot \ldots \cdot$ *end(mid)* performed as a result of dispatching a message by a dispatcher mechanism.

5.2 Defining the Happens-Before

A program's semantics is defined as a set of traces where a finite trace $\pi = \alpha_0 \cdot \alpha_1 \cdot \cdots \cdot \alpha_n$ is a sequence of operations. We use π_{tid} to obtain a trace where all operations are performed on thread tid. For a trace π, we use $\alpha <_\pi \beta$ to denote that operation α occurs before operation β in π. The happens-before relation $\prec \subseteq Op \times Op$ is a binary relation that is irreflexive and transitive. For convenience we use $\alpha \prec \beta$ instead of $(\alpha, \beta) \in \prec$.

Happens-Before Rules The HB rules are formalized in Fig. 4. What follows is a description of individual rules. All rules are designed such that they introduce an ordering in the form $\alpha \prec \beta$. To avoid clutter, all rules use the implicit condition $\alpha <_\pi \beta$ in their premise.

EVENTOP. Operations performed within the same event are ordered according to the trace order.

LOOPERATOMIC. A pair of events dispatched by the same `Looper` is ordered whenever there is at least one pair of ordered operations between them. Instead of testing all possible pairs of operations between two events, it is sufficient to test whether the *begin* operation of the first event is ordered before the *end* operation of the second event. This rule takes advantage of fact that the events dispatched by the `Looper` dispatcher are executed atomically and are not interruptible.

$\eta \backslash \gamma$	$Delayed$	$AtTime$	$Idle$
$Delayed$	$\eta.delay \leq \gamma.delay$	$false$	$\eta.delay = 0$
$AtTime$	$false$	$false$	$false$
$Front$	$true$	$true$	$true$
$Idle$	$false$	$false$	$true$

Table 2. Conditions for ordering events based on their enqueue messages type defining the function $looper_{ord}(\eta, \gamma)$.

THREADINIT and THREADEXIT. All operations in a thread are ordered after *thread_init* and before *thread_exit*. THREADINIT together with rule THREADFORK is used to order operations performed before *fork* with the operations performed by a thread created as a result of a *fork* operation. THREADEXIT is similarly used together with THREADJOIN.

MSGENQUEUE. Message *enqueue* is ordered before the start of message dispatch denoted by operation *begin*.

MSGREMOVE. Operation *remove* is ordered after all dispatched messages. The intuition behind this rule is that at the time of executing *remove* operation, either the corresponding *enqueue* was already dispatched or it will never be dispatched (since the remove deletes it from the message queue). We note that this ordering is sound only if the corresponding *enqueue* is guaranteed to be executed before the *remove*. Otherwise the message could be dispatched in case *enqueue* is reordered with the *remove*.

MSGBLOCKING. For blocking *enqueue* operations, not only is enqueue before *begin* as defined by MSGENQUEUE, but additionally *end* is ordered before *blocking_enqueue_end*.

MSGBEGIN#1. This rule defines the conditions under which, given two ordered enqueue operations $\eta \prec \gamma$, the resulting dispatched events α and β will also be ordered. First, both messages η and γ must be dispatched by the same `Looper` (i.e., $\eta.dispatcher = \gamma.dispatcher$). The condition $\eta.barrier \vee \neg\gamma.barrier$ captures that the order in which messages are dispatched should not be affected by barriers.

The function $looper_{ord}(\eta, \gamma)$ evaluates to values from Table 2 of row $\eta.type$ and column $\gamma.type$. For $\eta.type = Front$, we know that only another enqueue of type $Front$ can be dispatched before and therefore the whole row evaluates to $true$.

For $\eta.type = AtTime$ we cannot add any HB ordering as the order depends on external factors such as thread scheduling and can change between executions. To illustrate why two enqueue operations of type $AtTime$ can not be ordered, consider a scenario where initially `x = 0` and we perform two operations `postAtTime(task_1, x+100)` and `postAtTime(task_2,x)` inside of the same event. Depending on time t between the invocation of these two

$$\frac{\alpha.event \neq \bot \qquad \alpha.event = \beta.event \qquad \alpha <_\pi \beta}{\alpha \prec \beta} \text{ (EventOp)}$$

$$\frac{\alpha = register(c) \qquad \beta = invoke(c, \_)}{\alpha \prec \beta} \text{ (CallbackReg\#1)}$$

$$\frac{\alpha = register(c) \qquad \beta = unregister(c)}{\alpha \prec \beta} \text{ (CallbackReg\#2)}$$

$$\frac{\alpha = invoke(c, \_) \qquad \beta = unregister(c)}{\alpha \prec \beta} \text{ (CallbackUnreg)}$$

$$\frac{\alpha, \beta = invoke(c, sync) \wedge \alpha <_\pi \beta}{\alpha \prec \beta} \text{ (CallbackInv)}$$

$$\frac{\begin{array}{c}\alpha = end(mid) \qquad \eta = enqueue(mid) \\ \beta = begin(mid') \qquad \gamma = enqueue(mid') \qquad \eta \prec \gamma \\ looper_{ord}(\eta, \gamma) \qquad \eta.dispatcher = \gamma.dispatcher \\ \eta.type \in \{Delayed, AtTime, Front, Idle\} \\ \gamma.type \in \{Delayed, AtTime, Idle\} \\ \eta.barrier \vee \neg\gamma.barrier\end{array}}{\alpha \prec \beta} \text{ (MsgBegin\#1)}$$

$$\frac{\begin{array}{c}\alpha = end(mid') \\ \gamma = enqueue(mid') \qquad \beta = begin(mid) \\ \eta = enqueue(mid) \qquad \eta \prec \gamma \prec \beta \\ \eta.dispatcher = \gamma.dispatcher \qquad \gamma.type = Front \\ \eta.type \in \{Delayed, AtTime, Front, Idle\} \\ \gamma.barrier \vee \neg\eta.barrier\end{array}}{\alpha \prec \beta} \text{ (MsgBegin\#2)}$$

$$\frac{\begin{array}{c}\eta = begin(mid) \qquad \alpha = end(mid) \\ \beta = begin(mid') \qquad \gamma = end(mid') \\ \eta \prec \gamma \qquad \alpha.dispatcher = \beta.dispatcher \\ \alpha.dispatcher_{type} = \texttt{Looper}\end{array}}{\alpha \prec \beta} \text{ (LooperAtomic)}$$

$$\frac{\alpha = enqueue(mid) \qquad \beta = begin(mid)}{\alpha \prec \beta} \text{ (MsgEnqueue)}$$

$$\frac{\alpha = fork(\_, tid) \qquad \beta = thread\_init(tid)}{\alpha \prec \beta} \text{ (ThreadFork)}$$

$$\frac{\alpha = thread\_exit(tid) \qquad \beta = join(\_, tid)}{\alpha \prec \beta} \text{ (ThreadJoin)}$$

$$\frac{\alpha = thread\_init(tid) \qquad \beta \in \pi_{tid} \setminus \alpha}{\alpha \prec \beta} \text{ (ThreadInit)}$$

$$\frac{\beta = thread\_exit(tid) \qquad \alpha \in \pi_{tid} \setminus \beta}{\alpha \prec \beta} \text{ (ThreadExit)}$$

$$\frac{\alpha = notify(id) \qquad \beta = wait(id)}{\alpha \prec \beta} \text{ (NotifyWait)}$$

$$\frac{\begin{array}{c}\alpha = begin(mid) \\ \beta = remove(mid) \qquad \gamma = enqueue(mid) \qquad \gamma \prec \beta\end{array}}{\alpha \prec \beta} \text{ (MsgRemove)}$$

$$\frac{\alpha = end(mid) \qquad \beta = blocking\_enqueue\_end(mid)}{\alpha \prec \beta} \text{ (MsgBlocking)}$$

Figure 4. HB rules defining ordering constraints. All rules use implicit condition $\alpha <_\pi \beta$ in the premise.

operations there are two possible outcomes: i) if $t < 100$, then `task_2` is dispatched first, ii) if $t \geq 100$, then `task_1` is dispatched first.

When both operations are of type $Delayed$, the resulting events can be ordered only if $\eta.delay \leq \gamma.delay$. Otherwise, the ordering depends on the execution time between η and γ in a given trace which is non-deterministic. When the second enqueue is of type $Idle$, we can order the events only in case $delay = 0$. For any greater value of $delay$, it is possible that the $Idle$ message will be dispatched first in case the `Looper` has no messages to dispatch before $delay$ time elapses.

MsgBegin#2. This rule complements rule MsgBegin#1 where the second ordered *enqueue* operation γ is of type $Front$. We know that γ will be dispatched before any previously enqueued operation η that is still waiting to be dispatched. To guarantee that η is still waiting to be dispatched we use condition $\gamma \prec \beta$, where β is the *begin* operation of an event dispatched as a result of enqueueing message η.

Speculative Happens-Before Rules Many system processes that communicate with the application using IPC or native messages include complex synchronization mechanisms and are partly written in native code. Such code can enforce additional orderings on the events of the application. To capture these orderings, we introduce speculative HB rules. The rules are referred to as speculative because they are based on empirical observations of how the Android system behaves by using careful experimental evaluation (as opposed to source code review) of the `Binder` framework. As a result they may introduce more orderings than strictly necessary. We discuss the effects of speculative rules on the concurrency analysis in Section 8.2.

$$\frac{\begin{array}{c}\alpha = end(mid) <_\pi \beta = begin(mid') \\ \alpha.type, \beta.type \in \{Input, Display\} \\ \alpha.type = \beta.type \qquad \alpha.dispatcher = \beta.dispatcher\end{array}}{\alpha \prec \beta} \text{ (Native)}$$

$$\frac{\begin{array}{c}\alpha = end(mid) \qquad \eta = enqueue(mid) \\ \beta = begin(mid') \qquad \gamma = enqueue(mid') \qquad \eta <_\pi \gamma \\ \eta.sync \vee \neg\gamma.sync \qquad \eta.type = \gamma.type = IPC \\ \eta.pid = \gamma.pid \qquad \eta.dispatcher = \gamma.dispatcher\end{array}}{\alpha \prec \beta} \text{ (IpcHandle)}$$

$$\frac{\alpha, \beta \in \pi_{tid} \qquad \alpha <_\pi \beta \qquad \alpha.event = \beta.event = \bot}{\alpha \prec \beta} \text{ (ThreadOp)}$$

$$\frac{\begin{array}{c}\alpha = end(mid) \qquad \eta = enqueue(mid) \\ \beta = begin(mid') \qquad \gamma = enqueue(mid') \qquad \eta \prec \gamma \\ \eta.type = \gamma.type = IPC \qquad \neg\eta.sync \qquad \neg\gamma.sync \\ \eta.tid = \gamma.tid \qquad \eta.dispatcher = \gamma.dispatcher\end{array}}{\alpha \prec \beta} \text{ (IpcAsync)}$$

Figure 5. Speculative HB rules.

NATIVE. All native messages of the same type delivered to the application are ordered according to trace order. This rule assumes that messages of a given type are enqueued by a single process and that the internal logic of this process ensures the messages are always ordered. For native $Input$ messages, this translates to the observation that when the user interacts with the device, the system will deliver both $Input$ messages to the application in the same order as they were performed, which should intuitively be the case.

THREADOP. This rule states that operations in the same thread and outside of events are ordered by trace order.

IPCHANDLE. IPC messages enqueued by the same process to the same dispatcher are ordered by trace order. This rule uses the observation that IPC messages are usually enqueued to a single dispatcher from a single remote process and we assume the remote process is properly synchronized. This rule is important in order to capture the happens-before ordering of the interaction with Android system services such as `AudioManager`, `WifiManager` or `LocationManager`. Such system services run in a separate process and communicate with the user application using the IPC messages.

IPCASYNC. For two ordered IPC messages $\eta \prec \gamma$ enqueued by the same thread to the same *dispatcher*, the resulting events are fully dispatched in the same order they were enqueued (i.e., dispatch of the second message starts only after first finished dispatching). Note that the messages need not be dispatched by the same `Binder` thread. In case $\eta \prec \gamma$ where $\neg\eta.sync$ and $\gamma.sync$ the events cannot be ordered as it is not guaranteed that two enqueues performed in quick succession are not dispatched concurrently.

5.3 Comparison with CAFA and DROIDRACER

Having formally defined the HB rules, we next discuss the differences compared to the HB rules presented in CAFA [6] and DROIDRACER [9]. The HB rules defined in CAFA are based on Android version 4.3, DROIDRACER uses version 4.0, and we use Android version 4.4. Although the framework core is relatively stable among different versions, there are still some changes that have to be accounted for occasionally.

We analyzed versions of Android from 4.0 to 4.4 spanning more than three years (October 2011 - November 2014) and during this time there was a single change to the framework that required adapting the HB rules (introduction of barriers) as well as a single major performance change that required adapting the actual implementation. Additionally, we note that there is a value in supporting newer framework versions as they often include new APIs that can be of significant value for the developers. Table 3 summarizes the differences between the happens-before model presented in our work vs. the two prior works, CAFA and DROIDRACER. We can see that our model is *strictly* more precise in all aspects than the HB models defined in these works:

HB Rule	Our work	CAFA	DROIDRACER
MSGBEGIN	all types	$Delayed, Front$	$Delayed$
MSGREMOVE	✓	✗	✗
barriers	✓	✗	–
IPCASYNC	✓	✓	✓*
MSGBLOCKING	✓	✗	✗
IPCHANDLE	✓	✗	✗

*manually created list of ordered system events

Table 3. Comparison of HB rules with related work. Common rules are omitted to remove clutter.

- *Complete handling of message types.* We handle all available message types (i.e., $Front$, $Delayed$, $AtTime$, $Idle$), compared to supporting only $Delayed$ by DROIDRACER and $Delayed$, $Front$ by CAFA.
- *Considers effect of message removal.* Our work is first to consider the effect of removing messages on the happens-before ordering.
- *Considers effect of barriers.* We also model the effect of barrier message dispatching. Here we note that DROIDRACER does not include barriers as it targets Android version 4.0 whereas barriers were introduced in version 4.1.
- *Models IPC communication*. We formalize the IPC communication provided by the `Binder` framework using the happens-before rules. This in contrast to the informal description provided by CAFA which also does not describe the different effects of synchronous and asynchronous messages. On the other hand, in DROIDRACER, IPC communication is captured via a set of rules (created by a domain expert) which define the ordering between messages sent from the system. We note that such an approach is inherently incomplete as it cannot capture user defined APIs and can also be costly to maintain.

In our work we introduce both, additional true HB orderings due to rules MSGBEGIN and MSGREMOVE (first two lines), *and* remove false HB orderings using the remaining entries. To quantify the effect of a more precise HB definition, we performed experiments where we enabled and disabled rules not found in CAFA and DROIDRACER. We found that on average, we remove 8.5% false HB orderings by considering the effect of barriers, and 61% HB orderings present due to imprecise handling of IPC communication (assuming that all messages from remote processes, except for lifecycle messages, follow trace order). Some of the false HB orderings that we removed even contradict the order of events as seen in the execution trace (this is caused by not considering the effect of barriers).

Finally, in our experiments we even discovered a bug (described in Section 3.3) in the framework which contradicts the official documentation – the documentation states that after a call to `removeUpdates`, the location updates will

no longer occur[2]. Detecting this bug would be impossible if we used the documentation as the specification for ordering framework events, as done by DROIDRACER.

Races Between Threads Even though the focus of our work is on Android specific concurrency errors (i.e., in the main application thread), for completeness, our system also detects races between threads. To detect these races, the only needed information is tracking lock and unlock operations. Just as in CAFA, we use the lock and unlock operations only to check for mutual exclusion using the lockset algorithm (these do not affect the HB relation). Therefore, checking for races between threads can be easily handled as in our approach, the most expensive and difficult part – defining (Section 5) and computing (Section 6) the HB graph – is done for the whole Android system, not just the main thread.

6. Scalable Computation of the HB Graph

A key challenge when designing our analysis system was coming up with a scalable algorithm for building the happens-before (HB) graph from a given execution. Such an algorithm is particularly critical because in the Android setting, the process of race detection (once the HB graph is built) is *not* the performance bottleneck – the expensive part is building the HB graph. In fact, the algorithm for building the HB graph is also the key performance bottleneck of both [6, 9] and is one of the underlying reasons for why these analyzers do not scale in practice.

The reason we need a new algorithm lies in the complexity of the HB rules discussed so far. We believe that once developed for Android, the HB graph construction algorithm will translate to frameworks with similar HB rules (possibly iOS or Windows Phone). It is important to note that we cannot simply reuse techniques developed in other domains such as analysis of web pages (e.g., [12]), because they target a much simpler fork-join happens-before model.

The algorithm we developed handles the full Android model, performs well on real world applications, is orders of magnitude faster than prior work, and is used for both detecting races on the main thread and for detecting races between threads. Our algorithm takes as input an execution trace and the formalized HB rules as described in Section 5, and outputs a HB graph. Once the full HB graph is obtained, we can then reuse existing techniques for detecting races (described in Section 7).

Overview Algorithm 1 provides a high level description of the HB graph construction. The function `BuildHB` takes as input a trace π consisting of a sequence of operations, as well as an initially empty HB graph (with nodes V and edges E). The algorithm then iteratively builds the graph by adding each operation from the trace to the set of nodes (line 3). The operations are processed in the trace order which is guaranteed by sorting the operations at line 2. We note that the sorting is not required in the actual implementation as long as we ensure that the inputs are already sorted. Next, the algorithm finds edges connecting to the newly added node, by checking which HB rules can be applied (using `applicableRules` on line 4). For most of the rules, we efficiently match all operations α that satisfy the rule preconditions and since each rule implies $\alpha \prec \beta$, we add (α, β) to the graph (line 7). Finally, we apply the complex rule LOOPERATOMIC (lines 8-9). This rule orders entire atomic events based on ordering of operations within the events. In this case, we also reprocess part of the input trace (all nodes ordered after the second event), because the added edge may affect other rules (lines 11 and 12).

Algorithm 1 Builds the happens-before graph (V, E) given an execution trace π.

```
1:  function BuildHB(π, V, E)
2:    for β ∈ sortByTraceOrder(π) do
3:      V ← V ∪ {β}
4:      rules ← applicableRules(β, V, E)
5:      for rule ∈ rules do
6:        for all α ∈ match(rule, β, V, E) do
7:          E ← E ∪ {(α, β)}
8:          for all α' ∈ match(LooperAtomic, β, V, E)
9:                and (α'.event, β.event) ∉ E do
10:           E ← E ∪ (α'.event, β.event)
11:           V_o ← {γ ∈ V | β.event ≺ γ}
12:           BuildHB(V_o, V, E)
```

Key Scalability Ingredients To make the algorithm scale, we instantiate the functions in Algorithm 1 as follows:

- *Efficient HB rule matching*: this is done via the `match` function where for a given $rule$ and operation β, the function finds all matching operations α that should be ordered before β. This is discussed in Section 6.1.
- *Evaluating each rule at most once*: this is accomplished by leveraging rule evaluation order. That is, the function `applicableRules` is designed so that it returns a *list* of rules. This enables us to use a simple loop with $rule \in rules$ instead of an expensive fixpoint iteration. Details of this technique are discussed in Section 6.2.
- *Trace optimization*: this technique prevents processing an operation multiple times due to recursive calls to `BuildHB` (discussed in Section 6.3).
- $\mathcal{O}(1)$ *graph connectivity queries*: we provide an efficient procedure for answering whether $(\alpha, \beta) \in E$, while the graph is being built. This is discussed in Section 6.4.

While instantiated for Android, we believe these techniques (or variants) will be of interest in other settings with complex HB rules.

[2] http://developer.android.com/reference/android/location/LocationManager.html#removeUpdates(android.location.LocationListener)

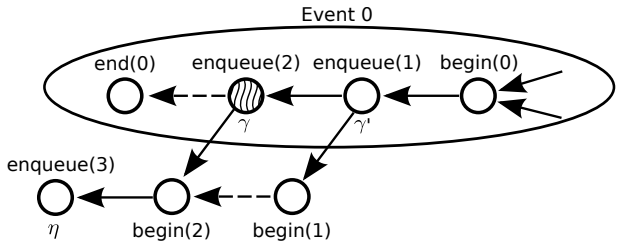


Figure 6. Example of inefficient graph pruning in case of multiple paths to the same node (from $enqueue(3)$ to $begin(0)$) in the happens-before graph.

Performance Benefits In our experiments, the above ingredients were critical to achieving a scalable concurrency analysis system. For example, disabling pruning (Section 6.1) and fast connectivity queries (Section 6.4) resulted in ≈4x and ≈50x slowdowns respectively for 2 minute traces. As the traces get larger, the effect of both optimizations becomes even more significant resulting in slowdowns of ≈7x and ≈140x for 10 minute traces.

6.1 Efficient Rule Matching

Our algorithm evaluates most rules in constant time by caching part of the rule: this allows for fast retrieval of a matching operation α given an operation β. This approach is applied to MSGENQUEUE, MSGREMOVE, MSGBLOCKING and all CALLBACK and THREAD rules, for which our instrumentation guarantees that the corresponding operation α is unique in the trace and is efficiently cached. As an example, consider rule MSGENQUEUE where for an operation $begin(mid)$ we want to find the corresponding operation $enqueue(mid)$ with the same mid. Since the mid is designed to be unique for all $enqueue$ operations (it consists of tuple (tid, id) where the implementation increments id whenever a new message is generated), we can use it to cache the $enqueue$ operation using mid as key. Consequently, when we later encounter operation $begin(mid)$ in the trace, we can easily retrieve the corresponding $enqueue(mid)$ from the cache.

Rules EVENTOP, THREADOP, IPCHANDLE and NATIVE are evaluated in $\mathcal{O}(1)$ by taking advantage of the transitivity of the HB relation. Here, it is sufficient to add an edge only to the latest α in the trace that matches the rule.

For performance reasons, rule LOOPERATOMIC is not evaluated directly. Instead, when other rules introduce an ordering $\alpha \prec \beta$, we perform the following equivalent evaluation. First, if β is not performed in a `Looper` event, the rule premise never evaluates to $true$ and hence we stop evaluation. Second, when both α and β are performed by the same `Looper`, it is sufficient to only check whether their enclosing events can be ordered. Finally, when β is performed by a `Looper` but α is performed either by a different `Looper` or not inside a `Looper`, we need to traverse a potentially linear number of nodes in the graph. This is because we need to find all operations ordered before α that are performed by the same `Looper` as operation β.

Finally, rules MSGBEGIN and IPCASYNC have worst case complexity linear in the number of nodes. The reason is that they require checking all ordered pairs of $enqueue$ operations $\eta \prec \gamma$. Even though worst case is linear, by pruning the graph traversal, we are able to effectively reduce the total number of visited nodes during analysis.

Graph Traversal Pruning Evaluating rule MSGBEGIN and IPCASYNC requires checking all ordered pairs of $enqueue$ operations $\eta \prec \gamma$ (implemented as a depth-first graph search). To improve the worst case linear complexity, we prune the graph search to significantly reduce the total number of visited vertices during analysis as follows:

- For two $enqueue$ operations $\eta \prec \gamma$ of type $Delayed$ we prune when $\eta.delay = \gamma.delay$ and rule MSGBEGIN evaluates to $true$.
- For two $enqueue$ operations $\eta \prec \gamma$ of type IPC we prune when rule IPCASYNC evaluates to $true$.

These conditions reflect the fact that pruning is performed only when we are guaranteed that (by the transitivity of the HB relation) the HB graph already contains all edges that might be added as a result of visiting pruned nodes.

Further, whenever node γ is pruned, we also prune *begin* node of $\gamma.event$. This is important as there might be multiple paths leading to the *begin* node, not all of which might be pruned. However, as long as at least one of them is pruned, evaluating such a path can be safely skipped. An illustrative example of the happens-before graph where this optimization is useful is shown in Fig. 6. The edges in the graph denote happens-before relationships (e.g., operation $begin(0)$ is ordered before operation $enqueue(1)$) and we perform search for all pairs of $enqueue$ operations $\eta \prec \gamma$. Whenever such a pair is found, we evaluate the corresponding happens-before rule (one of MSGBEGIN or IPCASYNC) and if possible, perform pruning. The algorithm performs depth-first graph search and finds an ordered pair of $enqueue$ operations $\eta \prec \gamma$ which can be pruned. However, as the algorithm continues, another ordered pair of $enqueue$ operations $\eta \prec \gamma'$ is found. Assume however that this pair does not satisfy the conditions for pruning, and therefore the depth-first search continues visiting node $begin(0)$ and all of its parents. By pruning not only the node η but also its corresponding $begin$ operation (in this case $begin(0)$), we can effectively prevent unnecessary graph search in this scenario.

6.2 Rule Evaluation Order

To ensure efficiency, we define evaluation order of the HB rules for every operation in the trace such that a rule is evaluated *exactly once*. That is, evaluating each rule once must produce the same result as evaluating all rules until a fixed point. Because the conclusion of all rules has the form

$\alpha \prec \beta$, only rules that use HB ordering in their premise can be affected by applying the conclusion of another rule. There are two rules that make use of HB ordering in their premise – MSGBEGIN and IPCASYNC. Note that for any operation β, at most one of them can be applied. This is because both rules are used to order operation $\beta = begin$ based on the type of its *enqueue* operation. Hence, we can evaluate all rules in any order except these two, which are evaluated last (in any order).

6.3 Processing Operations Only Once

The previous optimization guarantees that during operation processing, each rule is applied at most once. However, an operation may be processed multiple times due to the rule LOOPERATOMIC (recursive call in Algorithm 1). Consider the trace $e_1 \cdot e_2$ where the two events are:

$$e_1 : \quad begin(1)^1 \cdot enqueue(3, \dots)^2 \cdot register(c)^3 \cdot end(1)^4$$
$$e_2 : \quad begin(2)^5 \cdot enqueue(4, \dots)^6 \cdot invoke(c)^7 \cdot end(2)^8$$

Here, the order in which operations are processed is denoted by the superscripts. Consider that the first five operations were already processed and we are about to process $enqueue(4)$. At this point, rule MSGBEGIN is evaluated by finding all $enqueue$ operations ordered before $enqueue(4)$ and checking whether the resulting events can be ordered. Since events e_1 and e_2 are not yet ordered, no such $enqueue$ operation is found. As we process the next operation $invoke(c)$, rule CALLBACKREG orders $register(c)$ and $invoke(c)$ after which rule LOOPERATOMIC is used to order the events by introducing the edge $end(1) \prec begin(2)$. However, this introduces an ordering between $enqueue(3)$ and $enqueue(4)$, thus affecting the result of processing $enqueue(4)$ in the previous step. As a result, we need to evaluate $enqueue(4)$ again.

To prevent frequent occurrence of such cases (i.e., where same pair of operations needs to be evaluated several times), the HB algorithm moves operations – in this case $invoke$ – that are guaranteed to trigger the rule LOOPERATOMIC to the beginning of the event. This is sound if both such operations are from the same dispatcher. In the above example, this results in moving the $invoke(c)$ operation right after $begin(2)$, hence establishing an order between events e_1 and e_2 before $enqueue(4)$ is processed.

6.4 Fast Online Connectivity Queries

To keep the HB graph sparse (which is crucial for effective pruning) and to efficiently evaluate rules MSGBEGIN and LOOPERATOMIC, we need to efficiently check whether two nodes are reachable in the graph. For this purpose, we modified the chain-decomposition graph connectivity algorithm used for offline race detection in [12] to be usable in our online graph construction setting. Note that we use the terms offline and online to denote the design of the algorithm, that is, whether it requires the complete trace (offline) to run or whether it can also be used incrementally as the trace is being built.

The idea of chain decomposition is to assign every node in a graph to a chain, such that there is a path from a node to its successor in the chain. Then, a vector clock is allocated to each node in the graph and a connectivity query between any two nodes can be performed in constant time [12]. Each vector clock requires memory proportional to the number of chains. To make the chain assignment procedure work in an online setting, we used the fact that we add nodes to the graph in the order they appear in the trace. Since we apply many of the rules directly after adding a node, we greedily try to assign each node to a chain formed by a predecessor node in these directly applied HB rules (Algorithm 1, line 7), or to a new chain if no such chain is found.

Finally, for some of the rules, we must be able to add edges between a pair of (possibly internal) nodes α and β in the graph. To add edge from α to β, we join the vector clocks of each of the nodes β', for which $\beta \prec \beta'$, with the vector clock of α.

Memory Efficiency Memory efficiency of Algorithm 1 is dominated by the memory requirements of the vector clocks. This is due to the fact that vector clocks require $\mathcal{O}(nc)$ memory, where c is number of chains and n is the number of events. In our experiments, the number of chains is less than 5k for all analyzed traces (up to 10 minutes of app exploration, with up to 400k nodes). Many chains are generated when processing the system startup where for example a 10 minute Facebook trace (4577 chains, 380k nodes) has only slightly more than twice the chains of a 2 minute trace (2075 chains, 100k nodes) which demonstrates that the analysis scales well as the number of nodes increases.

7. Implementation

This section describes key implementation aspects of our system. The individual parts and the flow between them is shown in Fig. 7. These parts include: (i) a modified Android framework, (ii) an algorithm that builds the HB graph from a given trace, a race detector analyzing the HB graph, and (iii) an interactive web-based race explorer which allows one to easily explore the reported races and the HB graph.

7.1 Android Instrumentation

Our instrumentation extends the Android framework version 4.4. We created a custom system image that can be flashed onto a real device or used by a standard Android emulator. Instrumenting the framework instead of only the application is essential as many operations necessary to establish HB orderings are generated internally in the framework. The instrumentation is performed at the lowest possible API level (e.g., instrumenting a single abstract `View` class instead of all concrete UI components) and required modification of around 40 different Android API classes.

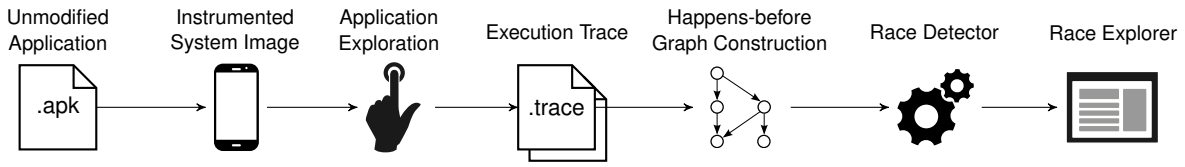


Figure 7. Overview of the analysis process.

Java Memory Locations Android applications and most of the Android framework are written in Java, where memory locations are object and class fields. The instrumentation is performed by modifying the portable interpreter found in the Dalvik VM (i.e., reads/writes to object and class fields are translated to *read* and *write* operations). Our modification is based on the portable interpreter, which is written entirely in C and is expected to run on a broad range of hardware platforms. To keep our instrumentation active at all times, we disabled the JIT compiler.

Logical Memory Locations We define *logical memory locations* to explicitly allow a suitable abstraction of the shared or external resources. For example, we instrumented two `SharedPreferences` methods `get(key,value)` and `put(key,value)` and translated them to corresponding *reads* and *writes* to logical memory location `key`.

To handle long-running programs, we do not record reads and writes of newly allocated objects and final static fields as well as reads and writes inside Java methods invoked from native code. Additionally, inside a single event, we only record the first reads and writes to the same memory location. We note that this optimization is enabled only when analyzing the main thread of the application as otherwise we record all reads and writes.

Message Dispatcher Operations We instrumented three message dispatchers – `Binder`, `Executor` and both Java and native `Looper`. For all message dispatchers the corresponding message container was extended to facilitate storing a unique mid assigned to a message upon its creation.

Callback Operations For intraprocess Android APIs, the instrumentation is done manually and includes user interface components, sensors and utility class listeners such as `View.OnClickListener`. For interprocess APIs, we take advantage of the fact that interprocess listeners are usually associated with a low level native resource used to facilitate the interprocess communication and the listener invocation. As a result, instead of instrumenting all the individual places where such listeners are created, we instrument the low level native resources for `Binder` and the native `Looper` classes.

Lock and Unlock Operations To keep track of lock and unlock operations, we instrumented the Dalvik VM and the Java package `java.util.concurrent.locks` which defines built-in primitives such as `ReentrantLock`. The instrumentation of Dalvik VM handles the Java keyword `synchronized` by instrumenting the corresponding opcodes `monitorenter` and `monitorexit` inside the interpreter. We note that no special handling is required for the `java.util.concurrent.atomic` package which defines set of lock-free thread-safe atomic classes as they all use native methods (e.g., `compareAndSet` or `getAndIncrement`) that do not access Java memory locations.

7.2 Race Detection

Once the HB graph is obtained, we use the modified graph connectivity algorithm from [12] to find races as described in Section 6.4. The algorithm first decomposes the graph into c chains of totally ordered nodes and then assign vector clocks of width c to all nodes. For a graph of n nodes and c chains, it requires $O(nc)$ space and $O(1)$ time per connectivity query.

7.3 Dealing with Benign Races

A key challenge with race detectors is that they often report too many races, beyond what is reasonable to expect from a developer to inspect manually. To address this issue, we employ the following three techniques.

7.3.1 Race Coverage

The first approach we use to filter races is the concept of coverage introduced in the EVENTRACER work [12]. Informally, covered races are races which are not guaranteed to occur due to ad-hoc synchronization by races on other memory locations. Conversely, *uncovered races* are *guaranteed* to be real races.

We note that race coverage guarantees that if a race (c,d) covers race (a,b), it also covers any other race (e,f) where e and f are not recorded due to not being the first read or write in an event for a given variable. This is because the definition of race coverage does not depend on the program's control flow or on the values being read/written. Further, we note that race coverage is only applicable when we are analysing the main thread of the application since its definition assumes the presence of atomic events.

Covering Races in Android Based on extensive experiments with coverage in Android, we reached a conclusion that we need a finer definition of coverage where we can control *which* races can cover which other races. Hence, we ensure that only memory locations in user space and a set of whitelisted framework variables such as the `mDestroyed` field in a `Activity` class may act as synchronization. We believe that the set of whitelisted variables we provided is generic for Android and is unlikely to change in future.

Operation Type	Count$^{(median/max)}$	Size$^{(median/max)}$
Happens-before	$59 / 62 \cdot 10^4$	13.45 / 14.13 MB
Memory Location	$514 / 533 \cdot 10^4$	85.52 / 90.08 MB
Debug Info	$1933 / 2124 \cdot 10^4$	97.1 / 116.04 MB

Table 4. Trace statistics for 6 minute application execution.

7.3.2 Race Filtering

We use several techniques to remove common classes of benign races by taking into account the context of the race operations. One such technique was previously presented by Raychev et al. [12]: races updating the same value, races with no non-local reads and lazy initialization. Additionally, we designed the following Android specific filters:

Races Entirely in the Framework We assume that all operations in the framework are properly synchronized. Thus, we mark as benign all races, for which the operations and their entire call context is in the Android framework. Note that this *does not* include the operations that are in the framework, but were called from user code (e.g., see Section 3.2).

Observably Commutative Operations We constructed a list of Android specific APIs and locations where we know that the operations commute. One such class of APIs are operations that update the UI components or interact with the graphics subsystem. We mark as benign all races where at least one of the operations belongs to such an API.

7.3.3 Race Grouping

Finally, we group multiple redundant races into single reports by event source such as *Input*, *Display*, *IPC* or *Thread*, as well as by the calling context of the race operations (i.e., the stack trace). Additionally, we split the reported groups into two categories: (i) **User vs. User race groups** where both accesses to the memory location are performed directly in the user code, and (ii) **User vs. Framework race groups** where one of the accesses is performed in the framework.

8. Evaluation

This section discusses our extensive experimental evaluation. We tested the following experimental hypotheses:

- *Analysis Performance:* The instrumentation of the Android framework does not significantly degrade the performance of the application.
- *Analysis Performance:* The algorithms scale to analyzing real-world application interactions in feasible time.
- *Analysis Usability:* Filtering and grouping techniques are effective in reducing the number of benign races, and the system is able to find harmful races.

We used a Nexus 10 device with a custom instrumented system image to collect program traces. The processing of traces was performed on a machine with 2.10GHz Intel Core i7-4600U CPU, 16GB of memory running Ubuntu 13.04.

Metric	Facebook	Dropbox	Twitter
2 minutes			
Application Events	6823	3761	5763
Runtime in *seconds*	28.9	19.8	20.2
Memory Usage in *GB*	1.93	1.79	1.06
10 minutes			
Application Events	50135	16261	19878
Runtime in *seconds*	126.7	77.9	91.4
Memory Usage in *GB*	10.45	6.38	6.45

Table 5. Performance metrics of end-to-end analysis of the main application thread.

We evaluated our system on 354 popular Android application downloaded from Google Play Store[3]. Each application was exercised with 1000 user events generated by AndroidMonkey[4] using the following command:

```
adb shell monkey -s 42 --throttle 60 -v 1000
```

This resulted on average in 60 seconds of active application usage. Our system was able to analyze the collected traces of each application on average in 24 seconds.

On average, our system discovered $3,093$ races per application main thread, but based on our filtering and grouping, we reduced this to only 4 reports in the user code, and 19 reports in APIs called from user code. Further, in 11% and 37% of the applications there were no races reported for user vs. user and framework groups respectively.

8.1 Performance

To test the efficiency and scalability of our system, we used three of the most popular, yet very complex, applications – Facebook, Dropbox and Twitter.

Overhead Our instrumentation is based on the portable interpreter of the Dalvik VM and does not support JIT compilation or use some of the assembly routines for particular platforms. Nevertheless, in practice the system feels nimble and responsive and such performance optimizations were not needed. The runtime overhead incurred by the instrumentation of reads and writes together with collection of debug information such as stack traces is $\approx 300\%$ (the overhead required to build the HB graph is negligible). We are able to achieve such small overheads because of efficient implementation coupled with the fact that part of the *main* thread's execution is spent in native code for which memory location instrumentation is disabled.

Trace Size Trace files statistics are shown in Table 4. The majority of recorded operations are due to collecting additional debug information used to improve usability of reported races. Because we instrument the framework, we need to handle on average $10 - 20 \times$ more locations than

[3] `https://play.google.com/apps`

[4] `developer.android.com/tools/help/monkey.html`

Metric \ Application	OI File Manager	NPR News	ATimeTracker	Aard Dict	AnyMemo	Flick-Uploader	aSQLiteManager	FeedEx
Number of uncovered races on main application thread	2251	455	685	1201	925	1349	1755	2004
User vs. User race groups	**4**	**4**	**0**	**6**	**3**	**12**	**1**	**4**
Harmful	3	1	0	2	2	2	0	1
Commutative	1	0	0	0	1	2	1	1
Synchronization	0	1	0	1	0	5	0	2
Harmless	0	2	0	3	0	3	0	0
User vs. Framework race groups	**9**	**7**	**6**	**7**	**13**	**9**	**6**	**12**
Harmful	6	1	1	0	3	0	2	1
Commutative	0	2	0	1	4	1	1	10
Synchronization	0	2	0	1	3	3	0	0
Harmless	3	2	5	5	3	5	3	1

Table 6. Races and groups reported by our tool for the *main application thread* after analyzing traces of 8 selected applications.

prior work (16K vs 646K for Twitter, 50K vs 800K for Facebook). Note that the number of nodes in the resulting HB graph is less that the number of HB operations recorded, since it is unnecessary for each operation to be a new node.

End-to-End Analysis Performance metrics relevant for end-to-end analysis of the main application thread are shown in Table 5. Given the same exploration time, the processing time is relatively stable among various apps. This is because the processing time is mainly spent in constructing the HB graph build for the entire Android system which is dominated by the framework and not the analyzed application. Further, the number of total events in the HB graph for the whole system is usually $3-6\times$ larger than the number of application events. We can efficiently analyze traces of up to 10 minutes of interaction even for large apps. Analyzing such interactions is practically infeasible for other works [6, 9] as they already take hours for much shorter interactions. The time it takes to run the offline race detector and apply the filters and grouping we developed is negligible and is less than a second for all analyzed traces.

The runtime of the end-to-end analysis including all application threads is 25% to 200% slower than analyzing only the main application thread. Even though the HB graph graph construction runtime is exactly the same as when analyzing only the main application thread (in both cases the HB graph is computed for the whole application), significantly larger amount of memory locations and debugging information (e.g., call stack traces) needs to be processed. However, this overhead is only linear in the size of the trace and does not affect the overall scalability of the analysis.

8.2 Usability

We evaluated the techniques for dealing with benign races (discussed in Section 7.3). Towards that, we evaluated several popular open source applications. This choice was done to allow for manual inspection of the reported races and assess their harm from the source code. For all experiments that detect data races on the main thread we always used the set of races computed using the covering scheme, filtering and grouping as discussed in Section 7.3. For experiments that detect traditional data races between threads, race coverage does not apply. Table 6 shows detailed results for the number of reported race groups in the selected applications for the main application thread. Based on our filters, our analysis system reported only 3.5 and 9 groups on average with high and normal priority respectively. The high priority groups contain races in the user code of the application, and the normal priority groups contain races in framework APIs called from the user code. Reports in user code contain on average 2.5 races while reports in user vs framework contain on average 22 races, illustrating the effectiveness of the race grouping technique. These numbers do not include races not shown due to the usage of race coverage.

We manually inspected each of the reports and classified them into one of four classes – harmful, commutative, synchronization and harmless. What follows is a short description of each class. We note that in our reports, we did not find any false positives due to deficiencies in our HB rules or instrumentation.

Harmful Races Upon manual inspection we confirmed that 11 out of 34 race groups in the user code and 14 out of 69 race groups in framework APIs called by user code can lead to 15 different harmful errors. The number of harmful races is usually higher than the number of harmful bugs as a single bug can manifest itself as different races. The reported races provide a range of different ways to trigger them (e.g., standard usage, pressing the back button, orientation change or external system event). We note that many of the reported races do not result in an exception but instead change the intended semantics of the application in a way that can go unnoticed by developers, but may plague user experience. Next, we discuss some of the harmful behaviors, in addition to the ones described in Section 3:

Metric \ Application	OI File Manager	NPR News	ATimeTracker	Aard Dict	AnyMemo	Flick-Uploader	aSQLiteManager	FeedEx
Number of all races found between application threads	142	947	0	1	1292	327	2	93
User vs. User race groups	**11**	**50**	**0**	**0**	**62**	**48**	**2**	**39**
Not inside atomic region	11	50	0	1	55	21	2	29
Inside atomic region	0	0	0	0	7	17	0	10
User vs. Framework race groups	**2**	**101**	**0**	**5**	**4**	**75**	**0**	**5**
Not inside atomic region	2	101	0	5	4	42	0	5
Inside atomic region	0	0	0	0	0	33	0	0

Table 7. Races reported by our tool *between application threads* after analyzing traces of 8 selected applications.

- `OI File Manager`: A harmful observed race is that while the contents of a current directory are loaded asynchronously in the background, the user navigates to a previous directory either by using UI components provided by the application or by pressing the back button on the device. However, in this scenario, the asynchronous task is not properly canceled and when it finishes, *contents of a wrong directory are displayed on the screen*.
- `FeedEx`: Here, articles are shown to the user one at a time using the `PageViewer` component. For usability, the `PageViewer` component loads an asynchronously selected article and its neighbors. Simultaneously, the application keeps a user preference which controls the article appearance on screen in the `mPreferFullText` field, which might be changed when the article finishes loading. Unfortunately, depending on the order in which articles finish loading, *the same set of articles might be shown differently between individual executions*.
- `NPR News`: Here, a harmful race occurs in two unordered application generated events. The application shows a `ListView` containing news articles and allows loading more articles asynchronously on demand by clicking a button at the bottom of the list. As a result, *it is possible to show the same article multiple times in the same list*.

Commutative and Synchronization Races Given the size of the framework, there are races which originate in the framework not properly marked by the filters we developed. An interesting example occurs between commutative operations on collections such as `ArrayList`, where `append` and `get` of an existing element commute, but we may observe a race (e.g., on field `mSize`). To handle such cases, we plan to adopt the commutativity analysis technique from [4].

Harmless Races We found two main types of harmless races: benign and inactive. Benign races are such that both event orderings are correctly handled. An example is a race between an event updating a `ListView` and a click on a list element. While both orders lead to different results, each is correct since the user always interacts with the most up-to-date items in `ListView`. Alternatively, inactive races do not lead to harmful behaviors in the current version of the application, but are an indicator of incorrectly ordered events. An example is a race that associates incorrect data with a UI control, but the invalid UI is not shown to the user. Such bugs are of interest, but are likely to be of lower priority.

Effect of Speculative Rules None of the speculative rules caused missing an actual race in our experiments. However, by disabling speculative rules we observed many more false races ($\approx 20\%$ more races for Facebook), mostly as a result of not ordering user input events.

To the best of our knowledge, rules NATIVE and IPCHANDLE never result in false negatives (i.e., not reporting an actual race). They are listed as speculative only because, even after careful study of the source code, we were not able to verify that the alternative ordering could not happen (it should be noted that the relevant implementation that affects happens-before is in the Linux kernel). This is in contrast to the rules in Fig. 4 which are all designed based on guarantees given by the actual implementation.

On the other hand, rule THREADOP can result in false positives in case the developer uses a custom message queue implementation instead of the one provided by the standard library. Thus, our tool allows to select whether this rule should be used with user created threads. However, none of the 8 manually analyzed applications contained a custom message queue and therefore disabling this rule for user space threads did not change the number of reported races.

Races Between Threads To evaluate races found between application threads we performed analysis on the same set of applications as in the previous step, as shown in Table 7. The applications `NPR News` and `AnyMemo` take advantage of the threads heavily to offload computation from the main thread, which also leads to a high number of reported races. This is worsened by the fact that these applications use pure Java threads instead of the abstractions provided by the Android framework (e.g., `AsyncTask`).

We analysed all these reports manually and discovered several harmful races and patterns that developers did not implement correctly. First, the developers often used shared

variables for notification purposes while forgetting that the values are not necessarily guaranteed to be updated atomically. In this setting, usually one thread only reads from a given variable and another thread only writes. An example of this harmful race can be found in `aSQLiteManager`. There, a thread that exports a database, records its progress into a shared variable `myProgress` and the main thread reads from this variable and displays the progress to the user. In such cases, an easy fix is to use atomic primitive types provided in the Android library. Second, we notice that the usage of atomic regions (i.e., using explicit locks or `synchronized` keyword) is quite rare. This often leads to concurrent usage of data structures and objects which are not thread safe. We believe that this is partially caused by the fact that the Android framework often hides the internal structures behind high level APIs. As an example, queries on `SQLiteDatabase` are not thread safe but queries on `ContentProvider` are thread safe. Similarly using `SharedPreferences`, as done in `OpenFileManager`, is not thread safe since the `SharedPreferences` class is internally backed by a non-thread safe hashmap.

Finally, we note that some of the races are benign as they are on objects recycled by the framework (such as `Parcel` or `TypeValue`). For these objects a common practice is to ask the framework for the object of the given type instead of creating a new instance manually. However, if the framework returns a previously used object, it is very likely that a race will be reported when that object is accessed. To address such scenarios, a simple filter can be added to mark these races as object reuse and to not report them.

9. Conclusion

We presented the first scalable analysis system for finding data races in Android applications. Our system is based on several key contributions: (i) fast algorithms for building and querying the happens-before graph capable of handling complex happens-before rules, (ii) a precise formal happens-before model of Android concurrency, and (iii) a thorough experimental evaluation illustrating that our techniques for race filtering and grouping result in a practically feasible number of reports (decrease the amount of false positives by orders of magnitude). No existing work deals comprehensively with these issues, yet these are important for analysis of complex concurrent systems.

We performed an extensive evaluation of our system on 354 real-world Android applications. Our results indicate that the system is practically useful and scales to realistic interactions with complex applications (e.g., Facebook). The system was successfully used to find 15 harmful bugs of diverse kinds in 8 open-source applications from Google Play Store, while (critically) reporting few false positives. Based on these results, we believe that our system represents a valuable tool for Android developers.

References

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM Conference on Automated Software Engineering*, ASE '12.

[2] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12.

[3] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN Conference on Object Oriented Programming Systems Languages Applications*, OOPSLA '13.

[4] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen. Commutativity race detection. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14.

[5] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09.

[6] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14.

[7] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA '13.

[8] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '13.

[9] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14.

[10] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev. Sdnracer: Detecting concurrency violations in software-defined networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15.

[11] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12.

[12] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13.

[13] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based gui testing of an android application. In *Proceedings of the 2011 4th IEEE International Conference on Software Testing, Verification and Validation*, ICST '11.