Automatically Verifying and Reproducing Event-Based Races in Android Apps

Yongjian Hu
University of California
Riverside, CA 92521, USA
yhu009@cs.ucr.edu

Iulian Neamtiu
New Jersey Institute of
Technology, NJ 07102, USA
ineamtiu@njit.edu

Arash Alavi
University of California
Riverside, CA 92521, USA
aalav003@cs.ucr.edu

ABSTRACT

Concurrency has been a perpetual problem in Android apps, mainly due to event-based races. Several event-based race detectors have been proposed, but they produce false positives, cannot reproduce races, and cannot distinguish between benign and harmful races. To address these issues, we introduce a race verification and reproduction approach named ERVA. Given a race report produced by a race detector, ERVA uses event dependency graphs, event flipping, and replay to verify the race and determine whether it is a false positive, or a true positive; for true positives, ERVA uses state comparison to distinguish benign races from harmful races. ERVA automatically produces an event schedule that can be used to deterministically reproduce the race, so developers can fix it. Experiments on 16 apps indicate that only 3% of the races reported by race detectors are harmful, and that ERVA can verify an app in 20 minutes on average.

CCS Concepts

•Human-centered computing → Smartphones; •Software and its engineering → Software defect analysis;

Keywords

Google Android; Event-based races; Race verification; Happens-before relation; Event flipping

1. INTRODUCTION

Concurrency has been a perpetual problem in Android apps since the platform's inception in 2007 [32]. The root of the problem is Android's event-based programming model, where lack of synchronization between events leads to *event-driven races*. To find such races, several detectors have recently been developed, e.g., DroidRacer [21], CAFA [18], and EventRacer Android [7] (for brevity, we will refer to the latter as simply "EventRacer" since the scope of this paper is Android apps). They operate in a similar fashion. First, they define a set of *Happens Before* (HB) rules for Android's event-driven model. Then, they instrument the Android platform to collect runtime traces and run the app under this instrumentation; the collected traces contain event begin/end/posting and memory read/write information. Finally, they analyze the trace according to the HB model graph. If there exist read/write or write/write operations on the same memory location and these operations are not ordered by HB, the tools report an event-driven race.

However, these tools have several drawbacks: (1) they are prone to false positives, (2) they cannot verify the effect of the race, e.g., is it a benign or harmful race, and (3) they do not give developers a way to reproduce the race. We now discuss these drawbacks and how our approach helps address them.

False positives. Most Android apps use ad-hoc synchronization to protect shared variable access across asynchronous events. Therefore, race detectors can improve their precision by identifying a broad range of synchronization operations, to avoid reporting safe/synchronized access as races. In our experience, even the most precise race detector currently available, EventRacer, is still prone to false positives. EventRacer attempts to filter out false positives by applying a technique called "race coverage" [24] which was previous used for event-driven races in web applications. While race coverage can greatly reduce the false positives rate, it still fails to identify certain categories (types) of false positives. In Section 3 we describe these categories.

Harmful vs. benign races. The second problem with current tools is that for true positives – accesses unprotected by synchronization – they fail to distinguish between benign and harmful races. Our study shows that only a very small portion of reported races are harmful. Previous studies have reached similar conclusions for desktop applications [26]. Since analyzing races requires substantial human effort, an approach with a high rate of false positives or benign races is less likely to be adopted by developers, as it is a high-investment/low-return activity. Thus we argue that we need an automatic race verification tool that can distinguish between benign and harmful races. In Section 3 we define benign races, harmful races, and false positives.

Reproducing races. Current Android race detectors do not help reproduce a race, hence developers have to manually adjust synchronization operations or timing (e.g., set breakpoints via debugger) until the race is reproduced – this is time-consuming and not guaranteed to succeed (in contrast, we provide deterministic replay, which guarantees that a race can be reproduced so the developers can help find and fix the cause of the race).

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany

http://dx.doi.org/10.1145/2931037.2931069

Our approach. To address these issues, we introduce ERVA (Event-race Reproducer and Verifier for Android)[1], an automated approach and tool for verifying and reproducing event-based races in Android apps. ERVA, described in detail in Section 4, takes as input a report of a potential race, categorizes the race, and uses a suite of techniques to categorize the race into three categories. First, if the race is a false positive, it is reported as such. If the race can be confirmed, it is classified as benign or harmful. To support this classification, we introduce *event dependency graphs (EDG)* and a novel definition of benign vs. harmful races in Android apps based on state comparison. If the race is harmful, ERVA automatically produces an event schedule that can be used to deterministically reproduce the race, so the developers can study the race, understand its cause, and fix it.

ERVA does not require access to the app source code, but rather relies on dynamic tracking of happens-before (HB) relationships, schedule replay, and "event flipping". Given an app, ERVA proceeds in two stages. In the first stage, ERVA runs the app in the EventRacer [7] race detector to obtain a set of candidate races (pairs of race events). While the app is running in EventRacer, ERVA records replay information (e.g., UI events, input stream, sensor streams) and synchronization information (e.g., begin/end of thread and synchronization actions, event posting, etc); this information is used in later phases. For each candidate race from the report, ERVA's post-run analysis will confirm whether the candidate is indeed a race, to distinguish between false positives and true positives.

The second stage examines the true positives to further distinguish between benign and harmful races. ERVA replays executions multiple times using the inputs recorded in the first stage, this time instrumenting the app to record app state. In each of these executions, ERVA "flips" – alternates the ordering of – the events to check their side effects, i.e., the effect of flipping on app state (app state includes all the UI view states, shared preferences, file, database, network traffic). If the flipping has no side effect, ERVA categorizes the race as benign, otherwise it is declared as harmful. Since ERVA employs replay, developers have the opportunity to replay the app with those inputs and event schedules that lead to harmful races, to facilitate finding and fixing the cause of the race.

In Section 5 we present a study and evaluation of running ERVA on 16 real-world Android apps. The study found that out of the 260 race reports in these apps, only 8 (that is, 3%) are harmful. Running ERVA takes about 20 minutes on average per app, which indicates that it is both effective and efficient at verifying and reproducing races.

In summary, our main contributions are:

1. Event dependency graphs and a definition of harmful vs. benign races for Android.

2. A practical tool, ERVA, which analyzes event-driven race reports to distinguish between false positives, benign races, and harmful races.

3. Debugging and fault location support: once the harmful race is confirmed, ERVA displays the event dependency graph as well as the flipped events, and can deterministically replay the app to help developers find the race's root cause.

[1]Available at http://spruce.cs.ucr.edu/valera/erva.html

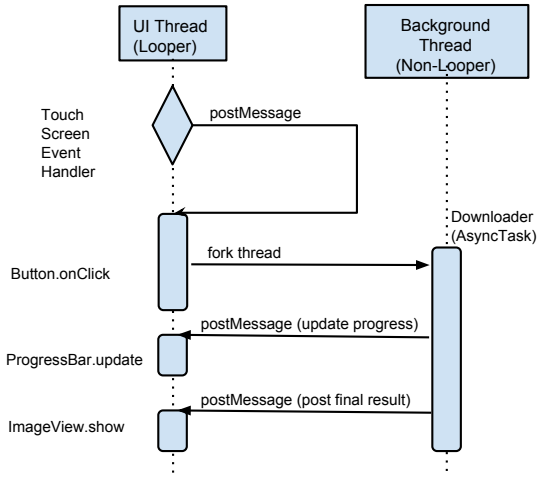

Figure 1: Thread model of a typical Android app.

4. An evaluation of ERVA on 16 real-world Android apps.

2. BACKGROUND: ANDROID AND ITS EVENT MODEL

The Android software stack consists of apps using the services of the Android Framework (AF). Each app runs as a separate process on top of a custom, smartphone version of the Linux kernel. Android apps are typically written in Java and compiled to either Dalvik bytecode that runs in a VM (Android version < 5.0), or directly to native code (Android version ≥ 5.0).

The Android platform is event-driven, with the AF orchestrating the app control flow by invoking user-provided callbacks in response to user and system events. The AF provides support for events, threads, and synchronization. In Android, threads can communicate with each other in two ways: via messages (the most common way) or via shared memory (as in traditional Java applications, used sparsely).

In Android's concurrency model, every application process has a main thread (also called "UI thread"); only the main thread can access the GUI objects, to prevent non-responsive threads from blocking the GUI. To update the GUI, other (non-main) threads can send messages to the main thread, and the main thread will dispatch these events to the appropriate user interface widgets. Long-running tasks such as network access and CPU-intensive operations are usually run in background threads. When these tasks are finished, the background threads post back messages (we call these messages *internal events*) together with the data to the UI thread. We now describe the Android threading model and then provide an example of how threads are used in a concurrent app.

Threading Model. The following grammar describes Android thread kinds.

$$\begin{array}{lcl} Thread & ::= & Looper \mid Non{-}looper \\ Non{-}looper & ::= & Background \mid Binder \end{array}$$

Looper threads are threads with an associated Looper object that confers threads message dispatching capabilities: the thread blocks waiting for messages and when a message comes, it is processed atomically. The main thread is a looper thread.

Background thread is the result of a regular thread fork() which does not register a Looper. *Binder thread* is created when an app is launched; binders are widely-used for inter-process communication (IPC). Each app holds a binder thread pool. The number of binder threads in the pool is automatically adjusted based on IPC usage.

Example. Figure 1 shows a standard Android app that downloads an image. When the user touches the touchscreen, the hardware touch events are delivered to the Window Manager Service (WMS). WMS keeps a record of all the apps' windows, i.e., window coordinates and layers. WMS checks the hardware touchscreen event's coordinates and sends it to the corresponding app. A handler is then invoked on the app's UI thread. The handler traverses the view tree hierarchy and invokes the corresponding view's action. If the user clicks a button, the handler posts an internal event with the onClick action to the UI thread's event queue. The onClick action forks a new background thread to download the image, offloading the task from the UI thread. The downloader thread may periodically send back internal events to show the percentage it has downloaded. When the download task is done, the downloader thread will post another event message along with the image to the UI thread. Finally, the UI thread decodes the image and displays it.

Event Model. The following grammar describes the Android event model.

Event	::=	*ExternalEvent* | *InternalEvent*
ExternalEvent	::=	*InputEvent* | *SensorEvent*
		| *IPC* | *HardwareInterrupt*
InputEvent	::=	*MotionEvent* | *KeyEvent*
SensorEvent	::=	*Compass* | *Accelerometer*
		| *Light* | ...
InternalEvent	::=	*Message* | *Runnable Object*

In Android, events can be either external or internal. *External events* originate in the hardware, cross into the OS and then into the AF. Apps can choose to use default handling for these events, in which case they are handled by the UI thread, or can register custom event handlers in other looper threads. Typical external events include input events (e.g., gesture or key events), sensor events (e.g., accelerometer, compass), IPC and hardware interrupts (such as VSYNC, a hardware "heartbeat" signal invoked 60 times per second). *Internal events* are messages or runnable objects sent from a non-looper thread to a looper thread. *Internal events* are created and sent via the Handler API at the AF or app level, rather than coming from the OS.

Event Posting. Event posting and processing is at the core of the Android platform. We have identified several event posting types. First, external events coming from the hardware or the OS that make a looper thread post messages to itself. For example, when the user clicks a button, the click gesture is actually a series of input events beginning with ACTION_DOWN and ending with ACTION_UP. In ACTION_UP, the UI thread will check which View object this event's coordinates are located in. If the View object has registered a click handler, the UI thread will post an onClickListener event to itself. Second, internal events, created and handled by the same thread — these events are created programmatically in the app code, in contrast to external events which come from the hardware or OS. Third, events (messages) generated by a looper, background or binder thread, and posted to another looper.

3. EVENT-BASED RACES: DEFINITION AND EXAMPLES

In this section we first present our model, including the happens-before relationship, which allows us to define true races, benign or harmful, as well as false positives. We then illustrate these on actual race reports in real-world apps.

3.1 Event-based Race Definition

We begin by defining the concurrency model in terms of threads, events, memory locations, and operations on them. ERVA first records per-thread traces of events/operations, then uses a set of rules to establish HB based on the traces, and finally classifies race reports into false positives, benign races, and harmful races. We now present the formal definitions that underlie ERVA's analyses.

Thread type	t	::=	$t^l \mid t^{nl}$
Access type	τ	::=	$read \mid write$
Memory location	$\rho \in Pointers$		
Memory access	α	::=	$\alpha_\tau(\rho)$
Message	$m \in Pointers$		
Runnable object	$r \in Pointers$		
Event posting	β	::=	$post(e, t^l, m \mid r, \Delta)$
Thread operation	γ	::=	$fork(t_1, t_2) \mid join(t_1, t_2)$
Operation	op	::=	$\alpha \mid \beta \mid \gamma$
Event	e	::=	$begin; op_1; ...op_n; end$
Looper trace	π_l	::=	e^*
Non − looper trace	π_{nl}	::=	$op_1; ...op_n$
Trace	π	::=	$\pi_l \mid \pi_{nl}$

Definitions. In our approach, threads t can be either loopers t^l or non-loopers t^{nl}. For each thread we record a trace. For looper threads, their traces π_l contains a (possible empty) sequence of events e. For looper threads, their traces π_{nl} contain sequences of operations. Operations op can be memory accesses α (which capture the location ρ and access kind $read$s or $write$s); thread operations γ, for example, $fork(parent_{tid}, child_{tid})$ or $join\ (parent_{tid}, child_{tid})$; or event postings β. Event postings create new events e (event types were defined in the "Event Model" part of Section 2) by either sending a message m or posting a runnable object r to looper thread t^l with a time delay Δ.

Happens-before relationship. Existing event-based race detectors [7, 18, 21] have proposed various HB definitions ($\prec$). We now proceed to define HB as a set of rules tied together by transitivity.

Program order rule: if an operation op_1 precedes another operation op_2 on the same thread in the trace, then they follow program order $op_1 \prec_\pi op_2$. Program order on non-looper threads implies HB, i.e., $op_1 \in t^{nl} \wedge op_2 \in t^{nl} \wedge op_1 \prec_\pi op_2 \implies op_1 \prec op_2$, but not on looper threads. Rather, HB on looper threads can only be introduced by the looper atomicity rule, discussed next

Looper atomicity rule: the order of operations executed within one event establishes HB; that is, if $op_1 \in e_k \wedge op_2 \in e_k \wedge op_1 \prec_\pi op_2$, then $op_1 \prec op_2$.

Event order rule: $e_1 \prec e_2$ if $end(e_1) \prec begin(e_2)$.

Event post rule: new events can be posted from looper threads t^l or non-looper threads t^{nl}. For the former case, say $\beta = post(e_2, t_1^l, m \mid r, \Delta) \wedge \beta \in e_1 \wedge e_1 \in t_1^l$, i.e., event e_1 posts an event e_2 to the looper thread t^l that e_1 belongs to, then $e_1 \prec e_2$. For the latter case, say $\beta = post(e, t^l, m \mid r, \Delta) \wedge \beta \in t^{nl}$, i.e., an event e is posted by a non-looper thread, then $\beta \prec e$ and $\forall \alpha \in t^{nl} \wedge \alpha \prec \beta$ we have $\alpha \prec e$.

Thread rule: if thread t_i creates a new thread t_j ($\gamma = fork(t_i, t_j)$), then $\forall \alpha \in t_j$ we have $\gamma \prec \alpha$. Similarly, for a thread join $\gamma = join(t_i, t_j)$, we have $\forall \alpha \in t_j \implies \alpha \prec \gamma$.

External event rule: in our model, each external event sensor s_i has a $begin(s_i, \theta)$ and $end(s_i, \theta)$ where θ is the sequence number. The external events within the boundary of $begin(s_i)$ and $end(s_i)$ are ordered by HB. For example, a click operation is a series of external events from the touch-screen starting with ACTION_DOWN, many ACTION_MOVEs, and ending with ACTION_UP. Here the $begin(s_i, \theta)$ is ACTION_DOWN and $end(s_i, \theta)$ is ACTION_UP. All the external events $e_1, e_2, \ldots, e_n$ within this boundary follow HB order. However, if e_1 and e_2 are from two different sequences, then there is no strict HB order. An example is two click operations that could be triggered in alternative order.

Android component lifecycle rule: callbacks in different components such as Activity, Service, Fragment, View, etc. are ordered by HB. For instance, Activity's onCreate callback is always invoked before its onDestroy. Based on Android's documentation [2], a lifecycle graph [8] can be built to precisely describe the HB relation between Android component callbacks.

Transitivity: HB is transitive, that is $\alpha_1 \prec \alpha_2 \wedge \alpha_2 \prec \alpha_3 \implies \alpha_1 \prec \alpha_3$ and $e_1 \prec e_2 \wedge e_2 \prec e_3 \implies e_1 \prec e_3$.

Event-based races. We can now state the race definition. We say that event e_i *races* with event e_j if there exists a shared variable ρ such that $\alpha_i(\rho) \in e_i$, $\alpha_j(\rho) \in e_j$ and $e_i \not\prec e_j$. On occasion we will refer to the event pair that satisfies this definition as "racy".

False positive. We define as *false positive* a race reported between e_i and e_j by a race detector (e.g., EventRacer) whereas ERVA can establish that the events are actually ordered by HB, i.e., either $e_i \prec e_j$ or $e_j \prec e_i$.

Access influence. Let $\alpha_1 = \alpha_{\tau 1}(\rho_1)$ and $\alpha_2 = \alpha_{\tau 2}(\rho_2)$. We say that access α_1 *influences* access α_2 (denoted $\alpha_1 \twoheadrightarrow \alpha_2$) if executing α_1 leads to a different value for ρ_2 compared to omitting α_1.

Benign race. We say that two events e_i and e_j have a *benign race* if they have an event-based race (which we defined above) on at least one location ρ but $\forall \alpha_i \in e_j$ and $\forall \alpha_j \in e_j$, we have $\alpha_i; \alpha_j \not\twoheadrightarrow \alpha_{EVS}$ and $\alpha_j; \alpha_i \not\twoheadrightarrow \alpha_{EVS}$. That is, the different order of executing α_i and α_j does not have any effect on the externally visible state (EVS). EVS can be customized by the user; ERVA's default EVS definition is presented in Section 4.6.

Harmful race. We define as *harmful* a race where event execution order influences program state and this is reflected into the EVS. More precisely, we say that two events e_i and e_j have a *harmful race* if they have an event-based race on at least one location ρ and $\exists \alpha_i \in e_i$, $\exists \alpha_j \in e_j$, such that $\alpha_i; \alpha_j \twoheadrightarrow \alpha_{EVS}$ or $\alpha_j; \alpha_i \twoheadrightarrow \alpha_{EVS}$. Harmful races can have various consequences, e.g., crash, exception, erroneous GUI state; we provide examples of harmful races in real-world apps in Section 5.1.

3.2 False Positive Type-1: Imprecise Android Component Model

False positives may arise due to imprecise modeling of the Android components and their interaction. Figure 2 shows an example: a race reported by EventRacer in AnyMemo (a flashcard app) that is actually a false positive. The RecentList Fragment is a sub-class of Android's Fragment component. In the onResume() callback, the app performs a database query and updates the recent list ri to the fragment views. Since the database access is time-consuming, to make the app responsive, the database query is performed by a background task (lines 14–16). When the task is done, a callback will be posted to the main thread, and the main thread updates the UI (lines 18–22).

```
1  public class RecentListFragment extends Fragment {
2    private ArrayAdapter mAdapter = null;
3    private Handler mHandler = null;
4
5    @Override
6    public View onCreateView(...) {
7      ...
8      mHandler = new Handler();
9      mAdapter = new ArrayAdapter(...);
10   }
11
12   @Override
13   public void onResume() {
14     Thread thread = new Thread() {
15       public void run() {
16         // query database operations
17         mHandler.post(new Runnable() {
18             public void run() {
19               mAdapter.clear();
20               for (RecentItem ri : database)
21                 mAdapter.insert( ri );
22             }
23         });
24     }}
25     thread.start();
26 }}
```

Figure 2: False positive type-1 in the AnyMemo app.

The race detector reports a race between onCreateView() and onResume() callbacks. Due to imprecise modeling, the race detector cannot find any HB relation between these callbacks. Hence, since onCreateView() writes the mAdapter variable and onResume reads the same variable, a read-write race is reported.

However, this race is actually a false positive. According to the Android documentation[2] a Fragment's onCreateView() method is always invoked before its onResume() method. Thus this read-write race can never happen.

3.3 False Positive Type-2: Implicit Happens-before Relation

Another category of false positives is due to imprecise modeling of happens-before relationship. Figure 3 shows an example of FP caused by implicit HB relation in Cool Reader, an eBook reader app. EventRacer reports that the callbacks onRecentBooksListLoaded and getOrLoadRecentBooks have a race condition because they both access the mBooks shared object, but the tool cannot derive any HB relation between these two callbacks. The CoolReaderActivity is an instance of an Android Activity subclass, i.e., a separate screen. Its lifecycle starts with the onStart() callback invoked on the main thread. In onStart, the app first starts the database service CRDBService. If the service starts successfully, a Runnable callback will be posted back to the main thread indicating that the database service is ready. The callback first tries to

[2]http://developer.android.com/guide/components/fragments.html

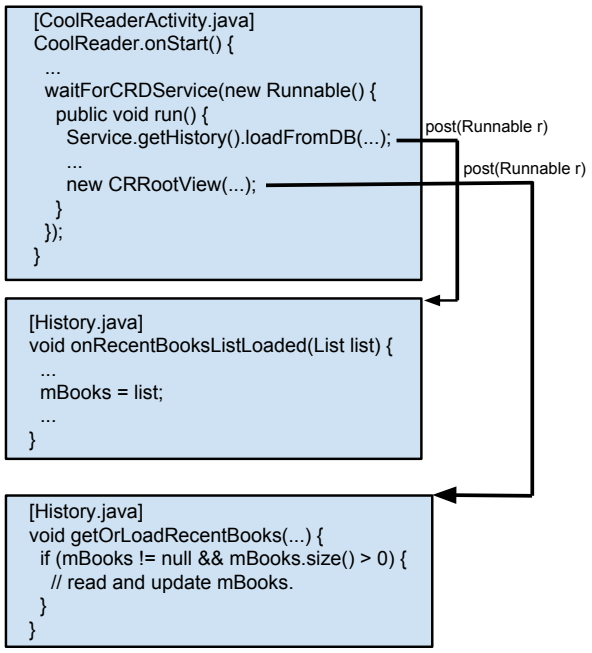

Figure 3: False positive type-2 in the Cool Reader app.

```
1  public class ImageLoader {
2    private Runnable mRunnable;
3
4    private void batchResponse(...) {
5      if (mRunnable == null) {
6        mRunnable = new Runnable() {
7          public void run() {
8            // deliver batched requests
9            mRunnable = null;
10         }
11       }
12       mHandler.post(mRunnable);
13 } } }
```

Figure 4: Benign race type-1 in the Volley library.

load some history records by invoking loadFromDB(), then creates a new CRRootView object. These two methods will both post callbacks (detailed implementation omitted).

Note that the loadFromDB and CRRootView initialization methods are invoked in the same action on the Looper thread (i.e., the main thread). According to the looper atomicity rule, loadFromDB happens before CRRootView; in other words, the calls loadFromDB and CRRootView are in program order. In the implementation of these two methods, they both use the Handler.post(Runnable r) to post callbacks. The post method inserts the actions into the queue in FIFO order. Since loadFromDB posts the callback before CRRootView, the callback onRecentBooksListLoaded will always happen before getOrLoadRecentBooks. However, EventRacer misses this implicit HB relation and thus reports a race, which is a false positive in this case.

3.4 Benign Race Type-1: Control Flow Protection

We now discuss a benign race in Volley, a popular HTTP library [4]. Figure 4 shows the relevant source code. EventRacer reports a race on the mRunnable object. The method

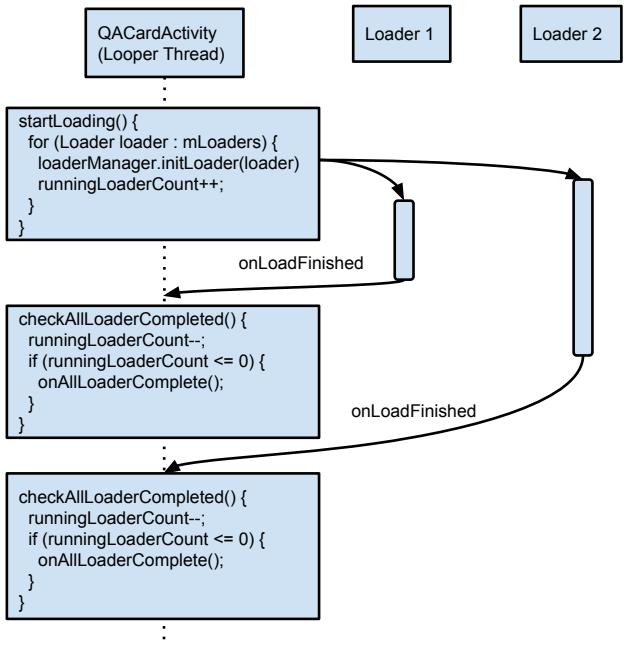

Figure 5: Benign race type-2 in the AnyMemo app.

batchResponse on line 4 and the creation of the Runnable object on line 6 are two distinct actions executed on the main thread. On line 6, the mRunnable object is updated to point to a new Runnable object while on line 9 it is set to **null**. Since EventRacer does not capture the HB relation between these two actions, it reports a write-write race, but it is a benign race.

The **null** test on line 5 can be true or false depending on when the next batchResponse is executed. Usually, the Runnable.run() is executed before the next batchResponse, the mRunnable will be set to **null** (line 9) hence in the next batchResponse a new Runnable (line 6) is created and posted to the main thread's looper (line 12). However, in cases when there are multiple batchResponse actions queued and executed before the Runnable.run(), the check on line 5 sees that mRunnable is already non-**null**, takes the **else** branch and does nothing. Thus the order in which the batchResponse and Runnable are executed does not matter due to the control flow protection offered by the **if** on line 5. This race is classified as benign.

3.5 Benign Race Type-2: No State Difference

Figure 5 shows an example of benign race type-2 in the AnyMemo app. When QACardActivity is launched, it will create several loaders to load data from the database or configuration files. In the startLoading method, the global variable runningLoaderCount tracks how many active loaders are currently running. When the loader finishes, it will post an onLoadFinished callback to the main thread and invoke the checkAllLoaderCompleted method. In this method, the variable runningLoaderCount is first decreased; if runningLoaderCount <= 0, it will invoke the onAllLoaderComplete callback to inform that all the loaders have finished their job.

Since the time spent in the loader is unpredictable, the order of these two onLoadFinished callbacks executed on the main thread is not deterministic. The race detector reports this pair of callbacks as a race because it cannot find any

HB relation and these callbacks do write to the same object, runningLoaderCount. Although this reported race is a true positive, it is actually harmless, because app state does not depend on the order in which the callbacks write to runningLoaderCount. ERVA can flip the execution order of the callbacks and does not find any harmful effect (EVS difference). Thus this race is classified as benign.

4. APPROACH

Figure 6 shows an overview of our approach; it consists of two phases, a *race detection phase* and a *race verification phase*. Note that ERVA relies on dynamic analysis and instrumentation hence the app source code is not required.

In the race detection phase, we run the app on an instrumented platform; traces described in Section 3.1 are collected at this stage. The platform's instrumentation consists of three main modules. First, a publicly-available custom Android emulator[3] with EventRacer running on top of it. Second, an *input capture* module provided by the VALERA [19] record-and-replay tool. Third, an *event capture* part, shown in thicker black line and font as it is a contribution of this work, unlike EventRacer and VALERA which are off-the-shelf tools. EventRacer runs the app and produces a *race report*. ERVA saves the instrumentation results in an *input log* and *EDG*, respectively. With these logs at hand, we proceed to the race verification phase.

In the verification phase, we replay the execution multiple times, flipping event order. The platform used for this phase can either be the emulator or an actual Android phone. Using the input log collected during detection, we use the *input replay* support of VALERA to ensure that the input provided to the app in this phase is the same as during detection (record). Due to *event flipping* we have multiple executions; we capture app state from each execution and then use *app state comparison* to classify potential race as either *false positive*, *benign race*, or *harmful race*. We now provide details on each of these components.

4.1 Race Detection

We choose EventRacer [7] as the race detector in ERVA as it is publicly available and robust. Compared with CAFA [18] and DroidRacer [21], EventRacer's HB model is more precise, while its race reports are easy to parse.

4.2 Input Capture and Replay

To capture app input for subsequent replay, we leverage VALERA [19], a tool that can record and replay app input, e.g., touchscreen, network, GPS, etc. VALERA can also capture and replay event schedules, but it does so "blindly" — it does not capture *event dependencies* that are instrumental for this work.

IPC Events. Android apps use IPC heavily, for isolation reasons. For instance, the Input Method Manager Service (IMMS) uses IPC: when the user inputs text into the current window, the soft keyboard is actually a global server service instead running in the app's address space. The IMMS receives the inputs from the user and dispatches them to the currently active window via IPC calls.

In Android, IPC is carried out via the Binder mechanism: each app has several Binder threads to handle incoming IPC calls. ERVA records the data and timing of each Binder transaction. The data contains two parts: primitive data and reference data. Reference data such as Binder object references and OS file descriptors are not deterministic across different runs. For primitive data, ERVA saves their concrete content while for reference data ERVA just keeps a slot in the log. The concrete value is filled in during the real execution.

Threads and Event Posting. ERVA intercepts thread and event operations to capture the HB relationships defined in Section 3.1. Android apps achieve asynchronous programming via message posting. When the background threads need to update the UI, they post messages (events) to the Looper on UI thread. As described Section 2, there are three types of event posting. ERVA captures these message by recording relevant API such as Handler.sendMessage and Handler.post.

4.3 Handling I/O Non-determinism

Besides event schedule non-determinism, I/O is another source of non-determinism. Since we use app state comparison to classify benign and harmful races, it is crucial to eliminate I/O non-determinism because it can affect app state and lead to divergence between different replays of the same execution. ERVA leverages VALERA [19] to make I/O input deterministic by recording and replaying I/O from a variety of sources: file system operations, network input, GPS, microphone, camera, random number API, etc.

4.4 Event Dependency Graph

By capturing the external and internal events and their posting, ERVA builds an event dependency graph (EDG). Figure 7 illustrates EDGs by showing an excerpt from the actual EDG of the TomDroid app. Each edge in the graph describes the causal relationship between events. For example, in Figure 7, the user performs two interactions with the app. First, the user clicks the 'Sync' button on the ViewNoteActivity. The onClick listener will create a background thread that performs a long-running task (data synchronization with the cloud). When the SyncThread is successfully created, the app will show an animation indicating that the task is running in the background. After the task is finished, the SyncThread will post an internal message to the UI thread to stop the animation. When the user is notified that the Sync task is done, she can use the 'Back' button to go back to the previous activity. The Back button press will trigger the onBackPressed() handler. The ViewNoteActivity sends an IPC binder transaction to inform the Activity Manager Service (AMS) to finish the current activity. AMS handles this transaction and then sends back an IPC to the app telling the UI thread to switch to the NoteListActivity . This activity contains updated content hence the user sees an updated screen.

The EDG precisely describes each event transition. Using the EDG, ERVA knows the root cause of a particular event. For instance, the ViewNoteActivity.updateUI() is triggered because the user has clicked the 'Sync' button and this event will create another background thread. During replay, an event can be ready to replay if and only if its recorded preceding event has been replayed. This is controlled by ERVA's underlying scheduler which will be described next.

4.5 Event Flipping

In Android, each Looper has an associated Message Queue. The Looper runs in an infinite loop, waiting for incoming

[3]http://eventracer.org/android/

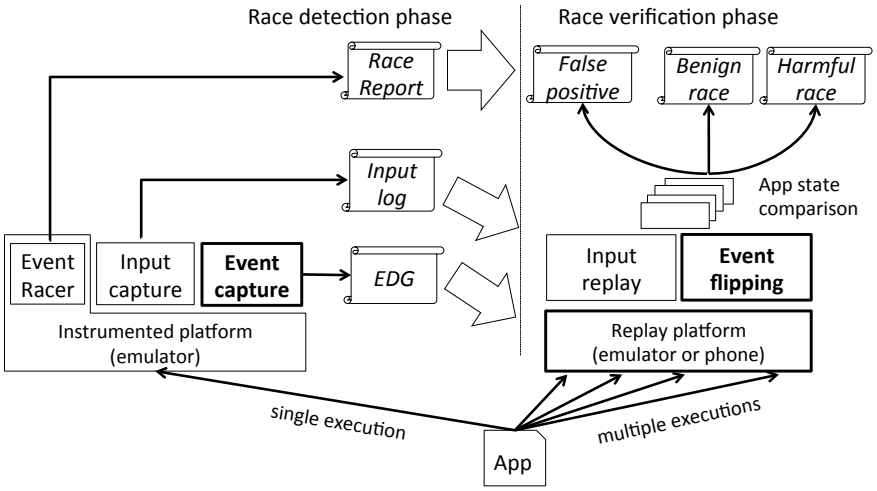

Figure 6: Overview of ERVA.

events and placing them in a queue. Messages (i.e., events) are dispatched by invoking the event handler's callback. We changed the Looper implementation to support flipping "racy" (unordered by HB) pairs of events, as follows. During replay, ERVA retrieves all recorded events (VALERA saves those to support event replay). Whenever our modified Looper receives an event, it checks whether this event is executable according to the EDG. If all the precedent events in the EDG have been executed, the message will be dispatched to the handler as usual. Otherwise, the event is postponed (added to a "pending" queue) because it could potentially be flipped.

For example, in Figure 7, the ViewNoteActivity.updateUI() and NoteListActivity .onResume() do not have an HB relation according to the race detector, which means their execution order can be flipped. To flip the event, ERVA adds a "fake" dependency edge in the EDG as shown in Figure 8. During replay, the updateUI event handler comes before the Back Key handler, but this time updateUI cannot be executed because it has a preceding event (NoteListActivity . onResume) in the EDG. Thus the event is added to the pending queue. When the NoteListActivity is brought back into the foreground, the onResume callback is invoked. After onResume is finished, the looper scheduler notices that onResume has a succeeding edge in the EDG, i.e., updateUI. The scheduler then inspects the pending queue, finds updateUI and allows it to execute. To summarize, this strategy guarantees that the order of events is flipped compared to the original (record) order.

4.6 State Recording and Comparison

Replay-based race classification has been used in prior tools [5, 26]: it starts from an execution that experienced one ordering of the racing accesses and re-runs it while enforcing another ordering, then it compares the states of the program to check whether the race is benign or harmful. The main problem of these tools is that, by using instruction-level deterministic replay their overhead is too high and would not work for Android apps for several reasons. First, Android devices usually have limited resources of computation and storage. Second, whole-system instruction-level replay would be difficult on mobile devices without hardware changes. Third, Android apps are sensitive to timing: large slowdown is likely to incur ANR (Android No Respond) errors. Fourth, Android's UI gestures are very sensitive to input timing and large overhead may change gesture semantic leading to replay divergence [15, 19].

We define *Externally Visible State* (EVS) as the subset of app state that might be accessed, or viewed, by the user; in ERVA the EVS consists of GUI objects (layouts, views, images) and Shared Preferences (a system-wide key-value store where apps can save private or public data [3]). The extent of the EVS can be customized by ERVA users. However, for this work we decided to limit the EVS to just the GUI and Shared Preferences for two reasons: (1) capturing more state e.g., file contents, would incur higher overhead and lead to spurious differences; and (2), Android event-race bugs tend to manifest as GUI differences or crashes [21, 18, 24].

Hence instead of recording and comparing whole-memory contents, ERVA finds state differences (hence harmful races) via EVS snapshot differencing, as follows: (1) in the original event order execution, ERVA snapshots the EVS upon entering or leaving each activity into $EVS_{original}$; (2) likewise, ERVA snapshots the EVS after the event order is flipped, into $EVS_{alternate}$; and (3) ERVA compares $EVS_{original}$ and $EVS_{alternate}$ to find differences—a benign true should show no difference, that is, the user cannot tell the difference between the original and alternate executions. Note that some differences might still exist in hidden state, e.g., memory contents or the VM stream, but these differences are not our focus — in our experience, many are spurious — rather, we expose those races that lead to visible EVS differences. In additions, ERVA allows the EVS definition (i.e., its extent) to be customized by the user.

4.7 Race Verification

As described in Section 3, ERVA classifies race reports into five bins: two types of false positives, two types of benign races, and harmful races. We now describe how ERVA performs this classification.

False positives type-1 occur because race detectors do not model the app's lifecycle callback events (or do not model them precisely). The result of event flipping is deadlock because an event with logical timestamp 1 cannot happen before another event with logical timestamp 2. Once ERVA detects deadlock after flipping the events, we bin this report

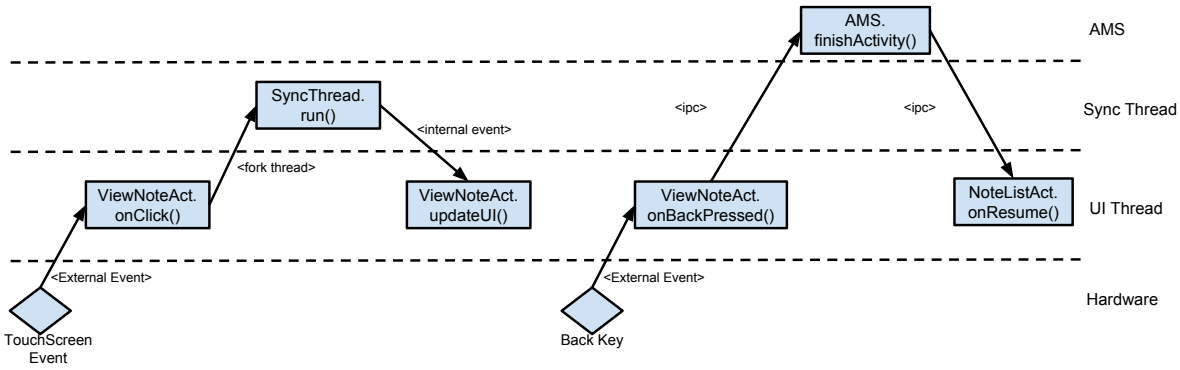

Figure 7: Excerpt from the EDG of the TomDroid app.

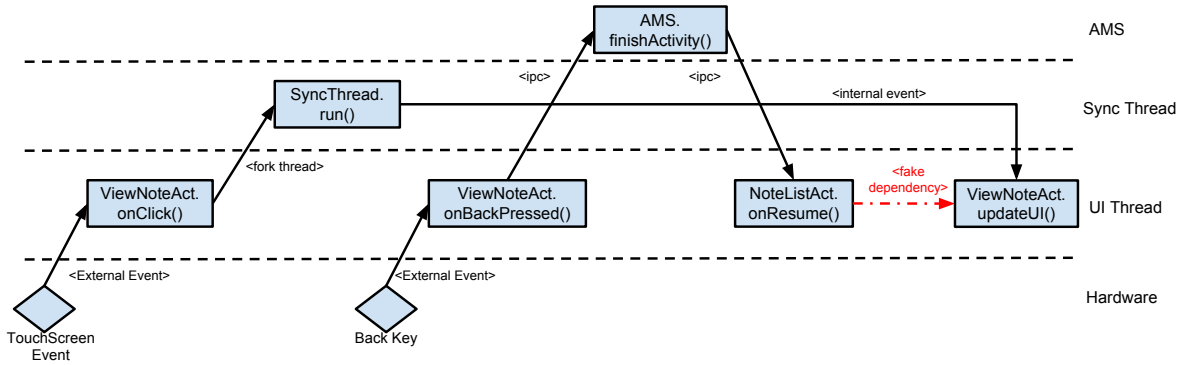

Figure 8: Performing event flipping on TomDroid's EDG in Figure 7.

as false positive type-1.

For *false positive type-2*, the cause is missing implicit HB relations. ERVA detects this type of FP by analyzing the EDG. For example, for onRecentBooksListLoaded and getOrLoadRecentBooks (the racy pair of events in the Cool Reader example from Section 3.3) the EDG shows that the event posters are within the save event and are ordered by program order. Since the Handler.post(Runnable) follows the FIFO property, these two events cannot be flipped. Note that, had one of the posters used postDelayed(Runnable r, long time), the events would be flippable.

For *benign race type-1*, memory accesses are protected by the control flow. ERVA first tries to flip the racy pair and finds the events can be flipped. Then, during event execution, ERVA enables tracing of instructions to detect reads and writes protected by control flow. In the example shown in Figure 4, the access of mRunnable is protected by the branch condition in line 5. By analyzing the instruction trace of flipped events, ERVA bins such reports as benign race type-1.

For *benign race type-2*, ERVA flips the order of events and does find that the memory value is different after flipping, thus this is a race. Next, ERVA dumps the state of the app (Section 4.6) and finds no difference. Thus ERVA considers this as a benign race type-2.

5. EVALUATION

We now describe our experimental setup, then evaluate the effectiveness and efficiency of ERVA.

Environment. The race detector used in our experiments is the publicly-available EventRacer for Android [1]. ERVA is based on Android version 4.3.0. All the experiments were conducted on the Android emulator on top of an 8-core 24GB desktop machine running 64-bit Ubuntu 14.04.2 LTS.

We have evaluated ERVA along two dimensions: (1) effectiveness in verifying races and (2) efficiency, i.e., the time required to process an app.

App dataset. We ran ERVA on 16 real-world apps (column 1 of Table 1). These apps were chosen according to several criteria: (a) spanning various categories, from note-taking to flashcards to news and utilities; (b) reasonable popularity — column 2 shows the number of downloads, in thousands, according to Google Play, all but two apps have at least 10,000 downloads while five apps have in excess of 1 million downloads; and (c) nontrivial size — column 3 shows their bytecode size, in KB.

5.1 Effectiveness

An effective race verification and reproduction tool should support developers in triaging reported races and allowing them to focus on true races, in particular on harmful races — this helps bug finding and fixing. We quantified ERVA's effectiveness on the 16 aforementioned apps.

We present the results of both EventRacer and ERVA in Table 1. The first set of grouped columns (columns 4–6) summarize EventRacer's output: the number of race reports and its breakdown as high or normal priority.[4] For example, for the CoolReader app, EventRacer reports 35 potential races; of these, 15 were high priority and 20 were normal priority. Presumably, the developer would proceed by trying to confirm the 15 high-priority races and then move on to the

[4]EventRacer classifies a report as high priority if the race is in app code and as medium priority if it is in the AF but invoked from the app.

Table 1: ERVA effectiveness.

App	# Downloads (thousands)	Bytecode size (KB)	EventRacer Android			ERVA			
			Race Reports	High Priority	Normal Priority	False Positives	True Positives	Benign Races	Harmful Races
AnyMemo	100–500	5,986	22	2	20	4	18	17	1
aLogCat	500–1,000	298	43	10	33	3	40	40	0
Aard Dictionary	10–50	3,466	6	1	5	1	5	5	0
AnStop Stopwatch	N/A	59	4	3	1	0	4	4	0
Cool Reader	10,000–50,000	854	35	15	20	20	15	15	0
DiskUsage	1,000–5,000	299	8	2	6	0	8	8	0
GhostCommander	1,000–5,000	1,699	30	8	22	11	19	18	1
GnuCash	50–100	5,511	26	2	24	8	18	18	0
Markers	500–1,000	89	8	2	6	0	8	8	0
Mirakel	10–50	6,191	20	1	19	8	12	12	0
Nori	1–5	1,045	5	2	3	2	3	3	0
NPR News	1,000–5,000	1,224	10	3	7	4	6	5	1
OI File Manager	5,000–10,000	675	8	3	5	2	6	5	1
OS Monitor	1,000–5,000	2,576	24	15	9	10	14	12	2
TextWarrior	50–100	237	5	3	2	0	5	4	1
Tomdroid	10–50	594	6	2	4	1	5	4	1
Total			*260*	*74*	*186*	*74*	*186*	*178*	*8*
(%)			*(100)*	*(28.5)*	*(71.5)*	*(28.5)*	*(71.5)*	*(68.5)*	*(3)*

remaining 20 normal-priority races – this is likely to be quite time-consuming. The last two rows show totals and percentages across all apps: out of 260 race reports, 74 (28.5%) are high priority while 186 (71.5%) are normal priority.

The remaining columns (7–10) summarize ERVA's output: the number of false positives, true positives, and among the latter, how many of the races were benign or harmful. For example, in the CoolReader app, ERVA has found that of the 35 reports produced by EventRacer, 20 were false positives and 15 are true positives; however, none of the true positives were harmful races.

The last two rows, which show totals and percentages across all apps, reveal that out of 260 race reports, 74 (28.5%) were false positives, 186 (71.5%) were true positives, and the 71.5% true positives were split as 68.5% benign, 3% harmful. Note that harmful races make up only 3% of the total number of race reports, which underscores the importance of race verification. We now discuss harmful races in detail.

Harmful races. Since ERVA offers deterministic replay, the schedules that expose the harmful races can be replayed, which helps developers find and fix the root cause of the race. We used this facility to manually confirm that the 8 races reported by ERVA as harmful were indeed harmful. Harmful races manifest in various ways. For example, some harmful races crash the app. In the TomDroid example discussed in Section 4.4, if the SyncThread and BACK key events are flipped, the app will crash due to a null pointer exception.

Even if the app does not crash, the consequences can still be deleterious. For example, AnyMemo has a harmful race that leads to an exception and different GUI state, and is caught by ERVA's state differencing. An excerpt[5] of the relevant code is shown next.

```
1 try {
2   // get data from database
3   adapter.insert(db.getData());
4 } catch (Exception e) {
5   Log.e("Exception Maybe caused by race condition.
        Ignored.");
6 }
```

[5]https://code.google.com/p/anymemo/source/browse/src/org/liberty/android/fantastischmemo/ui/RecentListFragment.java

If the event race occurs, the adapter object may be initialized improperly and its dereference will cause a NullPointerException. Interestingly, the developers are aware of the race but they simply use a **try** ... **catch** to handle the exception hence mask the effect of the bug. ERVA detects this bug via state differencing and reports that the race will cause a difference in the state of the View object.

Hence ERVA is effective at helping developers verify their apps, as well as find and fix races.

5.2 Efficiency

Since ERVA consists of detection and verification phases, we measured the time for each phase. We present the results in Table 2, individually for each app and the average across all apps in the last row. Recall that in the detection phase we run each app on an instrumented platform – we call this "online time" (column 2) and it takes on average 34 seconds per app. Following the online stage, EventRacer performs an offline analysis (column 3) which takes on average 54 seconds per app.

The time for the verification phase is presented in column 4: on average 1,111 seconds. This is due to ERVA having to perform multiple executions to flip events and compare state; we believe this time can be reduced substantially by using checkpointing to only replay program regions rather than whole executions [29], an idea we leave to future work. Finally, in the last column we present the total time (the sum of the detection and verification phases) for each app: it ranges from 245 to 3,297 seconds (1,198 seconds on average), which we believe is acceptable.

6. RELATED WORK

Race Detection. Race detection has been widely studied. Prior efforts have used either static [28, 12] or dynamic [13,

Table 2: ERVA efficiency: running time, in seconds.

App	Race Detection Phase		Race Verification Phase	Total
	Online	Offline		
AnyMemo	26.32	53.18	1264.56	1344.06
aLogCat	18.80	26.50	1671.84	1717.14
Aard Dictionary	30.26	48.91	391.08	470.25
AnStop Stopwatch	21.57	32.62	190.88	245.07
Cool Reader	43.88	102.47	3150.70	3297.05
DiskUsage	35.10	69.29	589.12	693.51
GhostCommander	27.53	44.70	1792.8	1865.03
GnuCash	38.81	57.29	2093.00	2189.10
Markers	25.08	36.74	435.84	497.66
Mirakel	39.78	65.12	1675.20	1780.1
Nori	48.15	57.32	507.80	613.27
NPR News	63.18	75.74	1305.60	1444.52
OI File Manager	37.56	41.03	632.32	710.91
OS Monitor	28.24	52.13	1414.56	1494.93
TextWarrior	30.11	47.50	320.10	397.71
Tomdroid	26.55	52.80	340.92	420.27
Average	*33.80*	*53.95*	*1111.02*	*1198.78*

9] 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 [18, 21]. 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 mobile or web applications. These apps adopt an asynchronous programming model and events handled by one thread come in non-deterministic order. Recent works have looked at detecting event-driven race patterns. For example, EventRacer [6, 24] detects event-driven races in web applications while EventRacer Android [7], CAFA [18] and DroidRacer [21] focus on Android apps. As mentioned in Section 1, these tools suffer from high false positive rates, cannot distinguish between benign and harmful races, and cannot reproduce races; these drawbacks are the main impetus for our work.

Race Classification. Race detectors that support race classification and prioritization are more likely to be adopted by developers, because developers can decide how to prioritize investigating race reports. Narayanasamy et al. [26] use instruction-level record-and-replay to replay alternate schedules, then compare the register/memory state to classify the races. Kasikci et al. [5] apply symbolic execution to classify the consequences of races by comparing their symbolic output result. However, both works focus on multi-threaded desktop/server apps; this approach is not suitable for mobile applications because of their event-driven nature. In contrast, ERVA captures and flips events according to the event dependency graph, rather than altering thread scheduling.

Model Checking. Model checking can help systematically explore all the nondeterministic schedules to find concurrency bugs. R4 [20] aims to find event-driven races in web applications; it uses dynamic partial order reduction (DPOR) [14] and bounded conflict-reversal to limit the total number of schedules to explore. Similarly, AsyncDroid [23] uses delay-bounded prioritized systematic exploration of the recorded schedule to find concurrency errors in Android apps. Unlike these model checking techniques which target finding new buggy schedules, ERVA checks only the potential racy events reported by the race detector and aims to verify whether they are false positives or harmless races. Thus, ERVA cannot detect bugs in unexplored schedules. R4 can check harmless races due to ad-hoc synchronization, but directly applying it to Android seems problematic: Android provides a number of system callbacks that have implicit happens-before relations, and ignoring these callbacks could cause false positives as the example of false positive type-1 shows. ERVA can check this type of false positives by flipping the events to see whether the system enters a deadlock condition. ERVA and model checkers could be combined. For example, the EDG from ERVA can be used as an auxiliary model for R4 and AsyncDroid in their exploration algorithm to check whether the new schedules are feasible or not. Furthermore, the EVS can be used to check the harmfulness of newly explored schedules.

Record and Replay. Record-and-replay has been widely studied and implemented on various platforms. On desktop/server platforms, replay approaches can be categorized into 3 groups: hardware modification [30, 22], virtual machine instrumentation [11, 27], and software-based [16, 31, 25, 10]. However, none of these approaches can be applied to mobile apps, because that would entail either changing the underlying mobile hardware (which is unrealistic) or the VM (which entails high overhead); and software-based approaches do not capture sufficient or suitable information for replaying Android apps due to their asynchronous nature. On the smartphone platform, tools such as RERAN [15] and Mosaic [17] support replaying GUI events, but not schedules. RERAN is device-dependent while Mosaic, like ERVA, is device-independent. Our own prior work, VALERA [19], supports schedule replay, and we use that support in ERVA. However, VALERA neither uses EDGs, nor can it flip events.

7. CONCLUSIONS

We have presented ERVA, an approach and tool for automatically verifying and reproducing event-based races in Android apps. ERVA addresses the imprecisions in current race detectors for Android by precisely modeling events and their dependencies, which allows it to categorize race reports and only point out those reports that are definite, harmful races. Experiments on 16 Android apps show that most races reported by race detectors are false positives or benign races, and that ERVA is an effective and efficient approach for automatically triaging and reproducing races.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1064646. Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

8. REFERENCES

[1] EventRacer Android. http://eventracer.org/android/.

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

[3] Android Developers. SharedPreferences. https://developer.android.com/reference/android/content/SharedPreferences.html.

[4] Android Developers. Transmitting Network Data Using Volley. https://developer.android.com/training/volley/index.html.

[5] C. Z. B. Kasikci and G. Candea. Data races vs. data race bugs: telling the difference with portend. In *ASPLOS'12*, pages 185–198.

[6] M. S. J. D. B. Petrov, M. Vechev. Race detection for web applications. PLDI'12, 2012.

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

[8] S. Blackshear, B.-Y. E. Chang, and M. Sridharan. Selective control-flow abstraction via jumping. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 163–182, New York, NY, USA, 2015. ACM.

[9] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. *SIGPLAN Not.*, 45(6), June 2010.

[10] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang. Octet: Capturing and controlling cross-thread dependences efficiently. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 693–712, 2013.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI'02*.

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

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

[14] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[15] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 72–81, Piscataway, NJ, USA, 2013. IEEE Press.

[16] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *OSDI'08*.

[17] M. Halpern, Y. Zhu, and V. J. Reddi. Mosaic: Cross-platform user-interaction record and replay for the fragmented android ecosystem. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015*, 2015.

[18] 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, pages 326–336, New York, NY, USA, 2014. ACM.

[19] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 349–366, New York, NY, USA, 2015. ACM.

[20] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. Vechev. Stateless model checking of event-driven applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 57–73, New York, NY, USA, 2015. ACM.

[21] 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, pages 316–325, New York, NY, USA, 2014. ACM.

[22] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05*.

[23] B. K. Ozkan, M. Emmi, and S. Tasiran. *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, chapter Systematic Asynchrony Bug Exploration for Android Apps, pages 455–461. Springer International Publishing, Cham, 2015.

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

[25] M. Ronsse and K. D. Bosschere. Recplay: A fully integrated proctical record/replay system. *TOCS*, pages 133–152, 1999.

[26] J. T. A. E. S. Narayanasamy, Z. Wang and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI'07*, pages 22–31.

[27] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX ATC'05*.

[28] J. W. Voung, R. Jhala, and S. Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the*

ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, pages 205–214, New York, NY, USA, 2007. ACM.

[29] Y. Wang, H. Patil, C. Pereira, G. Lueck, R. Gupta, and I. Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 98:98–98:108, New York, NY, USA, 2014. ACM.

[30] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03*, pages 122–135.

[31] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang. Order: Object centric deterministic replay for java. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, 2011.

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