Race Detection for Event-Driven Mobile Applications

Chun-Hung Hsiao[†] Jie Yu[*1] Satish Narayanasamy[†] Ziyun Kong[†]
Cristiano L. Pereira[‡] Gilles A. Pokam[‡] Peter M. Chen[†] Jason Flinn[†]

[†]University of Michigan [*]Twitter, Inc. [‡]Intel, Inc.

{chhsiao,jieyu,nsatish,kongzy,pmchen,jflinn}@umich.edu {cristiano.l.pereira,gilles.a.pokam}@intel.com

Abstract

Mobile systems commonly support an event-based model of concurrent programming. This model, used in popular platforms such as Android, naturally supports mobile devices that have a rich array of sensors and user input modalities. Unfortunately, most existing tools for detecting concurrency errors of parallel programs focus on a thread-based model of concurrency. If one applies such tools directly to an event-based program, they work poorly because they infer false dependencies between unrelated events handled sequentially by the same thread.

In this paper we present a race detection tool named CAFA for event-driven mobile systems. CAFA uses the causality model that we have developed for the Android event-driven system. A novel contribution of our model is that it accounts for the causal order due to the event queues, which are not accounted for in past data race detectors. Detecting races based on low-level races between memory accesses leads to a large number of false positives. CAFA overcomes this problem by checking for races between high-level operations. We discuss our experience in using CAFA for finding and understanding a number of known and unknown harmful races in open-source Android applications.

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

General Terms Algorithms, Languages, Reliability, Verification

Keywords Race detection, event-driven, mobile application, Android, concurrency, causality model, use-free race

1. Introduction

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

[1] This work was done when he was at the University of Michigan.

PLDI '14, June 9–11, 2014, Edinburgh, United Kingdom.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2784-8/14/06...$15.00.
http://dx.doi.org/10.1145/2594291.2594330

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

Wide use of asynchrony in event-driven mobile applications leads to pernicious concurrency bugs that are hard to find and debug. Thousands of events may get executed every second in a mobile system. Even if these events get processed sequentially in one thread, most events are logically concurrent to each other, as they may not be ordered by any programmer specified ordering operations.

Unfortunately, most existing tools such as data-race detectors [11] for finding concurrency errors assume a thread-based model. Naively applying these tools for event-driven mobile systems works poorly, because they implicitly assume that events handled in one thread are ordered by the program order. Recent work proposed a causality model for detecting races in the event-driven JavaScript applications [18, 20]. However, a causality model that accounts for the unique causal relations in a mobile system, and a race detector based on it has been lacking.

This paper presents the first race detection tool named CAFA for mobile applications. We chose to implement CAFA for the widely used Android platform [1], as the system and many of its applications are open-source. An Android application uses a special thread, named the *looper* thread, to select events from an event queue one at a time, and processes them sequentially by executing their handlers. There may be additional *regular* threads that communicate between each other and the looper thread using conventional synchronization operations.

CAFA is based on our new causality model for Android, which we use to infer the happens-before relations between events. We formulate this causality model based on the Android specification [1]. A unique aspect of this model is that it accounts for the causal relations between events due to the operations on the event queue and the properties of the events (e.g., delay) in an Android application. In addition, it accounts for the happens-before relations due to conventional synchronization operations.

We find that naively reporting races between assembly-level read and write accesses to a memory location in concurrent events leads to thousands of false positives. The reason is that concurrent events processed in the same looper thread could be *commutative* with respect to each other. Two events are commutative if they produce a correct result irrespective of the order in which they are executed by the looper thread. If two concurrent events processed in the same looper thread are commutative, then any conflicting memory accesses executed within them are not indicative of a race error.

Thus, only the conflicting operations in non-commutative events are indicative of race errors. Automatically determining if two events are commutative or not is a challenging problem, because it depends on high-level semantics of those events.

We address this problem by finding races between operations that are defined at a higher level semantics than low-level read and write accesses. In this study, we limit our focus to finding race errors that lead to *use-after-free* violations. A use-after-free violation arises when a pointer is dereferenced (used) after it no longer points to any object (freed). If a use and a free operation to the same pointer are executed in two concurrent events, it is possible that there exists an erroneous execution where the use is executed after the free. We refer to such races as *use-free* races. Use-after-free violations are common in Android applications, and a significant fraction of them are due to use-free races. CAFA finds such use-free races using our causality model.

While two concurrent events that contain a use-free race are likely to be non-commutative, it is not always the case. It is possible that when free is executed before the use, the programmer has taken care to either not execute the use or reallocate a new object before the use. We employ two simple heuristics, if-guard and intra-event-allocation, to filter the false positives due to this reason.

We implemented CAFA by extending several components in the Android stack. We extended the Android SDK to trace the event queue and synchronization operations to capture the causal relations. We modified the Dalvik Virtual Machine [2] to trace uses, free, allocation, and other low-level accesses necessary to implement our two heuristics. Finally, we extended the Android kernel to efficiently support a tracing device.

We studied several Android applications, which included applications such as FireFox and MyTracks (Google's GPS tracker). We found 67 previously unknown harmful races. We also detected 2 known harmful races. 60% of the races detected were harmful.

This paper makes the following contributions:

- We present the first causality model for the event-driven Android system.
- We discuss a race detector for finding races that lead to use-after-free violations using our causality model.
- We evaluate several Android applications and found 67 unknown and 2 known harmful races.

2. Background and Motivation

In this section we briefly describe the salient features of the event-driven programming model in Android. Then we discuss code examples illustrating the nature of concurrency bugs in these systems and the challenges in detecting them.

2.1 Event Driven Programming Model

We describe Android's event-driven programming model. It is a representative of the models used in a majority of current mainstream mobile platforms.

An Android application consists of several threads, which behave like conventional threads. A subset of these threads are designated as *looper* threads, whose role is to process *events* from their *event queues*. We describe each of these components below.

Event An *event* could be generated by an entity external to an application (e.g., sensor input, network, operating system), or internally by a thread or an event executed in the application. An event may be associated with a *time constraint* that determines when it can be processed. The time constraint can be either represented as an absolute timestamp, or as a delay with respect to the time when the event is generated. An event can be processed after its time constraint has been met. An event is processed by invoking its *event handler* based on its type.

Event queue Once an event is generated, it is placed in an *event queue*. Events in the queue whose time constraints have elapsed are processed in the order they were queued. Additionally, mobile platforms provide a `sendAtFront` API that places an event in the front of the queue. Some platforms provide mechanisms to prioritize events that need to be responsive, but these are not commonly used in the applications we studied.

Looper thread Each looper thread is associated with one event queue. The role of a looper thread is to continuously check its event queue, select and process one event at a time. Thus, all events executed in a looper thread are atomic with respect to each other. This is an important property of event-driven systems that programmers often rely on. However, the events in a looper thread are not atomic with respect to operations in the other threads.

Most event-driven programming models, including Android, allocate one looper thread for each event queue. Therefore in the remaining of this paper, we assume that *there is only one looper thread associated with each event queue.* The causality model we propose in Section 3 works on any event-driven programming model that follows this assumption, but is not applicable to a model that allows multiple looper threads sharing one event queue.

2.2 Challenges in Detecting Race Errors

Figure 1 presents a real race that leads to a use-after-free violation in Google's MyTracks application which is used to record GPS tracks. The race involves a looper thread from the MyTracks application and a regular thread in a different service process.

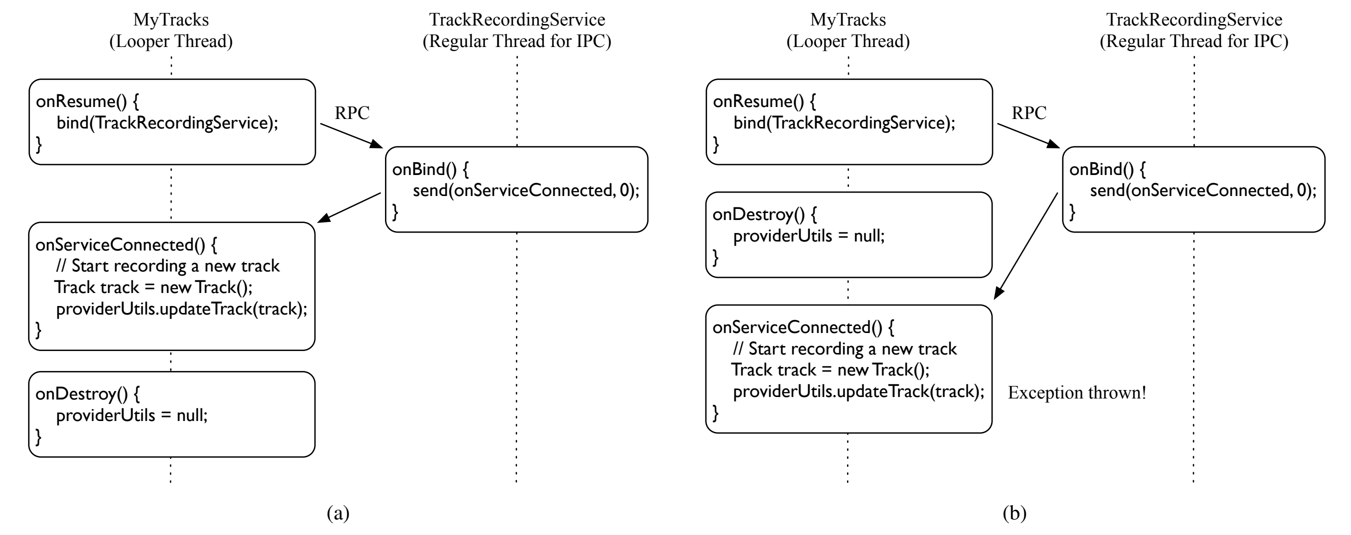
In the correct execution, when the user resumes the MyTracks application, it generates an onResume event. This event in turn invokes a remote procedure call (RPC) to establish a connection with an external service that tracks the GPS locations. The response received from the external process generates another onServiceConnected event which uses the providerUtils object. Later, the user stops the application, which generates the onDestroy event.

In the above execution, there is no programmer intended happens-before relation between the events onServiceConnected and onDestroy. Therefore, we consider them to be logically concurrent, even though they are executed sequentially in the same looper thread. Due to the lack of order between these two events, in another execution they might get executed in the reverse order as shown in Figure 1(b). This results in a null pointer exception to be thrown to the user, which is not a desirable behavior for the user.

The first challenge is in accurately modeling the causal relations between events, so that we can detect conflicting operations in concurrent events to find such errors. Section 3 presents a causality model.

The second challenge is in using the causality model to accurately find race errors in concurrent events. A conventional data-race detector [11] finds race bugs by finding conflicting memory accesses (read-write or write-write to the same location) that are not ordered by a happens-before relation. We found that naively reporting conflicting memory accesses in concurrent events leads to thousands of false positives. The reason is that, if two concurrent events executed in a looper thread are *commutative*, then they need not be ordered by a happens-before relation to ensure correctness.

Figure 2 shows an example for commutative events, which we commonly find in applications. The events are concurrent as there is no happens-before relation between them. Though the write and the read in two concurrent events to the variable manager.resideAllowed conflict, both orders for the events are correct. This conflict could



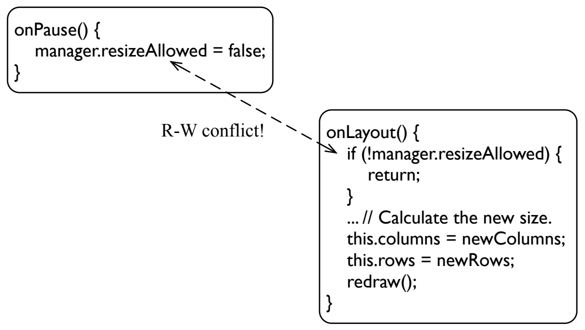
```
onResume() {
    bind(TrackRecordingService);
}

onBind() {
    send(onServiceConnected, 0);
}

onServiceConnected() {
    // Start recording a new track
    Track track = new Track();
    providerUtils.updateTrack(track);
}

onDestroy() {
    providerUtils = null;
}
```

Figure 1: A use-after-free violation in Google's MyTracks application. (a) A correct execution. (b) An incorrect execution where an use-after-free violation manifests owing to the lack of happens-before order between onServiceConnected and onDestroy.



```
onPause() {
    manager.resizeAllowed = false;
}

onLayout() {
    if (!manager.resizeAllowed) {
        return;
    }
    ... // Calculate the new size.
    this.columns = newColumns;
    this.rows = newRows;
    redraw();
}
```

Figure 2: A read-write race in the ConnectBot application. The memory write in onPause cannot be executed between the if statement and the succeeding statements that update the size information in onLayout because the two events in the same looper thread are executed atomically with respect to each other. Thus, this read-write race is not a bug.

cause an error only if the write to the variable can happen between the if-condition check and the return statement. This is not possible because the two events are guaranteed to execute atomically with respect to each other. Thus, reporting the read-write conflict would be false race error.

Automatically checking whether two events are commutative or not is a hard problem, as it depends on the high-level semantics of the two events. We tackle this problem by limiting our focus on a class of race errors that lead to use-after-free violations, and designing heuristics to check for the commutativity of operations that affect the execution of uses and frees.

3. Modeling Causality

The event-driven programming model described in the previous section differs substantially from the thread-based concurrency model that is assumed by most concurrency tools. Unlike in conventional thread-based causality models, we cannot assume that all the events executed in a thread are ordered by the program order. We present a causality model for the event-driven Android applications that relaxes this order, but accounts for additional happens-before relations due to the event queues.

3.1 Overview

An execution of an event-driven program in Android involves looper threads that process many events, as well as regular threads orchestrated by programmer-specified synchronizations such as locks, thread forks, and joins. In addition, events are generated through sends. The event handlers are either assigned along with event generation, or pre-assigned via a set of *event listeners*, which are functions that can be *registered* to the Android system and *perfermed* to process a specific class of events. Therefore, the causality model should account for the following three types of causal orders.

- We accounts for the *conventional* causal orders, including the program order in a regular thread or an event, and the orders due to programmer-specified synchronization operations, thread forks, and thread joins. This type of causal orders is similar to what we have in the conventional thread-based causality model, except for two major differences. First, *there is no program order between events processed by a looper thread*, because programmers do not intend to provide orders between events that are generated concurrently. Second, *no causal order is assumed between unlocks to succeeding locks*. That is because programmers do not intend to provide orders using locks in most scenarios, so assuming orders between critical sections protected by the same lock may introduce false happens-before relations.
- We accounts for the causal orders due to *event generation and execution*: events are sent before being executed, and event listeners are registered before being performed. In addition, we *conservatively assume causal relations between events generated in response to actions taken by entities external to an application*, such as network input, mouse clicks, or sensor readings, since these events might be causally ordered owing to user interactions.
- We accounts for the causal orders due to the *event queues*. In Androids event-driven programming model, events sent to the same event queue are atomically processed in the FIFO order by one looper thread once their time constraints are met, except

$$\begin{aligned} \mathit{Trace} &\rightarrow \mathit{Operation}^* \\ \mathit{Operation} &\rightarrow \mathit{begin}(t) \mid \mathit{end}(t) \mid \mathit{rd}(t,x) \mid \mathit{wr}(t,x) \mid \\ &\quad \mathit{fork}(t,u) \mid \mathit{join}(t,u) \mid \mathit{wait}(t,m) \mid \mathit{notify}(t,m) \mid \\ &\quad \mathit{send}(t,e,\mathit{delay}) \mid \mathit{sendAtFront}(t,e) \mid \\ &\quad \mathit{register}(t,l) \mid \mathit{perform}(t,l) \end{aligned}$$

$$\begin{array}{ll} t \in \mathit{Thread} \cup \mathit{Event} & x \in \mathit{Var} \\ u \in \mathit{Thread} & m \in \mathit{Monitor} \\ e \in \mathit{Event} & l \in \mathit{Listener} \end{array}$$

Figure 3: Operations in an event-driven program.

for those prioritized via the sendAtFront API. Therefore, it is incorrect to assume that there is never any order between events.

In the remaining of this section, we assume that *there is only one event queue and one looper thread* for simplicity. The model can be easily extended to model any event-driven systems that allocate only one single looper thread to every event queue. However, if a system allocates multiple looper thread to process one event queue, then no causal order can be guaranteed from the atomic FIFO processing order since events sent to the same event queue can be processed by concurrent looper threads. Therefore, our model is not applicable to such a system.

3.2 Event-driven Program Trace

We start with a definition of an execution trace of a event-driven program in Android. A program execution consists of a number of logically concurrent *tasks*, which are either events or regular threads that are spawned by programmers' code with thread forks. A trace of an execution is the list of operations performed by the various tasks in that execution. These operations are listed in Figure 3, which are described below.

The first set of operations are those that are analyzed in a conventional thread-based concurrency tools:

- $\mathit{begin}(t)$ and $\mathit{end}(t)$: begins and ends a task t, respectively.
- $\mathit{rd}(t,x)$ and $\mathit{wr}(t,x)$: reads and writes a value to variable x in task t, respectively.
- $\mathit{fork}(t,u)$: forks a new thread u from task t. $\mathit{join}(t,u)$: waits in task t until thread u ends.
- $\mathit{wait}(t,m)$ stalls task t until $\mathit{notify}(t,m)$, where m is the monitor.

While we account for mutual exclusion between the critical sections protected by the same lock, we do not assume a happens-before relation due to locks, as discussed in 3.1. Instead, we check the locksets for mutual exclusion, assuming that the critical sections are race-free since programmers handle them explicitly through locks.

The second set of operations are related to events:

- $\mathit{send}(t,e,\mathit{delay})$: enqueues a new event e at the end of the event queue in task t. e would be executed after *delay* milliseconds has elapsed since it is enqueued.
- $\mathit{sendAtFront}(t,e)$: enqueues a new event e at the front of the event queue in task t. $\mathit{sendAtFront}$ is used when the programmer wants to prioritize an event over earlier events.
- $\mathit{register}(t,l)$ and $\mathit{perform}(t,l)$: models the *event listener* programming construct in Android. An event listener can be performed as part of an event only after it has been registered with the runtime.

3.3 Causality Model

The *happens-before* relation $\prec_\alpha$ for a trace α is the smallest transitive closure over the operations in α that includes the following rules.

The first set of rules account for convensional happens-before relations:

Program-order rule: $a \prec_\alpha b$ if a occurs before b in α, and both a and b are performed by the same task.

Fork-join rule: $\mathit{fork}(t,u) \prec_\alpha \mathit{begin}(u)$ and $\mathit{end}(u) \prec_\alpha \mathit{join}(t,u)$.

Signal-and-wait rule: $\mathit{notify}(t_1,m) \prec_\alpha \mathit{wait}(t_2,m)$.

The second set of rules account for the happens-before relations for event generation and execution:

Event listener rule: $\mathit{register}(t,l) \prec_\alpha \mathit{perform}(e,l)$.

Send rule: $\mathit{send}(t,e,\mathit{delay}) \prec_\alpha \mathit{begin}(e)$ and $\mathit{sendAtFront}(t,e) \prec_\alpha \mathit{begin}(e)$.

External input rule: If e_1 and e_2 are generated from the external world, then $\mathit{end}(e_1) \prec_\alpha \mathit{begin}(e_2)$.

The last external input rule listed above conservatively assumes a happens-before relation between events generated by entities external to an application. This assumption could lead to false negatives, but not false positives. We partly address this problem by tracing all the dependent applications and service processes in the Android system.

We derived the atomicity and event queue rules by understanding the guarantees provided to the programmer by the Android API.

Atomicity rule: If $\mathit{begin}(e_1) \prec_\alpha \mathit{end}(e_2)$, then $\mathit{end}(e_1) \prec_\alpha \mathit{begin}(e_2)$.

The above rule models the fact that all events executed in a looper thread are atomic with respect to each other. If any operation in event e_1 happens before any operation in e_2, then all operations in e_1 and e_2 are ordered. For example, in Figure 4a, $\mathit{fork}(A,T)$ happens-before $\mathit{perform}(B,L)$, using which we can derive that $\mathit{end}(A)$ happens-before $\mathit{begin}(B)$. Henceforth, for brevity, we simply state that event A happens-before B, instead of $\mathit{end}(A)$ happens-before $\mathit{begin}(B)$.

The final set of rules account for the happens-before relations due to the event queue:

Event queue rule: When the sends of two events are ordered, then these two events are also ordered if any of the following conditions holds:

1. If $\mathit{send}(t_1,e_1,\mathit{delay}_1) \prec_\alpha \mathit{send}(t_2,e_2,\mathit{delay}_2)$ and $\mathit{delay}_1 \leq \mathit{delay}_2$, then $\mathit{end}(e_1) \prec_\alpha \mathit{begin}(e_2)$.
2. If $\mathit{send}(t_1,e_1,\mathit{delay}_1) \prec_\alpha \mathit{sendAtFront}(t_2,e_2)$ and $\mathit{sendAtFront}(t_2,e_2) \prec_\alpha \mathit{begin}(e_1)$, then $\mathit{end}(e_2) \prec_\alpha \mathit{begin}(e_1)$.
3. If $\mathit{sendAtFront}(t_1,e_1) \prec_\alpha \mathit{send}(t_2,e_2,\mathit{delay}_2)$, then $\mathit{end}(e_1) \prec_\alpha \mathit{begin}(e_2)$.
4. If $\mathit{sendAtFront}(t_1,e_1) \prec_\alpha \mathit{sendAtFront}(t_2,e_2)$ and $\mathit{sendAtFront}(t_2,e_2) \prec_\alpha \mathit{begin}(e_1$, then $\mathit{end}(e_2) \prec_\alpha \mathit{begin}(e_1)$.

The first event queue rule orders two events, if their sends are ordered, provided the delay condition holds. In Figure 4b, a regular thread executes two sends with the same delay in the program order. Enqueued events are processed in the FIFO order, guaranteeing that event A happens-before B. However, even when two sends are ordered, if the delay of the earlier event is even slightly greater

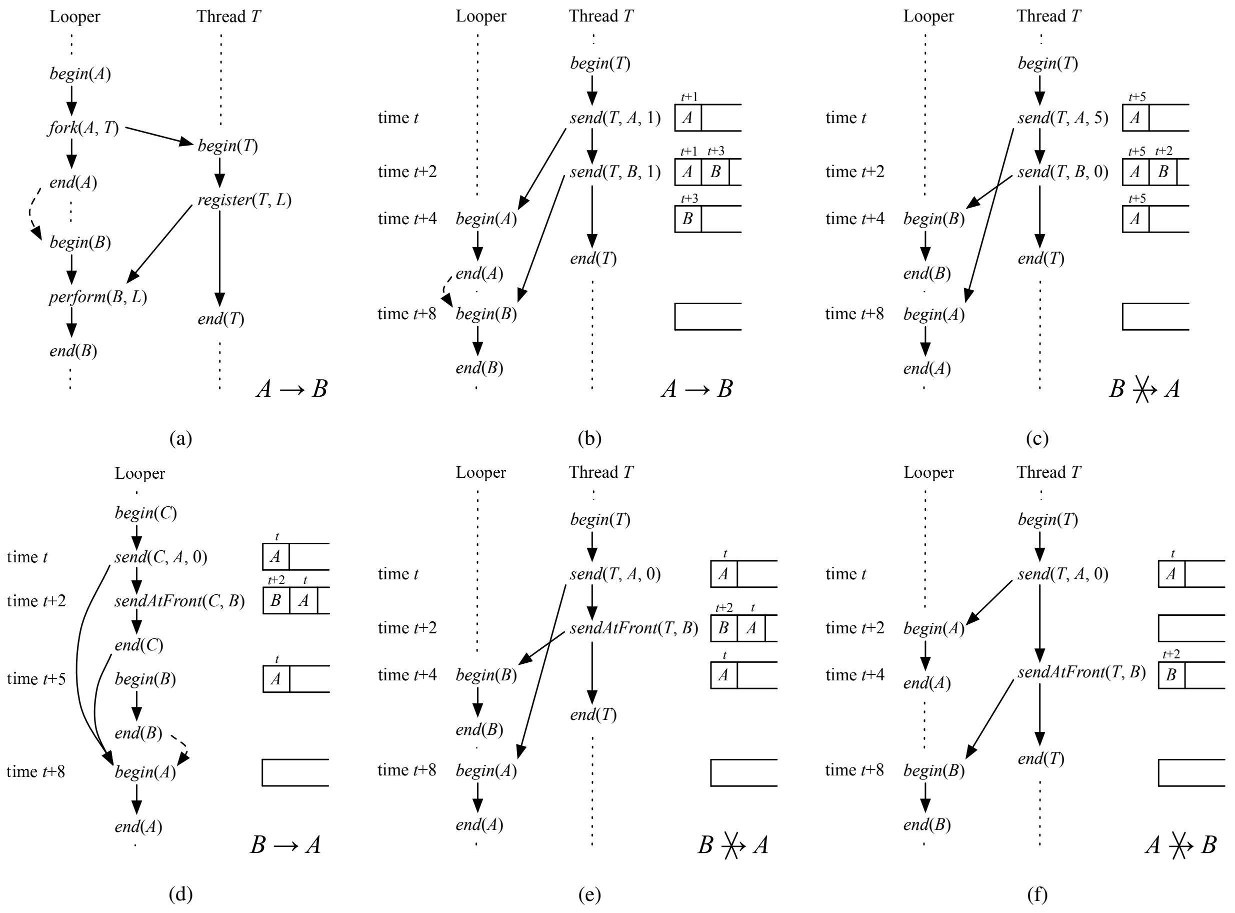


Figure 4: Examples illustrating causal relations due to atomicity and event queue rules. The derived happens-before relations are noted in each figure, and the dotted arrows indicate these derived relations. A cross implies that a happens-before relation cannot be derived. (a) The atomicity rule: since $fork(A,T) \prec perform(B,L)$, A happens-before B. In (b) to (f), the right columns show the statuses of the event queue. For each event, the earliest time when it can be processed is marked on top of each cell. (b) A happens-before B as their sends are ordered and their delays are the same. (c) A is processed after B owing to its higher delay, and thus no happens-before can be derived between A and B. (d) Event queue rule 2: since $sendAtFront(C,B) \prec begin(A)$, B is guaranteed to be enqueued in the front of the queue before A can be processed, so B always happens-before A. (e) and (f) show two scenarios where the $sendAtFront(C,B) \prec begin(A)$ property is not true, and as a result no happens-before relation can be derived between A and B.

than the delay of the latter event, then we cannot infer any happens-before between the two events, because there is a chance that the latter event can execute before the earlier event. For example, in Figure 4c, the send of A happens-before the send of B, but it is possible for B to execute before A as its delay is the lesser of the two.

The second event queue rule accounts for the happens-before due to a special send function in Android named $sendAtFront$. As explained in Section 3.2, $sendAtFront$ enqueues an event at the beginning of the event queue. This results in subtle new happens-before relations. As shown in Figures 4e and 4f, although $send(A)$ happens-before $sendAtFront(B)$, the two events A and B are not ordered, because both execution orders are possible as shown in the figures. However, there is one special condition under which we can derive a happens-before relation between events A and B. This is when it is guaranteed that $sendAtFront(B)$ happens-before $begin(A)$. Figure 4d shows one such instance where such a guarantee can be made. In this example, $send(A)$ and $sendAtFront(B)$ are both executed within event C. Also, event C is executed by the same looper thread as the one that processes events A and B. Since it is guaranteed that C ends before any other event can be processed, it in turn guarantees that $sendAtFront(B)$ happens-before $begin(A)$.

The third event queue rule simply states that if $sendAtFront(A)$ happens-before $send(B)$, then we can always derive that event A happens-before B. The fourth event queue rule states that if $sendAtFront(A)$ happens-before $sendAtFront(B)$, then under a certain condition similar to the condition in the second rule, we can derive that event A happens-before B. Notice that Android API does not allow programmers to specify a delay with $sendAtFront$ as it is meant for programmers to specify high priority events that need to be processed as soon as possible.

4. Race Detection

Data races are common concurrency bugs in multithreaded programs. Our initial study of bugs reported for open-source Android applications indicates that race-related bugs are prevalent in event-driven programs as well. However, conventional thread-based data race detectors work poorly if we apply them to find bugs in event-driven systems. In this section, we briefly discuss the limitations of these thread-based data race detectors, and present our race detector that finds use-after-free violations using the causality model discussed in the previous section.

4.1 Use-Free Races

Conventionally, a data race is defined as a pair of memory accesses, of which at least one is a write, and are not ordered by a happens-before relation. This definition, however, is not useful for detecting bugs in a mobile application. For example, there are 1,664 such races in a 30-second trace of ConnectBot, and most of them are not harmful bugs. One major reason is that we find a number of read-write and write-write races between concurrent events that are *commutative*. Two concurrent events are commutative if they produce a correct result irrespective of the order in which they are executed. Figure 2 shows a false positive example in ConnectBot.

We tackle this problem by limiting our focus on finding *use-after-free* violations due to races between concurrent events, and devising two heuristics to check if the racy events are commutative or not. A *free* is a write operation that sets an object pointer to null. A *use* is a read operation to an object pointer that would be dereferenced later. There is a *use-free* race if a use and a free are not ordered by a happens-before relation according to the causality model. Note that a use-free race is a special kind of read-write race, and may trigger a *use-after-free* violation if the free is executed before the use. By limiting our focus on this special kind of data race, we can find meaningful bugs without introducing lots of false positives.

Figure 1 shows a typical example of a use-free race: the use of providerUtils in onServiceConnected is racy with the free to provideUtils in onDestroy. In this case, reversing their execution order would cause a harmful program behavior and thus it is a use-after-free violation.

However, even if two events contain use-free races, it is possible for the racy events to be commutative for two reasons. The first reason is that if the use is not executed when the event containing the free is processed first, then no use-after-free violation would occur. For example, as can be seen in Figure 5, onPause and onFocus are commutative, because it is guaranteed that when onFocus is processed before onPause, the use in onPause will not be executed owing to the if-condition guarding the use.

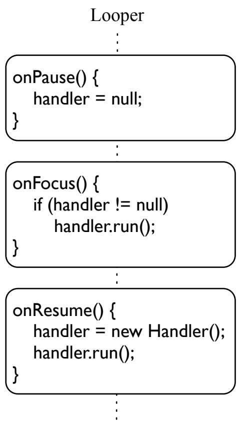
The second reason is that, if the pointer accessed by the use is assigned to a valid object address before the use is executed, then no use-after-free violation would be triggered. We call such an assignment an *allocation* to the pointer. For example, in Figure 5, onPause and onResume are commutative, because within onResume, there is always an allocation before the use.

In the rest of this section, we present an algorithm to find use-free races that may potentially cause use-after-free violations. Also, to reduce false positives, we present two effective heuristics, if-guard and intra-event-allocation. They check for the common cases of commutative events containing use-free races and filter the race warnings.

4.2 Finding Use-Free Races

There are well-known algorithms for finding data races based on a conventional causality model, such as the vector clock algorithm [11]. However, there are several challenges in adapting such an algorithm to use the event-driven causality model described in Section 3:

- The vector clock algorithm does not scale well as the number of concurrent tasks grows. In an event-driven system, the number of events can be very large (in the thousands). Also, many implementations assume that the number of tasks is known *a priori*, which is often not true in an event-driven system.



```
onPause() {
    handler = null;
}
```

```
onFocus() {
    if (handler != null)
        handler.run();
}
```

```
onResume() {
    handler = new Handler();
    handler.run();
}
```

Figure 5: An example of commutative events that contain uses and frees.

- There are operations whose happens-before relations rely on future operations, and thus the happens-before relations cannot be derived when these operations are observed. Figure 4a gives one such example. The relation $end(A) \prec begin(B)$ is derived based on $fork(A,T) \prec perform(B,L)$, and thus cannot be derived at the time $begin(B)$ is observed, since $perform(B,L)$ is a future operation with respect to $begin(B)$.
- To infer all happens-before relations, we need more complex checks on past operations than what are maintained in the vector clock algorithm. For example, in Figure 4d, the relation $end(B) \prec begin(A)$ is derived based on $send(C,A,0) \prec sendAtFront(C,B) \prec begin(A)$, which cannot be easily checked through comparing the vector clocks of $end(B)$ and $begin(A)$ since the latter relations involve past operations of $end(B)$ and $begin(A)$.

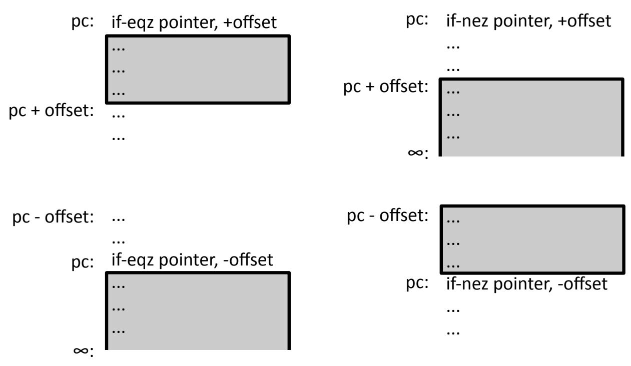
Because of the above reasons, we decided to collect the traces during an execution, and then run an offline algorithm to find use-free races. The offline algorithm first builds the *happens-before graph* for a trace α. It is a directed acyclic graph consisting of all operations in the trace as its vertices. For any operations a and b, $a \prec_\alpha b$ if and only if there is a path from a to b in the graph. Then to test if two operations are ordered, we simply perform a reachability test on the happens-before graph.

4.3 Pruning False Positives

To reduce the number of false positives, we present two simple heuristics: if-guard and intra-event-allocation. These heuristics recognize two common programming patterns that make concurrent events containing use-free races commutative. *Both heuristics are only applicable to events that are sent to the same event queue and processed by the same looper thread.*

If-Guard Check Programmers often check if a pointer is null before using it. Branch instructions used to perform this check can be leveraged to check if a use is safe or not. Therefore, in addition to log the operations described in Figure 3, we also log the following branch instructions that test on object pointers: if-eqz (jump if a pointer is null), if-nez (jump if a pointer is not null), and if-eq (jump if two pointers are equal).

The if-guard check for if-eqz and if-nez is illustrated in Figure 6. The heuristic works as follows. Suppose there is an if-eqz instruction at address pc, and it performs a forward jump to address pc + offset if pointer is null. Then we assume that *the code between* pc *and* pc + offset *is executed only if the branch is not taken at run-*



pc: if-eqz pointer, +offset
pc + offset:
pc: if-nez pointer, +offset
pc + offset:
∞:
pc - offset:
pc: if-eqz pointer, -offset
∞:
pc - offset:
pc: if-nez pointer, -offset

Figure 6: An illustration of the if-guard check. In each case, the branch instruction is located at pc and conditionally jumps forward or backward to pc + offset or pc - offset respectively. ∞ indicates the end of the current function. If a use of pointer in the shadowed area is executed after the branch instruction at runtime, pointer would be guaranteed non-null, so the use would be safe.

time, which guarantees that pointer would be non-null, and any use of pointer in this code region would thus be safe. Therefore, any use-free races involving such uses are ignored. Similar arguments can be applied to backward jumps, and to if-eqz instructions.

Note that the assumption we make is not sound. Without a control flow analysis, we cannot ensure that the code between pc and pc + offset is executed only if the branch is not taken. However, this assumption is often true for compiler-generated code in most applications. Hence we design the if-guard check based on this assumption.

In addition, we have found that the if-eq instruction that tests on two object pointers is often used to check if an object pointer is equal to this object in Java. Therefore, the if-eq instruction provides the same safety guarantee as if-nez does, and it is included in the if-guard check as well.

Intra-event-allocation If there is an allocation after a free in an event, then the null value written by the free will never become visible to any other event executed in the same looper thread. Therefore, use-free races involving such a free are ignored. Similarly, if there is an allocation before a use within the same event, then it is guaranteed that the use cannot read any null value written by a free outside the event. Any use-free races involving such a use are also filtered.

5. Implementation

We built a tool called CAFA to detect use-free races in Android applications. CAFA consists of a customized Android ROM (based on the Android Open Source Project [1] 4.3) and an offline analysis tool. The customized ROM instruments several key components in Android (e.g., the Dalvik Virtual Machine (DVM), and the core framework libraries) to collect execution traces for target applications and system services. The collected traces are later used by the offline analysis tool to reconstruct the happens-before graphs and detect use-free races. The customized ROM can be directly installed on some of the Android devices such as Google Nexus 4. CAFA is completely transparent, and thereby can trace and analyze unmodified Android applications.

In this section, we mainly discuss our instrumentation framework. We omit the details of the offline analysis tool as its implementation is pretty straightforward. This section is organized as follows. We first provide an overview for our instrumentation

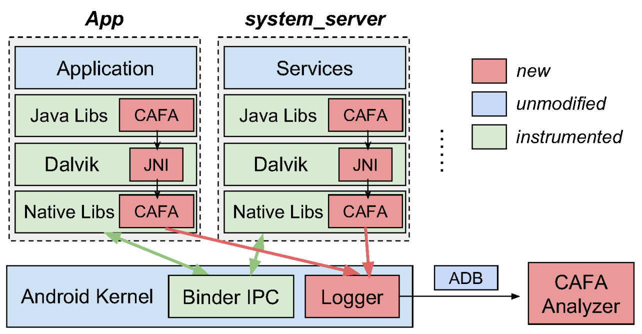


Figure 7: CAFA architecture overview. RED represents newly added components, BLUE represents unmodified components, and GREEN represents instrumented components.

framework. Then, we provide details about how we instrument various components in Android that allows us to capture the causalities described in Section 3. Finally, we discuss how we find and instrument potentially racy operations to capture use-free races.

5.1 Overview

Figure 7 shows the architecture of CAFA. We introduce a new logger device in Android kernel. All execution traces are sent to this logger device. The CAFA offline analyzer, which may reside in a remote server, can directly read traces from this logger device through the Android Debug Bridge (ADB). One can also choose to dump traces into a flash storage and process them later. We add a new native library called CAFA which provides interfaces for writing traces to the logger device. Each component in Android that needs instrumentation is linked with this native library (e.g., the Dalvik Virtual Machine and the core native utility library). We also add a Java binding for the CAFA native library to provide interfaces for Java programs (e.g., Java core library and Android core framework).

Inter-Process Communications (IPC) are heavily used in Android. For example, whenever an application wants to access system services like the GPS and camera services, it has to initiate Remote Procedure Calls (RPC) with a remote process called system_server. If we only collect traces for the target application just like most conventional race detectors do, we may miss many causalities caused by IPCs. For instance, an application may initiate an RPC call with the GPS service asking for the current location. Later, it receives a message from the GPS service containing the coordinates of the current location. If we do not collect traces for the GPS service, we may miss the causality between the RPC call and the receipt of the message in this case. Therefore, we collect traces not only for target applications, but for system services as well. We also instrument the IPC framework in Android (called Binder) which enables us to establish causalities across process boundaries.

5.2 Instrumentation for Capturing Causalities

As we described above, we need to instrument various components in Android to collect execution traces so that the offline analysis tool can later reconstruct the happens-before graph based on these traces. The following lists the major components that we have instrumented.

- **Java Core Library**. To track thread forks and joins, we instrument the Java core library in Android (i.e., we modify the java.lang.Thread class) to emit a trace entry every time a thread is forked or joined.

- **Dalvik Virtual Machine**. We also need to instrument all the thread begins and ends, as well as all synchronization primitives used by Java programs (e.g., java.lang.Object.wait and synchronized{...}). We achieve that by modifying the Dalvik Virtual Machine in Android. We assign a unique object ID for each object created by the virtual machine. This unique ID will later be used by the offline analyzer to capture those causalities caused by threading and synchronizations.
- **Android Core Library**. We also instrument the Android core framework library at both the native and the Java layer to capture traces like event begins and ends as well as sending of events (e.g., in android.os.Handler and android.os.Looper). These traces are crucial for us to analyze the causalities caused by event queues as we discussed in Section 3. In addition, the causalities caused by event listeners are done by instrumenting the listener registration functions and the internal functions that invoke the listeners. Currently we instrument for all event listeners in the android.app, android.view, android.widget, and android.content packages. Although we have accounted for most causalities due to event listeners, the packages listed above do not contain all event listeners and thus some causalities would be missed by CAFA.
- **Binder IPC Framework**. In order for the offline analyzer to capture the causalities caused by IPCs, we instrument the Binder IPC framework in Android. In fact, all the Remote Procedure Calls in Android are handled by the Binder IPC framework. A unique transaction ID is generated each time a process initiates a RPC call. The Binder transaction data is piggybacked with this transaction ID and sent to the remote process that handles this RPC call. All transaction related operations are also recorded and tagged with the corresponding transaction IDs. Later, the offline analyzer can capture the causalities caused by those IPCs by correlating transaction operations with the same transaction ID.
- **Other IPC Channels**. In Android, some latency critical IPCs, such as the display events and the input events, are performed through pipes (or Unix domain sockets), instead of the Binder IPC framework. CAFA handles these IPCs similarly by tagging those messages sent through pipes (or Unix domain sockets) with uniquely generated IDs.

5.3 Logging Potentially Racy Operations

Logging the low-level read and write operations as well as certain branch instructions is mainly done by instrumenting the DVM bytecode interpreter to log all related Dalvik bytecode instructions. In addition, method invocation/return and certain branch instructions are also instrumented for logging. All of the instrumentation are done in the portable interpreter mode, and we are also porting CAFA to the fast interpreter mode for better efficiency. Currently we don't support CAFA in JIT interpreter mode due to the complexity of tracing accesses in native code. The detailed instrumentation in the DVM bytecode interpreter for use-free race detection is described as follows.

- **Frees**. The Dalvik instruction set provides a set of instructions to write values to object pointers (e.g., i-put-object, s-put-object, and a-put-object). We instrument the DVM bytecode interpreter to emit a trace entry when such an instruction is executed. The log includes the ID of the object being dereferenced (if any), the address of the object pointer, and the written value. If the written value is null, then the instruction is a free; otherwise it is an allocation.
- **Uses**. Unlike frees, uses are harder to detect. A use involves a read from an object pointer (e.g. i-get-object, s-get-object, and a-get-object) to get an object, followed by an instruction that dereferences the object. The dereference instruction can be either an access to a field of the object, or a method invocation on the object. We instrument the DVM bytecode interpreter to emit a trace entry for the former read instruction to log the address of the pointer and the ID of the object it gets, and another entry that logs the ID of the dereferenced object for the latter field access or method invocation. The difficulty is that when we see a dereference instruction, we only know the ID of the dereferenced object, but have no idea which pointer it dereferences, since we cannot afford a data flow analysis at runtime. Therefore, we match a dereference instruction with its nearest previous pointer read that gets the same object ID. Although this heuristic is neither sound nor complete, it works well in practice.
- **Calling Context Stack**. The calling context stack is traced for 3 purposes: (1) To provide context information for reasoning about races. (2) To compute the relative address of each instruction so they can be mapped to the static Java code. (3) To log method invocations that dereference objects. For each method invocation, its method and return addresses are logged. We only log the name of a function upon its first invocation to reduce the size of a trace. For a method return, the method and return addresses are logged again. In addition, method exits through exception throwing are also logged.
- **If-Guard Check**. We instrument the DVM bytecode interpreter to log the if-eqz, if-nez, or if-eq instructions that test on object pointers. For an if-eqz instruction, a trace entry is emitted only when the branch is not taken; for an if-nez or if-eq instruction, a trace entry is emitted only when the branch is taken. The entry contains the current and target addresses of the branch instruction, as well as the ID of the testing object. Since we only have the object ID but no pointer address, we use a heuristic similar to the one that recognizes uses: a branch instruction is matched with its nearest previous pointer read to decide which pointer it tests on.

6. Evaluation

We applied CAFA on various open-source Android applications and successfully detected many real use-after-free violations. In this section, we first briefly describe the applications we tested, and then provide the accuracy and performance evaluation. All of our experiments were conducted on the latest 16GB model of Google Nexus 4, which is equipped with a Qualcomm Snapdragon S4 Pro quad-core ARMv7 processor. We built CAFA on the Android Open Source Project 4.3 r1.1.

6.1 Tested Applications

We used CAFA to detect use-free races in 10 popular open-source Android applications described as follows. These applications were picked from the built-in applications of the Android Open Source Project, the list of free and open-source Android applications in Wikipedia [3] and the F-Droid repository [4]. Each trace was collected through an execution of 10–30 seconds on the instrumented system.

ConnectBot[2] is an SSH client for Android. The version we tested is 1.7, which contains a known bug.[3] To collect the trace, we clicked on a remote host from the host list, and entered the password when a password prompt showed up. The trace stopped after we successfully logged in to the remote host.

[2] http://code.google.com/p/connectbot/

[3] https://code.google.com/p/connectbot/source/detail?r=90632bd675a9

MyTracks[4] records the user's moving information through collecting the GPS signals and using Google Maps to get the geographical information. We tested on version 1.1.7, which contains the bug shown in Figure 1. The trace was collected via running this application to record a short track, pausing it through switching to another application, and then switching it back.

ZXing[5] uses the built-in camera on a mobile phone to scan a barcode and then decode it to digital numbers. We tested on version 4.5.1. When collecting the trace, we scanned a real barcode, paused it by switching to the home screen, then switched it back and did another scan.

ToDoList[6] is an application such that you can add a to-do list widget on the screen to put notes and check completed tasks. We tested on version 1.1.7. The operations we performed to collect the trace are adding two notes to the widget, and then deleting them.

Browser is the built-in browser in the Android Open Source Project. When collecting the trace, we visited the Google homepage, search for "cse," click the link to the CSE department of University of Michigan, and then click the back button after the page is completely loaded.

Firefox[7] is a powerful open-source browser by Mozilla, and is available on Android. We tested on Firefox 25. We performed the same operations as testing Browser to collect the trace.

VLC[8] is an open-source media player on Android. We tested the latest version (0.2.0). We played a video clip for a few seconds, paused it and switched to the home screen, then switched back and continued playing for a few more seconds.

FBReader[9] is a free e-book reader on Android. We tested on version 1.9.6.1. We used this application to read its tutorial from the first page to the last page, rotate the phone, then move back to the first page to collect the trace.

Camera is the built-in camera software in the Android Open Source Project. We collected the trace by taking a picture, switching to the home screen, then switching back and taking another picture again.

Music is the built-in audio player in the Android Open Source Project. To collect the trace, we played an MP3 file for a few seconds, paused the music and switched to the home screen, then switched back and resumed the music for a few more seconds.

6.2 A Survey of Use-After-Free Violations

We have found several use-after-free violations among the tested applications. Most of these violations might be triggered when the application switches to the paused state. Typically, a cleanup procedure (e.g., freeing pointers) is called at this moment. As a result, any event that is scheduled after the pause event, which might be sent from another thread or process, would crash the application if it tries to use the freed pointers. For example, ZXing contains a bug of this kind.

Usually use-after-free violations would cause exceptions when they are triggered. Some of these exceptions would be caught to prevent the applications from crashing. However, sometimes these exceptions are not handled properly such that the behaviors of the applications do not meet a user's expectation. We consider such races buggy and believe that programmers should take care of such races more carefully. For example, CAFA reported that MyTracks contains use-after-free violations in the following method in MyTracks.java:

```
public void onServiceConnected(...) {
  ...
  try {
    // TODO: Send a start service intent and broadcast
    // service started message to avoid the hack below
    // and a race condition.
    ...
    startRecordingNewTrack(...);
  } finally {
    ...
  }
}
```

The startRecordingNewTrack method contains the racy code illustrated in Figure 2. As described in the TODO comment, instead of fixing the program state in the finally block to avoid a crash, a more appropriate way is to enforcing a happens-before order between this event and the racing event.

Another example of improperly handled exceptions can be found in ToDoList, where the author simply resolved the exception with the following code:

```
try {
  ...
  db.updateNote(...);
} catch (NullPointerException npe) { /* do nothing */ }
```

Although the above code prevents the application from crashing, the latest user input would not be written to the database and the data would be lost.

6.3 Precision

Table 1 shows the use-free races detected by CAFA in the tested applications. CAFA reported 115 use-free races, among which, 69 of them could lead to use-after-free violations and thus are *harmful*. These violations are further classified into 3 categories: (a) *intra-thread* violations due to races that happen between events in a looper thread; (b) *inter-thread* violations due to races that happen between threads but cannot be detected by a conventional data race detector; (c) *conventional* violations due to races that happen between threads and can be detected by a conventional detector. Here the "conventional" detector assumes a total order for all events in the same looper thread, but no causal order between unlock operations and their succeeding lock operations. Because we relax the event order within the looper thread, we are able to capture more inter-thread races than a conventional detector. In summary, 60% of the reported races are harmful. Note that the intra-thread violations are bugs, and the inter-thread and conventional violations are considered bugs in a DRF0 memory model [7], but may not necessarily be bugs in a stronger memory model such as SC [14].

We also analyzed the causes of false positives, including false races and benign races, and classified them into three major categories.

Type I false positives are false races due to missing happens-before order for event listeners. As described in Section 5, currently we only instrumented the event listeners in specific packages of the Android library. Having a more thorough instrumentation, we believe it would be very promising to remove most of the false positives of this class.

Type II false positives are benign races that happen primarily because the if-guard and intra-event-allocation heuristics we used are not able to precisely check if two events are commutative. For example, the if-guard check infers that a use is safe only if there is

[4] http://code.google.com/p/mytracks/

[5] http://code.google.com/p/zxing/

[6] https://github.com/chrispbailey/ToDo-List-Widget

[7] http://www.mozilla.org/en-US/firefox/fx/

[8] http://www.videolan.org/vlc/

[9] http://fbreader.org/FBReaderJ

Application	Events	Races reported	True races			False positives		
			(a)	(b)	(c)	I	II	III
ConnectBot	3,058	3	0	2	0	1	0	0
MyTracks	6,628	8	1	3	0	0	4	0
ZXing	4,554	5	0	2	0	1	1	1
ToDoList	7,122	9	8	0	0	0	1	0
Browser	3,965	35	0	8	19	1	7	0
Firefox	5,467	25	0	6	10	4	5	0
VLC	2,805	7	0	0	1	0	5	1
FBReader	3,528	9	1	3	1	2	2	0
Camera	7,287	9	1	1	0	0	5	2
Music	6,684	5	2	0	0	0	2	1
Overall		115	13	25	31	9	32	5

Table 1: Races reported by CAFA. (a) Races that lead to intra-thread violations. (b) Races that lead to inter-thread violations. (c) Races that lead to conventional violations.

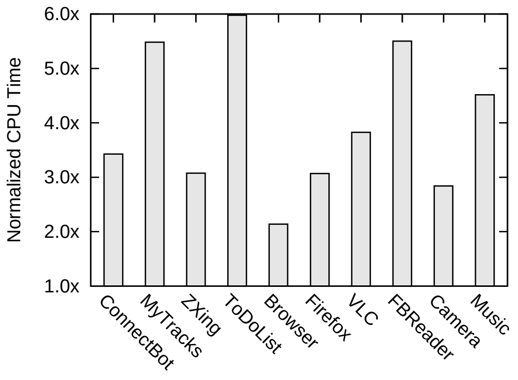


Figure 8: The slowdown for CAFA to collect traces on various applications.

a pointer test, but if the programmer uses a boolean flag to indicate that the pointer is safe for dereference, if-guard would not infer this information.

Type III false positives are false races that happen because CAFA mistakenly matched dereference instructions to incorrect pointer reads. When such mismatches happen, the pointer reads would be incorrectly recognized as uses, and false races would be reported if there are racing frees. Currently CAFA only uses the traces to recognize use operations. It can be improved by performing a static data flow analysis on the Dalvik bytecode of the applications to accurately match the dereference instructions to the corresponding pointer reads.

6.4 Performance

Figure 8 shows the slowdown of CAFA when collecting traces for each application. The slowdown is between 2x to 6x compared to their uninstrumented executions. The running time of the offline analysis depends on the number of events in a trace. For most applications, the running times vary from 30 minutes to 10 hours, except for ToDoList and Music, which took about 16 hours and 1 day respectively, due to the excessive amount of events.

7. Related Work

In this section, we discuss closely related work. We first compare our work with existing data race detection techniques in the literature. Then, we discuss techniques for detecting other types of bugs in mobile applications.

7.1 Race Detection for Thread-based Programs

Many studies have been done in the literature to detect data races, either statically [10, 22], or dynamically [11, 15]. However, these techniques are mostly designed for thread-based programs, and usually work poorly for event-based programs. For example, FastTrack [11] assumes that all memory accesses from the same thread are totally ordered, thus no race will be reported for accesses executed in the same thread. As we already pointed out, this assumption is too strict (missing potentially races) for event-based programs since events executed in the same thread in an event-based program can be logically concurrent.

7.1.1 Race Detection for Event-based Programs

Recently, a few researchers have shifted their focuses, and start looking at race detection techniques for event-based programs. WebRacer [18] and EventRacer [20] are two recent studies focusing on detecting races for one type of event-based programs: web applications. A web application is typically executed by the browser in a single thread in an event-driven style. These authors have shown that even if there is only one thread executing, races are still possible. For example, a buggy web application may allow a JavaScript function to be invoked even before it is loaded by the browser, causing unexpected behaviors. To detect such type of races, these authors redefined the happens-before relation (causality) for web applications, based on which they build tools and have successfully found races in many popular web sites.

Though closely related, our work differs with theirs in the following aspects. First, ours model the event queue and theirs do not: two events are executed in order if they are generated in order. This property is absent in their work and thus their algorithm is not applicable for detecting such causalities. We argue that this is important rule since a programmer would expect such a behavior. This property makes our work more general and can be easily applied to various types of event-based applications that rely on event queues, including web applications on which their work has focused and actor based programs [5, 6, 8]. Second, the types of bugs they targeted are also web application specific. In contrast, we focus on a more general type of bugs: *use-after-free* violations. Finally, we target Android applications, which have a hybrid threading model (looper threads combined with regular threads), where events can spawn regular threads and communicates with other events and threads through the shared memory. This poses more challenges when modeling causality for them.

P [9] is a domain-specific language to write asynchronous event-driven code. A P program can be fully verified using model checking. However, P is mostly designed for writing device drivers, which usually have relatively small size. Its technique cannot be easily applied to large systems like Android.

7.1.2 Effect-Oriented Race Detection

In this paper, we take an effect-oriented approach [25] by focusing on *use-after-free* violations. ConMem [25] also takes an effect-oriented approach. Instead of detecting low-level conflicts between memory accesses, it focuses on those races that could potentially cause memory errors. For example, it looks for races between dereference and nullification of a pointer. The main difference between ConMen and our work is that we target event-based programs, which have a different causality model than the thread-based programs they focus on. It turns out that using an effect-oriented approach for detecting races is more important and necessary for event-based programs because low-level memory conflicts can even happens within a thread under our new causality model. As a result, deciding if two conflicting low level memory accesses are commutative becomes more challenging. Using an effect-oriented approach can help us avoid a large volume of false positives.

7.1.3 Predictive Race Detection

We take a predictive approach [13, 24] to detect races in Android applications. We relax the happens-before order between events executed by the same looper thread, as well as the happens-before order caused by critical sections, because we think there is no causality between them and they can potentially be executed in a different order in alternate executions. As a result, like other predictive race detectors, our tool can find more bugs but at the cost of producing false positive (unlike a sound dynamic race detector such as FastTrack [11] which does not produce false positives). Smaragdakis et al. [21] discussed a sound predictive race detection technique in the past, but it is for thread-based programs and has a high computational cost.

7.2 Commutative Analysis

We choose to focus on high-level *use-after-free* violations because automatically deciding if two low-level memory accesses are commutative or not is hard. Huang et al. [12] discussed a few heuristics in the past to check if two critical sections are commutative or not. However, their approach is for thread-based programs, thus cannot be directly applied to event-based programs.

7.3 Bug Detection for Mobile Applications

Mobile computing has become increasingly popular recently. Many tools have been proposed to detect various kinds of bugs in mobile applications. Performance bugs are a class of bugs that received wild interests lately as mobile applications are usually user facing and latency critical. Performance bug detection tools like AppInsight [19] and Panappticon [23] analyze critical paths in the system to identify performance bottlenecks in the applications that could potentially cause user perceived delays. Energy bugs are another type of bugs that have been studied recently. Pathak et al. have performed a series of studies [16, 17] on classifying and characterizing energy bugs in Android applications. As far as we know, no previous work has looked at detecting races in Android applications.

8. Conclusion

Mobile applications are increasingly popular and are written by common programmers. These applications are written in an asynchronous event-driven model, which is prone to concurrency errors. Unfortunately, currently we do not have adequate tools to help programmers find races in these event-driven mobile systems. This paper presented the first causality model for the Android systems, and a tool that detects use-after-free races using this causality model. Our study showed that a significant number of harmful races could be found with adequate accuracy.

Acknowledgments

We thank the anonymous reviewers for comments that improved this paper. This work is supported by the National Science Foundation CAREER program and Intel, Inc.

References

[1] Android Open Source Project. `http://source.android.com/`.

[2] Dalvik Technical Information. `http://source.android.com/devices/tech/dalvik/index.html`.

[3] List of free and open-source Android applications. `http://en.wikipedia.org/wiki/List_of_free_and_open-source_Android_applications`.

[4] F-Droid. `https://f-droid.org`.

[5] Twitter Finagle. `https://github.com/twitter/finagle`.

[6] Libprocess. `http://www.eecs.berkeley.edu/~benh/libprocess/`.

[7] S. V. Adve and M. D. Hill. Weak ordering&mdash;a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 2–14, New York, NY, USA, 1990. ACM. ISBN 0-89791-366-3. . URL `http://doi.acm.org/10.1145/325164.325100`.

[8] J. Armstrong, R. Virding, and M. Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993. ISBN 978-0-13-285792-5.

[9] A. Desai, V. Gupta, E. K. Jackson, S. Qadeer, S. K. Rajamani, and D. Zufferey. P: safe asynchronous event-driven programming. In *PLDI*, pages 321–332, 2013.

[10] D. R. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, pages 237–252, 2003.

[11] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[12] R. C. Huang, E. Halberg, and G. E. Suh. Non-race concurrency bug detection through order-sensitive critical sections. In *ISCA*, pages 655–666, 2013.

[13] V. Kahlon and C. Wang. Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs. In *CAV*, pages 434–449, 2010.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, C-28(9):690–691, Sept 1979. ISSN 0018-9340. .

[15] R. H. B. Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *Workshop on Parallel and Distributed Debugging*, pages 1–11, 1993.

[16] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *HotNets*, page 5, 2011.

[17] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys*, pages 267–280, 2012.

[18] B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *PLDI*, pages 251–262, 2012.

[19] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: mobile app performance monitoring in the wild. In *OSDI*, pages 107–120, 2012.

[20] V. Raychev, M. T. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *OOPSLA*, pages 151–166, 2013.

[21] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *POPL*, pages 387–400, 2012.

[22] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC/SIGSOFT FSE*, pages 205–214, 2007.

[23] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. A. Dinda. Panappticon: Event-based tracing to measure mobile application and platform performance. In *CODES+ISSS*, pages 1–10, 2013.

[24] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. W. Reps. ConSeq: Detecting Concurrency Bugs through Sequential Errors. In *ASPLOS*, pages 251–264, 2011.

[25] W. Zhang, C. Sun, J. Lim, S. Lu, and T. W. Reps. Conmem: Detecting crash-triggering concurrency bugs through an effect-oriented approach. *ACM Trans. Softw. Eng. Methodol.*, 22(2):10, 2013.