Precise Static Happens-Before Analysis for Detecting UAF Order Violations in Android

***Abstract*—Unlike Java, Android provides a rich set of APIs to support a hybrid concurrency system, consisting of both Java threads and an event queue mechanism for dispatching asynchronous events. In this model, concurrency errors often manifest themselves in the form of order violations. An Android order violation occurs when two events access the same shared object in an incorrect order, causing unexpected program behaviors (e.g., null pointer dereferences).**

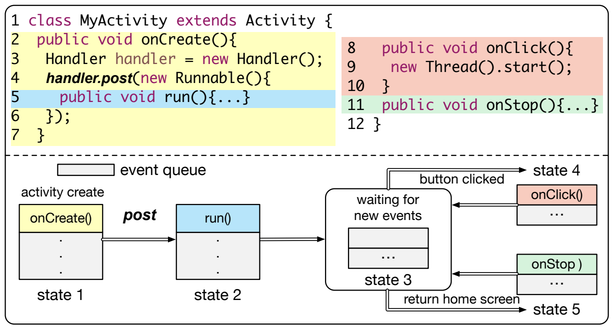
This paper presents SARD, a static tool for detecting both intra- and inter-thread use-after-free (UAF) order violations, when a pointer is dereferenced (used) after it no longer points to any object (freed), through systematic modeling of Android's concurrency mechanism. We propose a new flow- and context-sensitive static happens-before (HB) analysis to reason about the interleavings between two events to effectively identify precise HB relations and eliminate spurious event interleavings. We have evaluated SARD by comparing against NADROID, a state-of-the-art static order violation detection tool for Android. SARD significantly outperforms NADROID in terms of both precision (by reporting three times fewer false alarms than NADROID given the same set of apps used by NADROID) and efficiency (by running two orders of magnitude faster than NADROID).

I. Introduction

The significant growth of multi-core smart phone devices provides unprecedented opportunities for mobile apps to perform sophisticated tasks that are comparable to softwares on desktop/laptop computers. In order to explore the full capability of multi-core mobile phones, Android provides a rich set of APIs to support a hybrid concurrency system consisting of both traditional Java threads and an event queue mechanism for dispatching asynchronous events. However, such a system introduces both intra-thread concurrency bugs (caused by asynchronous events) and inter-thread bugs (caused by both events and Java threads) which are extremely difficult to detect.

Android execution is driven by asynchronous events in event queues. An Android event can be posted into an event queue externally via UI interactions (e.g., click and swipe) and system notifications (e.g., activity create) or internally via calling event-posting related APIs (e.g., `handler.post(...)` `and handler.sendMessage(msg)`) in application code. Later, Android's `Looper` object constantly fetches an event from the head of the queue and dispatches the event by executing its corresponding asynchronous method.

Android Event Queue. An event queue follows the first-in-first-out (FIFO) principle for dispatching events in a sequential order. However, due to the event-driven nature of Android, both posting and dispatching an event in the queue are non-deterministic driven by user interactions. Worse, apart from the main UI thread, Android allows Java multithreading and



```
1 class MyActivity extends Activity {
2  public void onCreate(){
3   Handler handler = new Handler();
4   handler.post(new Runnable(){
5     public void run(){...}
6   });
7  }
8    public void onClick(){
9     new Thread().start();
10   }
11   public void onStop(){...}
12 }
```

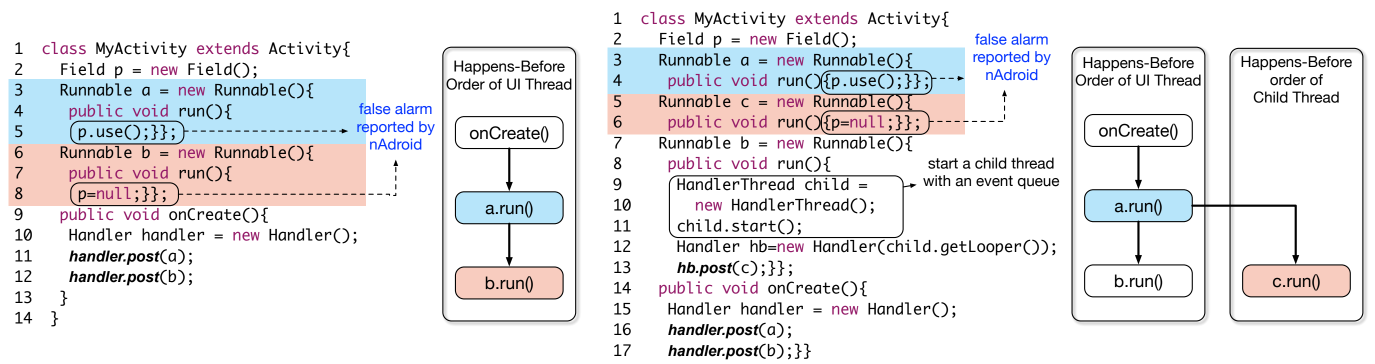
Fig. 1: Asynchronous event dispatching in Android.

every thread can create and maintain its own event queue (only one event queue for each thread), allowing events in different queues to be executed in parallel under an unbounded number of event interleavings, making it very difficult to reason about the execution orders of Android events.

Figure 1 demonstrates the non-determinism when dispatching asynchronous events from the event queue of the UI thread. The four events and their corresponding code snippets are depicted using four different colors. The five rectangle boxes represent the five states of the event queue. An arrow between two states shows a (possible) state transition.

When an Android activity (e.g., `MyActivity` in Figure 1) starts, the `onCreate()` event is automatically posted into UI thread's queue in order to launch the activity. Consequently, Android's `Looper` dispatches this event from the queue and then executes its corresponding `onCreate()` method (line 2-6) as highlighted in yellow. At lines 4-5, a new user-defined event `run()` of a `Runnable` class is added into the queue via `handler.post(...)` as highlighted in blue. Since every asynchronous event is dispatched in the FIFO order and executed atomically, `run()` is only dispatched after finishing executing the `onCreate()` method at line 6. The UI thread's queue is allowed to receive other events once after the `onCreate()` event has been processed. After dispatching `run()`, the queue becomes empty and waits for new events. As highlighted in red and green, the order of executing events `onClick()` and `onStop()` is non-deterministic. `onClick()` will be first executed if a user clicks the corresponding GUI button. `onStop()` will be executed first if the 'home' button is pressed to return back to the home screen.

Order Violations in Android. Unlike the case of the traditional thread-based concurrency model in Java, Android concurrency bugs often manifest themselves in the form of order violations [1, 2]. An Android order violation occurs


false alarm reported by nAdroid
Happens-Before Order of UI Thread
onCreate()
a.run()
b.run()
false alarm reported by nAdroid
start a child thread with an event queue
Happens-Before Order of UI Thread
onCreate()
a.run()
b.run()
Happens-Before order of Child Thread
c.run()

```
1  class MyActivity extends Activity{
2    Field p = new Field();
3    Runnable a = new Runnable(){
4      public void run(){
5      p.use();}};
6    Runnable b = new Runnable(){
7      public void run(){
8      p=null;}};
9    public void onCreate(){
10     Handler handler = new Handler();
11     handler.post(a);
12     handler.post(b);
13   }
14 }
```

(a)Intra-thread happens-before relation.

```
1  class MyActivity extends Activity{
2    Field p = new Field();
3    Runnable a = new Runnable(){
4      public void run(){p.use();}};
5    Runnable c = new Runnable(){
6      public void run(){p=null;}};
7    Runnable b = new Runnable(){
8      public void run(){
9       HandlerThread child =
10        new HandlerThread();
11      child.start();
12      Handler hb=new Handler(child.getLooper());
13      hb.post(c);}};
14   public void onCreate(){
15     Handler handler = new Handler();
16     handler.post(a);
17     handler.post(b);}}
```

(b) Inter-thread happens-before relation.

Fig. 2: Examples of happens-before relation in Android: 🟦 denotes the event with field use, 🟩 denotes the event with field free, and $\longrightarrow$ denotes a happens-before relation.

when two events access the same shared object in an incorrect order, causing unexpected program behaviors (e.g., null pointer dereferences). For example, given a pair of memory accesses, e.g., $L1 : p = null$ and $L2 : .. = p.use()$, where $L1$ in an event $e1$ should always happen after $L2$ in another event $e2$. An use-after-free (UAF) order violation occurs when $e1$ happens before $e2$, resulting in a null dereferenced (used) after p no longer points to any object (freed). UAF order violations in Android severely affect the user experience of an app, e.g., unexpectedly terminating an app or being leveraged by attackers to launch a security attack [3].

Challenges. It is challenging to find order violations in Android due to its complicated concurrency mechanism involving both Java threads and event queues. Apart from the normal order violations caused by Java threads, where the above mentioned two statements $L1$ and $L2$ reside in two Java threads, Android has two unique types of order violations. Given that $L1$ in $e1$ happens before $L2$ in $e2$, (1) an *intra-thread violation* occurs if $e1$ and $e2$ are in the same event queue, or (2) an *inter-thread violation* happens if $e1$ and $e2$ are in different parallel queues in two threads. Reasoning about these two types of order violations is challenging, since Android allows an individual thread to maintain its own event queue, which can accept asynchronous events posted from the queues of another thread. Furthermore, threads can also be created via an asynchronous event, which significantly complicates the analysis of event interleavings.

Existing Work and Limitations. Most of the existing Java-based concurrency bug detection tools [4–21] are unaware of Android events. Simply applying these tools for detecting order violations in Android works poorly due to the complicated concurrency model for dispatching non-deterministic events. Existing efforts in detecting UAF order violations mostly focus on dynamic analysis [22, 23, 1], which first collects execution traces by exercising an app at runtime through fuzzing [22, 23] or manual exploration [1]. Then an off-line detection on the collected traces is performed. Due to the nature of dynamic analysis, the dynamic race detection approaches face limited code coverage in the presence of an unbounded number of event interleavings. In addition, the existing tools usually require multiple runs to generate more traces for a more effective detection, resulting in extra runtime overheads.

Static detection of order violations will not suffer from the above mentioned limitations. However, static techniques for UAF [2, 24] detection are relatively unexplored due to the difficulty in modeling abstract asynchronous events under infinite event interleavings. The static tool SIERRA [24], includes an event-based race detection, but does not consider inter-thread order violations. A recent work NADROID [2] presents a static approach to detecting order violations by converting asynchronous events into threads and then applying a traditional data race detection tool for analyzing Java programs [4]. However, NADROID relies on coarse-grained flow- and context-insensitive event modeling, which misses happens-before (HB) relations between (1) events inside one event queue, and (2) events in different queues residing in multiple threads, causing a large number of false alarms.

Figure 2(a) gives an intra-thread false alarm reported by NADROID, which fails to capture the HB relation from event `a.run()` (blue) to `b.run()` (red) due to ignoring the program control-flows that affect the event dispatching orders. Lines 11 and 12 in `onCreate()` post two events `a.run()` and `b.run()` to the event queue with their method bodies containing a field use $p.use()$ and a field resetting $p = null$, respectively. The UAF violation (lines 5 and 8) reported by NADROID is a false alarm since `b.run()` is always dispatched after `a.run()` due to the control-flow execution order (lines 13 and 14) inside the atomic method `onCreate()` for posting the two events.

Figure 2(b) demonstrates an inter-thread false alarm reported by NADROID, which ignores calling contexts when inferring inter-thread HB orders. The HB order of `onCreate()`, `a.run()` and `b.run()` is the same as that in Figure 2(a). The only difference is that `b.run()` posts a new event `c.run()` at line 13 to the event queue of a parallel thread `child` created at lines 9-11. NADROID conservatively assumes event `c.run()` can happen in parallel with event `a.run()` without performing any analysis and reports a false order violation. However, `c.run()` in the `child` thread is posted via the callsite at line 13 in `b.run()`, which must be executed after `a.run()`. On the contrary, SARD is able to infer this strict inter-thread HB relation from `a.run()` to `c.run()` by analyzing the program control-flow from line 16 to line 17, thereby eliminating the false alarm reported by nADROID.

Our Solution. To address the aforementioned limitations, this paper presents SARD, a static approach to detect UAF order violations, the most common type of races in Android [2]. SARD systematically models Android's asynchronous events to detect both intra- and inter-thread order violations. A new flow- and context-sensitive static happens-before analysis is proposed to reason about the interleavings between events in a single and/or multiple event queues to identify precise HB relations and significantly remove spurious event interleavings. Our static happens-before relations can also be used to accelerate dynamic analysis by avoiding exercising event orders that are statically proved to be safe.

SARD performs context-sensitive analysis by distinguishing the calling contexts leading to an API call that creates or dispatches an event. Our flow-sensitive analysis precisely reasons about control-flow execution order inside an atomic method for determining event-posting orders. In addition, nADROID assumes that only one event queue for all events across all threads during their static modeling, i.e., causing unsound HB relations. SARD's modeling is more sound to be able to discover more UAFs than nADROID.

We have evaluated SARD using 27 real-world large Android apps. Experimental results show that SARD significantly outperforms nADROID, a state-of-the-art static Android order violation detection tool, in terms of both precision (by reporting three times fewer false alarms and 3 more true alarms than nADROID given the same set of apps used by nADROID) and efficiency (by running two orders of magnitude faster than nADROID).

This paper makes the following key contributions:

- We present a new static order violation detection approach by precisely reasoning about the happens-before relations between asynchronous events in Android.
- We introduce a new flow- and context-sensitive modeling of Android events from single and multiple event queues to reason about event interleavings.
- We evaluate our tool on 27 real-world Android apps. SARD significantly outperforms nADROID, the state-of-art static tool, in terms of both efficiency (175 times faster) and precision (1058 fewer false positives and 3 more true alarms).

II. MOTIVATING EXAMPLE

This section revisits the example in Figure 2(b) to demonstrate how SARD precisely extracts the intra- and inter-thread happens-before relations that are missed or incorrectly inferred by nADROID. As shown in Figure 3(b), SARD can precisely identify the five HB relations, i.e., $onCreate() \prec a.run()$, $onCreate() \prec b.run()$, $onCreate() \prec c.run()$, $a.run() \prec b.run()$, $a.run() \prec c.run()$, among which $a.run() \prec b.run()$, $a.run() \prec c.run()$ are missed by nADROID (yellow) and $b.run() \prec c.run()$ is incorrectly introduced (green).

A. *Existing Work*

The missing and incorrect HB relations produced by nADROID are due to (1) imprecise flow- and context-insensitive modeling event interleavings in an event queue, i.e., missing happens-before orders. (2) an unsound assumption that only one event queue for all events across all threads during their static modeling, i.e., causing incorrect HB relations.

nADROID first converts all the asynchronous events into traditional threads in order to leverage CHORD, a traditional multi-thread race detector for Java [4]. nADROID builds a harness main method for an Android app and creates artificial threads to invoke methods of asynchronous events in the harness main. Since CHORD is not aware of Android events, nADROID generates conservative happens-before relations of the converted threads to model Android event execution orders. nADROID's modeling is imprecise with conservative assumptions that `onCreate()` happens before all other Android events (including callbacks and runnable events) of an activity. `onDestroy()` is assumed to happen after all other callbacks. As depicted in Figure 3(b), nADROID conservatively assumes that `onCreate()` happens before `a.run()`, `b.run()` and `c.run()`, while two HB relations (yellow) are missed.

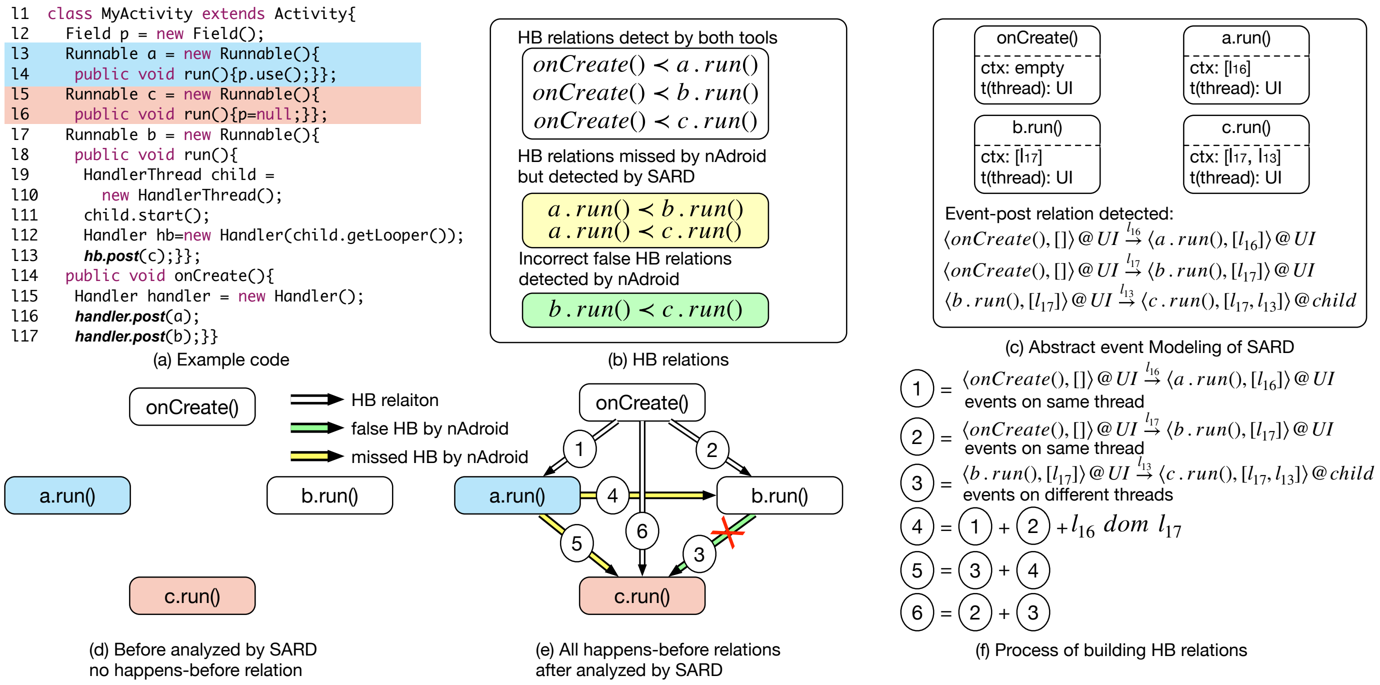
As highlighted in green, nADROID is also unsound by missing may-happen-in-parallel relations between two events on different queues in parallel threads, thereby producing incorrect happens-before orders. nADROID infers an incorrect happens-before relation between `b.run()` and `c.run()`, since nADROID assumes a single unique event queue. However, the two events actually reside in parallel event queues in different threads, i.e., UI and child threads.

B. SARD

SARD performs flow- and context-sensitive analysis that correctly handles the calling contexts and program control-flows when analyzing the event-posting and event-dispatching related Android APIs. SARD precisely models an abstract event via a calling context under an abstract thread.

SARD first performs a pre-analysis to model the four abstract events with its corresponding context information given in Figure 3(c). The context of each abstract event is a stack represented by a sequence of event-posting invocations. For example, the context of an abstract event `c.run()` is represented as $[l_{17}, l_{13}]$ created via invocations l_{17} and l_{13}.

SARD provides fine-grained modeling of events by distinguishing event queues in different threads. Abstract threads including the default UI thread and child threads at a thread creation site (e.g., l_9 - l_{10}) are modeled. SARD determines the thread where an event resides in by analyzing the Android `handler` object. For example, event `c.run()` is in the `child` thread since `c.run()` is posted to `child` via the `handler` created at line l_{12}. Figure 3(c) gives the four abstract events and their corresponding contexts, from which we can directly obtain their event-post relations. We use $\langle onCreate(), []\rangle @UI \xrightarrow{l_{16}} \langle a.run(), [l_{16}]\rangle @UI$ to denote that `onCreate()` in the event queue of UI thread posts



```
l1  class MyActivity extends Activity{
l2    Field p = new Field();
l3    Runnable a = new Runnable(){
l4     public void run(){p.use();}};
l5    Runnable c = new Runnable(){
l6     public void run(){p=null;}};
l7    Runnable b = new Runnable(){
l8     public void run(){
l9      HandlerThread child =
l10       new HandlerThread();
l11     child.start();
l12     Handler hb=new Handler(child.getLooper());
l13     hb.post(c);}};
l14   public void onCreate(){
l15    Handler handler = new Handler();
l16    handler.post(a);
l17    handler.post(b);}}
```

(a) Example code

HB relations detect by both tools
$onCreate() \prec a.run()$
$onCreate() \prec b.run()$
$onCreate() \prec c.run()$
HB relations missed by nAdroid but detected by SARD
$a.run() \prec b.run()$
$a.run() \prec c.run()$
Incorrect false HB relations detected by nAdroid
$b.run() \prec c.run()$

(b) HB relations

onCreate()
ctx: empty
t(thread): UI
a.run()
ctx: [l16]
t(thread): UI
b.run()
ctx: [l17]
t(thread): UI
c.run()
ctx: [l17, l13]
t(thread): UI
Event-post relation detected:
$\langle onCreate(), []\rangle @UI \xrightarrow{l_{16}} \langle a.run(), [l_{16}]\rangle @UI$
$\langle onCreate(), []\rangle @UI \xrightarrow{l_{17}} \langle b.run(), [l_{17}]\rangle @UI$
$\langle b.run(), [l_{17}]\rangle @UI \xrightarrow{l_{13}} \langle c.run(), [l_{17}, l_{13}]\rangle @child$

(c) Abstract event Modeling of SARD

(d) Before analyzed by SARD no happens-before relation

(e) All happens-before relations after analyzed by SARD

1 = $\langle onCreate(), []\rangle @UI \xrightarrow{l_{16}} \langle a.run(), [l_{16}]\rangle @UI$
events on same thread
2 = $\langle onCreate(), []\rangle @UI \xrightarrow{l_{17}} \langle b.run(), [l_{17}]\rangle @UI$
events on same thread
3 = $\langle b.run(), [l_{17}]\rangle @UI \xrightarrow{l_{13}} \langle c.run(), [l_{17}, l_{13}]\rangle @child$
events on different threads
4 = 1 + 2 + $l_{16}\ dom\ l_{17}$
5 = 3 + 4
6 = 2 + 3

(f) Process of building HB relations

Fig. 3: An motivating example. $\langle e_x, c_x\rangle @t_x \xrightarrow{s} \langle e_y, c_y\rangle @t_y$ means that event x with context c_x in event queue of thread t_x posts event y with context c_y to event queue at thread t_y with statement s. $s\ dom\ s'$ signifies that s dominate s' in the control-flow graph.

a.run() to the UI's event queue under the invoking statement at line l_{16}. Similarly, the other two event-posting relations are $\langle onCreate(), []\rangle @UI \xrightarrow{l_{17}} \langle b.run(), [l_{17}]\rangle @UI$ and $\langle b.run(), [l_{17}]\rangle @UI \xrightarrow{l_{13}} \langle c.run(), [l_{17}, l_{13}]\rangle @child$.

Given the abstract event modeling and event-posting relations, we will start inferring HB relations (Figure 3(f)) for detecting UAF order violations. Initially, there are no HB relations for the four events, implying that any two events may happen in parallel as illustrated in Figure 3(d).

From the first two event-posting relations ① and ② in Figure 3(c), we can infer that $\langle onCreate(), []\rangle \prec \langle a.run(), [l_{16}]\rangle$ and $\langle onCreate(), []\rangle \prec \langle b.run(), [l_{17}]\rangle$. Because SARD's modeling can distinguish the queues of different threads, i.e., b.run() and c.run() are running on different threads $UI \neq child$ as identified by ③ in Figure 3(c), resulting in a may-happen-in-parallel relation instead of an unsound HB relation reported by NADROID, which assumes a single event queue for all threads.

By considering the control-flows, NADROID determines that a.run() and b.run() are posted in order as denoted by the dominant relation $l_{16}\ dom\ l_{17}$. With this flow-sensitive information together with ① and ②, NADROID can easily infer ④, i.e., a.run() happens before b.run().

We can obtain ⑤, i.e., a.run() happens before c.run() since ④ indicates that a.run() is executed before b.run(), which has posted the c.run() for execution (③).

Since onCreate() happens before b.run(), as identified by ②, the HB relation ⑥ holds transitively for any other events (e.g., c.run()) posted by b.run() (③).

Finally, based on the HB relations in Figure 3(e), SARD can prove the absence of UAF violations in this example because the field usage at l_4 in a.run() always happens before the null pointer assignment at l_6 in c.run() according to ⑤, which is missed by NADROID, causing a false alarm.

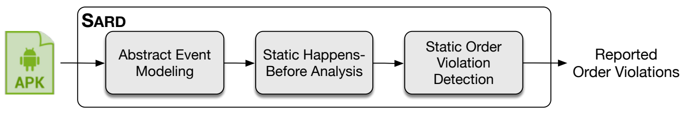

SARD
APK
Abstract Event Modeling
Static Happens-Before Analysis
Static Order Violation Detection
Reported Order Violations

Fig. 4: An overview of SARD.

III. APPROACH

In this section, we introduce the approach used in our tool SARD. Figure 4 gives an overview of SARD. SARD first models all the context-sensitive abstract events and identifies the event-posting relations. Then, SARD performs a flow- and context-sensitive analysis to infer the happens-before(HB) relations for all the abstract events. Finally, SARD detects order violations between two events based on their HB relations and the field usage in the two events. SARD applies a lightweight feasible path analysis to further prune out false alarms by identifying spurious UAF pairs.

A. Abstract Event Modeling

Abstract events are modeled context-sensitively to distinguish events under different calling contexts. A context-sensitive abstract event is denoted by $\langle e, c\rangle$, where the context $c \in \mathbb{C}$ of an event e is a stack represented by a sequence of call statements leading to the executing method of e. An abstract thread t is the thread where an abstract event resides

Three Event-Creation APIs	Abstract Event $\langle e, c\rangle @t$

handler-based
`1 Runnable a = new Runnable(){`
`2  public void run(){...}};`
`3 HandlerThread child =`
`    new HandlerThread();`
`4 child.start();`
`5 Handler handler =`
`    new Handler(child.getLooper());`
`6 handler.post(a);`
`7 handler = new Handler();`
`8 handler.post(a);`
	`e: a.run()`
	`c: [line 6]`
	`t: child(line 3)`
	$\langle a\,.\,run(), [l_6]\rangle @\,child$
	
	`e: a.run()`
	`c: [line 8]`
	`t: UI thread(null)`
	$\langle a\,.\,run(), [l_8]\rangle @\,UI$

(a) handler-based API example

activity-based
`1 Runnable a = new Runnable(){`
`2  public void run(){...}};`
`3 public void foo(){`
`4  activity.runOnUiThread(a);}`
	`e: a.run()`
	`c: [line 4]`
	`t: UI thread(null)`
	$\langle a\,.\,run(), [l_4]\rangle @\,UI$

(b) activity-based API example

thread-based
`1 Runnable a = new Runnable(){`
`2  public void run(){...}};`
`3 public void foo(){`
`4  Thread child = new Thread(a);`
`5  child.start();}`
	`e: a.run()`
	`c: [line 5]`
	`t: child(line 4)`
	$\langle a\,.\,run(), [l_5]\rangle @\,child$

(c) thread-based API example

Fig. 5: Examples of modeling abstract events based on API calls.

m: the containing method of a call statement s

$$\frac{e \textit{ is callback method}}{\langle\langle e, \emptyset\rangle @UI} \quad \text{[C-CALLBACK]}$$

$$\frac{s : \texttt{handler-based API call} \quad e = \texttt{getTgt(s)} \qquad \langle m, c\rangle \quad c' = c.append(s) \quad t = \texttt{getThread(handler)}}{\langle e, c'\rangle @t} \quad \text{[C-POST]}$$

$$\frac{s : \texttt{activity-based API call} \quad e = \texttt{getTgt(s)} \qquad \langle m, c\rangle \quad c' = c.append(s)}{\langle e, c'\rangle @UI} \quad \text{[C-UI]}$$

$$\frac{s : \texttt{thread-based API call} \quad e = \texttt{getTgt(s)} \qquad \langle m, c\rangle \quad c' = c.append(s) \quad t = \texttt{getAllocSite(thread)}}{\langle e, c'\rangle @t} \quad \text{[C-THREAD]}$$

$$\frac{s \notin \mathbb{I} \quad \langle m, c\rangle \quad c' = c.append(s)}{\langle m', c'\rangle} \quad \text{[C-CONTEXT]}$$

Fig. 6: Rules for abstract event modeling.

in. In SARD, we use the allocation sites of thread objects to represent abstract threads. If an event runs in the UI thread that does not have allocation sites in application code, we use UI to represent its abstract thread. We use $\langle e, c\rangle @t$ to denote that a context-sensitive event e runs in an abstract thread t under context c.

Each abstract event is modeled based on a sequence of event-creation API calls $s \in \mathbb{I}$. We systematically models 22 APIs which are classified into three categories (1) `handler-based`, (2) `activity-based` and (3) `thread-based`. In Figure 5, the left column gives the code examples for the three categories of event-creation APIs.

The API methods in the first category are the methods of class `Handler`, an internal class in Android. Figure 5(a) gives an example to demonstrate that two abstract events created under different contexts running on their corresponding threads via the `handler-based` API. At lines 6 and 8, `handler.post(a)` uses a handler object to post the runnable event (`a.run()`) to the event queues of the UI and child threads, respectively.

The second category creates an event via the API `runOnUiThread(runnable)` provided by Android's `Activity` class. All events created via `Activity` are posted only to the queue of UI thread. For example, at line 4 in Figure 5(b), `activity.runOnUiThread(runnable)` posts a runnable object to UI's event queue.

The last `thread-based` category contains the API methods that fork a traditional Java thread. For the statement at line 5 in Figure 5(c), it is treated as posting an asynchronous event to a special thread `child` whose queue only contains this event.

Figure 6 gives the rules to model context-sensitive abstract events by handling the three above mentioned types of API calls. Rule [C-CALLBACK] builds an abstract event for every Android callback method, which runs in the UI thread. For every callback event e, its context is $c = \emptyset$, since it is created in the Android framework but not through the event-creation APIs in the application code.

If a call statement s invokes a `handler-based` API, we apply [C-POST] to model the corresponding abstract event. `getTgt(s)` returns the corresponding event posted by s. For example, if s is $handler.post(runnable)$, `getTgt(s)` returns `run()` of the runnable object. `getThread(handler)` is used to get the abstract thread based on the object `handler` used in s. With event e, context c' and abstract thread t, we create a context-sensitive event $\langle e, c'\rangle @t$. Figure 5(a) gives two context-sensitive events created by this rule, i.e., $\langle a.run, [l_6]\rangle @child$ and $\langle a.run, [l_8]\rangle @UI$.

For a call s (e.g., `activity.runOnUiThread(a)` in Figure 5(b)) that invokes an `activity-based` API method, we only create an event running in the UI thread with its corresponding context following [C-UI]. Figure 5(b) shows an example for creating a context-sensitive event $\langle a.run, [l_4]\rangle @UI$.

Rule [C-THREAD] is applied to build an abstract event if s is a `thread-based` API call (e.g., `child.start()` in Figure 5(c)). The abstract thread t is modeled by `getAllocSite(thread)`, which finds the thread allocation site, where the thread object is created. By applying this rule, we extract a context-sensitive event $\langle a.run, [l_6]\rangle @child$ as illustrated in Figure 5(c).

For a call statement s that does not invoke any event-posting API denoted by $s \notin \mathbb{I}$, we apply [C-CONTEXT] to build new contexts on method m' invoked by s. Based on the context c of the method m, which is the containing method of s, a new context c' is created by appending s to c.

Once the abstract events are modeled, we construct an event-posting relation between different events. For an invoking statement s of event method e', if its containing

$$\frac{\langle e_x, c_x\rangle @t_x \xrightarrow{s} \langle e_y, c_y\rangle @t_y \quad t_x = t_y}{\langle e_x, c_x\rangle \prec \langle e_y, c_y\rangle}\ \text{[INTRA-POST]}$$

$$\frac{\begin{array}{c}\langle e_x, c_x\rangle @t_x \xrightarrow{s} \langle e_y, c_y\rangle @t_y \\ \langle e_x, c_x\rangle @t_x \xrightarrow{s'} \langle e_z, c_z\rangle @t_z \\ t_y = t_z \quad s\ dom\ s'\end{array}}{\langle e_y, c_y\rangle \prec \langle e_z, c_z\rangle}\ \text{[INTRA-SAME]}$$

$$\frac{\begin{array}{c}\langle e_x, c_x\rangle @t_x \xrightarrow{s} \langle e_y, c_y\rangle @t_y \\ \langle e_z, c_z\rangle @t_z \xrightarrow{s'} \langle e_w, c_w\rangle @t_w \\ \langle e_x, c_x\rangle \prec \langle e_z, c_z\rangle \quad t_x = t_z\end{array}}{\langle e_y, c_y\rangle \prec \langle e_w, c_w\rangle}\ \text{[INTRA-DIFFERENT]}$$

$$\frac{\begin{array}{c}e_x, e_z\ \mathit{are\ callback\ events} \\ \langle e_x, c_x\rangle @t_x \xrightarrow{s} \langle e_y, c_y\rangle @t_y \\ t_x = t_y \quad \langle e_x, c_x\rangle \prec \langle e_z, c_z\rangle\end{array}}{\langle e_y, c_y\rangle \prec \langle e_z, c_z\rangle}\ \text{[INTRA-INFER]}$$

$$\frac{\begin{array}{c}\langle e_y, c_y\rangle @t_y \xrightarrow{s} \langle e_z, c_z\rangle @t_z \\ \langle e_x, c_x\rangle \prec \langle e_y, c_y\rangle\end{array}}{\langle e_x, c_x\rangle \prec \langle e_z, c_z\rangle}\ \text{[COMBO]}$$

Fig. 7: Rules for detecting happens-before relations.

method could be reached in the call graph from event method e without passing through other event methods, we build $\langle e, c\rangle @t \xrightarrow{s} \langle e', c'\rangle @t'$ to denote that event e in abstract thread t under context c posts e' to the queue of thread t' under context c' using statement s.

B. Static Happens-Before Analysis

Given the abstract events, SARD builds the Happens-Before (HB) relations between two events to identify every safe use-free pair $\langle s_{use}, s_{free}\rangle$. This ensures that a field use statement s_use in one event always happens before the free statement s_free in another event.

An abstract event can be an Android callback event or a normal event (built via the three types of API calls in the application code). We first build the HB relations between Android's callback events running on the UI thread. When creating an Android Activity or a Service component, event `onCreate()` is firstly invoked before any other callback events. Therefore, we create the HB relations from `onCreate()` to every other callback event. Similarly, all other callback events happen before `onDestroy()` since it is the last to be invoked when exiting an Android component.

Next, SARD performs a flow- and context-sensitive analysis to build the HB relations (1) between two normal events or (2) between a callback event and a normal event following the rules in Figure 7. We use $\langle e_x, c_x\rangle \prec \langle e_y, c_y\rangle$ to denote an HB relation from $\langle e_x, c_x\rangle$ to $\langle e_y, c_y\rangle$, where event e_x under context c_x always happens before e_y under context c_y.

[INTRA-POST] extracts the happens-before relations between two events that have event-posting relations. For an event $\langle e_y, c_y\rangle$ posted by event $\langle e_x, c_x\rangle$, if they are in the same thread $t_x = t_y$, we build $\langle e_x, c_x\rangle \prec \langle e_y, c_y\rangle$. Let us revisit the example in Figure 3, `onCreate()` happens before both `a.run()` and `b.run()` based on this rule. Because of our precise model, we know that `b.run()` does not happen before `c.run()`, because these two events will be running on different threads.

[INTRA-SAME] defines the HB relations between two events posted by the same event $\langle e_x, c_x\rangle$ by considering the control flow information between the two events. In this rule, $\langle e_y, c_y\rangle$ and $\langle e_z, c_z\rangle$ are both posted by $\langle e_x, c_x\rangle$ using statements s and s'. We use s *dom* s' to signify that s dominates s' in the control-flow graph, i.e., every path from the method entry of event $\langle e_x, c_x\rangle$ to s' must go through s. Based on this information and the FIFO policy for an event queue, we know that $\langle e_y, c_y\rangle$ is posted before $\langle e_z, c_z\rangle$. If $\langle e_y, c_y\rangle$ and $\langle e_z, c_z\rangle$ are in the same thread, we know that $\langle e_y, c_y\rangle$ must happen before $\langle e_z, c_z\rangle$. In Figure 3, we find that `a.run()` happens before `b.run()` by applying this rule.

[INTRA-DIFFERENT] builds the HB relations for two events $\langle e_y, c_y\rangle$ and $\langle e_w, c_w\rangle$, which are posted by two different events $\langle e_x, c_x\rangle$ and $\langle e_z, c_z\rangle$, respectively. We can obtain that $\langle e_y, c_y\rangle$ happens before $\langle e_w, c_w\rangle$ if (1) $\langle e_y, c_y\rangle$ and $\langle e_w, c_w\rangle$ are in the same abstract thread, and (2) $\langle e_x, c_x\rangle$ happens before $\langle e_z, c_z\rangle$.

[INTRA-INFER] is used to model the HB relations between a callback event and a normal event posted by a callback event. We build this rule based on the Android event-driven mechanism by which the UI thread can only execute one event at a time and no other callback events (e.g., `onClick()`) can be posted to UI thread's event queue if an existing event on the UI is executing. In this rule, we can infer $\langle e_y, c_y\rangle \prec \langle e_z, c_z\rangle$ if there exists a callback event $\langle e_x, c_x\rangle$ such that (1) $\langle e_x, c_x\rangle \prec \langle e_z, c_z\rangle$ holds, i.e., the callback event $\langle e_z, c_z\rangle$ can only be posted into UI's queue after executing $\langle e_x, c_x\rangle$ and (2) $\langle e_x, c_x\rangle$ has already posted $\langle e_y, c_y\rangle$ into UI's queue. Therefore, $\langle e_y, c_y\rangle$ always happens before $\langle e_z, c_z\rangle$.

Given an HB relation $\langle e_x, c_x\rangle \prec \langle e_y, c_y\rangle$, we can easily build $\langle e_x, c_x\rangle \prec \langle e_z, c_z\rangle$ if $\langle e_z, c_z\rangle$ is posted by $\langle e_y, c_y\rangle$. Rule [COMBO] can establish intra-thread or inter-thread HB relations where $\langle e_x, c_x\rangle$ and $\langle e_z, c_z\rangle$ can be either in the same or different threads. In Figure 3, `onCreate()` and `a.run()` both happen before `c.run()` are inferred based on this rule.

C. Static Order Violation Detection

After building the HB relations for all the abstract events, we detect UAF order violations. For every event, we collect its field usage operations, i.e., field use and object free statements. SARD regards each statement that dereferences a field as a field use s_{use} and the statement that sets a field to null as a field free s_{free}. We use the alias analysis in [25] to collect every candidate pair $\langle s_{use}, s_{free}\rangle$, where two operations that access the same object and the two field usage statements s_{use} and s_{free} are from different events. The set $\mathbb{P}$ represents all the UAF pairs detected by SARD, where $s_{use} \mid\mid s_{free}$ denotes that

$$\mathbb{P} = \{\langle s_{use}, s_{free}\rangle \in \mathbb{P} \mid s_{use} \,||\, s_{free} \quad s_{free} \Rightarrow s_{use}\}$$

s_{use} and s_{free} from two happen-in-parallel events, i.e., no HB relation between these two statements can be obtained from our static HB analysis. $s_{free} \Rightarrow s_{use}$ denotes a *UAF-feasible* path satisfying that (1) the path is a control-flow feasible from s_{free} to s_{use}, and (2) there is no assignment to initialize the field object between s_{free} to s_{use} along this path.

SARD performs a light-weight path-sensitive analysis by analyzing the immediate branch conditions of two statements s_{use} and s_{free}, where the two events s_{use} and s_{free} run on the same thread.

```
public void onCreate(){
 if(field != null)
  field.use();}
```

(a)

```
public void onCreate(){
 field = new Field();
 field.use();}
```

(b)

```
public void onCreate(){
 field = getField();
 field.use();}
```

(c)

```
public void onCreate(){
 field = null;
 field = new Field();}
```

(d)

```
public void onCreate(){
 field = null;
 field = getField();
```

(e)

```
public void onCreate(){
 field = null;
 finish();}
```

(f)

Fig. 8: Infeasible path examples.

Figure 8 gives six typical examples of infeasible paths. Figure 8(a) demonstrates the situations when a null check happens before a field use statement. With such a check, the use statement will never be executed after a field free statement.

Figure 8(b) and (c) show the scenarios when a field assignment statement is executed before field use statements. In SARD, an assignment statement can be either a statement directly assigning a new value to the field or indirectly via Android's system methods (e.g., `activity.getIntent()`). We assume that assigning a value to a field by invoking any system method will initialize the field. Figure 8(d) and (e) depict the assignment statements after the free statements. All these four examples illustrate infeasible paths that can never trigger any UAF order violation due to the fact that the field object is initialized before any use.

In Android, `finish()` can be invoked to terminate an Android component. Figure 8(f) shows that if there is an invocation `activity.finish()` after a field free statement, no other events in this component can be executed, resulting in an infeasible path from s_{free} to s_{use}.

IV. EVALUATION

The objective of our evaluation is to demonstrate that SARD can effectively detect both intra- and inter-thread UAF order violations with low false alarms and high efficiency in real-world Android apps. For the same set of apps, SARD significantly outperforms NADROID [2], a state-of-the-art static analysis tool, in terms of both efficiency (by running 175 times faster than NADROID) and precision (by reporting three times fewer false alarms and identifying 3 more true alarms than NADROID). In addition, SARD also achieves a false negative rate that is two times lower than NADROID on the apps with the ground truth UAF order violations.

A. *Implementation*

SARD is built upon FLOWDROID [26], a static taint analysis for Android apps. We use FLOWDROID to decompile an Android application and then obtain all callback methods of the application. FLOWDROID uses the SPARK [27] pointer analysis in SOOT [25] framework to construct a call graph. SARD uses the call graph and the alias information provided by FLOWDROID and SPARK for our field usage analysis to support our static happens-before inference.

B. *Experimental Setup and Methodology*

In order to fairly compare SARD against NADROID, we use all the 27 real-world Android apps also used in NADROID. These applications exhibit a wide range of event usages through a wide variety of event creation APIs. Since our approach adopts a more precise flow- and context-sensitive modeling of Android concurrency system, SARD can successfully identify happens-before relations that are missed by NADROID, thereby eliminating spurious UAF violation pairs (Section III-B). SARD also applies a light-weight path-sensitive analysis to discover UAF infeasible paths to further remove more false alarms (Section III-C).

To further validate the effectiveness of SARD, we use 8 Android apps that has been manually injected with real UAF order violations to demonstrate that SARD can find UAF order violations in a low false negative rate.

Our experiments are conducted on a quad-core i5-6500 3.2GHz machine with 16GB RAM running Ubuntu 16.04 LTS. The analysis time of every app is the average of three runs. Our evaluation answers the following research questions (RQs):

- **RQ1.** Can SARD effectively and efficiently detect UAF order violations in real-world Android apps?
- **RQ2.** Does SARD perform better than NADROID a stat-of-the-art static tool in detecting UAF order violations?
- **RQ3.** Can SARD recall more manually injected UAF order violations than NADROID?

C. *RQ1: Effectiveness and Efficiency of* SARD

In this section, we evaluate the overall performance and the effectiveness of SARD in removing false alarms.

Table I illustrates the effectiveness of SARD in analyzing 27 large real-world Android apps, consisting of 537K lines of Java code in total. This table is partitioned into five parts (separated by "||"). The first part gives the information of the apps used in our evaluation, including the names of the apps and their lines of code (LOC).

The second part of Table I gives the number of *raw* potential order violations generated by SARD, which will gradually scrutinized by our precise HB analysis. A candidate violation

TABLE I: The effectiveness and efficiency of SARD in analyzing 27 real-world Android apps.

App Name	LOC	#Potential Violations	#Feasible Violations	#False Positives Eliminated by SARD							#Violations Reported	Time (secs)
				[CALLBACK RELATION]	[INTRA-POST]	[INTRA-SAME]	[INTRA-DIFFERENT]	[INTRA-INFER]	[COMBO]	Total		
SoundRecorder	1194	9	9	9	0	0	0	0	0	9	0	0.96
Swiftnotes	1571	0	0	0	0	0	0	0	0	0	0	1.02
Photoaffix	1924	379	66	38	14	0	0	1	8	61	5	17.50
MLManager	2073	64	0	0	0	0	0	0	0	0	0	11.79
InstaMaterial	2248	102	3	0	0	0	3	0	0	3	0	2.46
Tomdroid	2372	0	0	0	0	0	0	0	0	0	0	1.66
ToDoList	2637	44	10	10	0	0	0	0	0	10	0	4.47
SGT puzzle	2944	585	7	0	7	0	0	0	0	7	0	1.14
Aard	3684	718	121	0	28	0	18	0	15	61	60 (8)	15.44
Clipstack	3948	0	0	0	0	0	0	0	0	0	0	1.99
KissLauncher	5210	0	0	0	0	0	0	0	0	0	0	1.42
Zxing	6453	113	15	4	1	0	0	5	5	15	0	9.34
DashClock	10147	25	9	0	2	1	2	0	4	9	0	2.49
Dns66	10423	11	7	7	0	0	0	0	0	7	0	12.60
Music	10518	22633	3545	388	379	0	610	1159	955	3491	54	28.02
CleanMaster	11014	19	17	0	0	0	7	0	0	7	10	15.98
Omninotes	13720	2395	116	0	44	12	6	0	54	116	0	30.29
Solitair	15478	18	0	0	0	0	0	0	0	0	0	7.82
MyTracks_1	27080	2458	467	65	8	0	0	2	0	75	392 (45)	12.07
Mms	27578	2657	1066	63	4	0	0	4	78	149	917	20.91
Browser	30675	10647	788	271	84	3	118	158	147	781	7	76.26
ConnectBot	32645	150	44	30	0	0	0	0	1	31	13 (13)	80.47
MyTracks_2	37031	10894	938	55	203	152	4	3	0	417	521 (52)	62.56
MiMangaNu	37827	6	0	0	0	0	0	0	0	0	0	5.21
QKSms	56082	493	45	0	0	0	0	0	1	1	44 (28)	20.90
K9-Mail	78437	2413	255	116	41	0	0	0	20	177	78	63.81
Firefox	102658	20721	1134	6	385	7	8	0	48	454	680 (1)	66.63
Total	537571	77554	8662	1062	1200	175	776	1332	1336	5881	2781 (147)	575.19

pair generated by SARD has two parts: (1) a pair of conflict operations (i.e., a free statement in event e_x and a use statement in event e_y) on the same field object, and (2) the corresponding contexts of the two events $\langle e_x, c_x \rangle$ and $\langle e_y, c_y \rangle$.

The third column of Table I gives the afore-mentioned potential order violations in each app without applying any refinement. The fourth column gives the remaining number of violation pairs after we have applied our feasible path analysis as discussed in Section III-C.

The third part in Table I ranges from Column 5 to Column 11. The first six columns, respectively, illustrate the capabilities of each rule of our model (Figure 7) in eliminating false positives by working together to extract the HB relations in an app. Furthermore, the last column gives the total number of false positives removed by SARD's precise HB relations. For all the apps, the HB relations between only Android callback events help us eliminate a total of 1062 false pairs. Furthermore, SARD's HB relation rules [INTRA-POST], [INTRA-SAME], [INTRA-DIFFERENT], [INTRA-INFER] and [COMBO] are effective in removing 1200, 175, 776, 1332 and 1336 false pairs, respectively. In total, 5881 false alarms have been removed from the original 8662 feasible violations.

Finally, the fourth part gives the number of UAF order violations reported by SARD for each app, where the number in brackets is the true violations checked manually. In total, 2781 violations are reported by SARD and 147 of them are true UAF order violations based on our manual inspection.

Figure 9 visualizes the data in Table I. The deep blue slice in Figure 9(a) represents 75% of false positives removed by SARD HB relation rules. This part is further decomposed into six slices in the pie chart given in Figure 9(b) with each slice representing the percentage of false positives removed by each rule of our HB analysis.

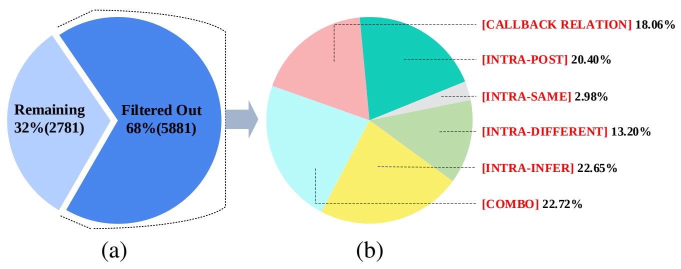


Fig. 9: Pie charts for visualizing the data in Table I.

Table I illustrates the analysis time (including call graph construction, our abstract event modeling, static HB analysis and static order violation detection) spent on each app in the last column. For the 27 Android apps, SARD spends only 21.3 seconds for each app on average and 575.19 seconds in total. This confirms that SARD is efficient in analyzing the real-world Android apps.

D. *RQ2: Comparing* SARD *with* NADROID *in Effectiveness and Efficiency*

In this section, we compare SARD with NADROID in finding UAF order violations in real-world Android apps. Since both SARD and NADROID do not find any violations in 11 apps listed in Table I, the remaining 16 apps are used in this experiment. As the order violations reported by NADROID are not differentiated by the contexts of events that have field usage operations, we merge the contexts of events among our reported violation pairs and then compare with NADROID. For effectiveness, we compare the order violations that are detected as potential violations by both tools. For efficiency, we measure the analysis time of SARD and NADROID.

TABLE II: Comparing SARD with NADROID in effectiveness and efficiency. SARD ∩ NADROID indicates the number of UAF order violations that are both reported by SARD and NADROID. SARD \ NADROID represents the number of UAF order violations reported by SARD alone and NADROID \ SARD represents the number of UAF order violations reported by NADROID alone.

App Name	**#Violations Reported**			**SARD \ NADROID**			**NADROID \ SARD**			**Time (secs)**	
	NADROID	**SARD**	**NADROID ∩ SARD**	**#True Positives**	**#False Positives**	**Total**	**#True Positives**	**#False Positives**	**Total**	**SARD**	**NADROID**
PhotoAffix	4	5	0	0	5	5	0	4	4	17.50	502.22
Aard	48	13	13	0	0	0	0	35	35	15.44	4367.66
KissLauncher	36	0	0	0	0	0	0	36	36	1.42	586.93
Zxing	2	0	0	0	0	0	0	2	2	9.34	3828.89
Dns66	13	0	0	0	0	0	0	13	13	12.60	703.58
Music	207	48	38	0	10	10	0	169	169	28.02	664.09
CleanMaster	0	6	0	0	6	6	0	0	0	15.98	8496.46
Solitaire	1	0	0	0	0	0	0	1	1	7.82	388.68
MyTracks_1	80	52	48	2	2	4	0	32	32	12.07	2708.79
Mms	312	182	120	0	62	62	0	192	192	20.91	1119.86
Browser	0	7	0	0	7	7	0	0	0	76.26	3338.98
ConnectBot	13	13	13	0	0	0	0	0	0	80.47	1265.98
MyTracks_2	71	74	56	1	17	18	0	15	15	62.56	14983.00
QKSMS	19	11	11	0	0	0	0	8	8	20.90	12603.63
K-9 Mail	336	76	20	0	56	20	0	316	316	63.81	25477.80
FireFox	468	68	31	0	37	57	0	437	437	66.63	11095.86
Total	1610	555	350	3	202	205	0	1260	1260	502.37	88303.51

Table II compares SARD and NADROID in more detail. Its second and third columns show the number of order violations reported by NADROID and SARD, respectively. For the 16 apps, SARD reports 555 violation pairs while NADROID reports 1610 ones. Among the violations detected by these tools, 350 of them are reported by both tools and these violations contain all the 88 true violations mentioned in NADROID [2]. Note that each tool reports some UAF order violations that are missed by its counterpart. We break down the 205 (1260) violations reported by SARD (NADROID) alone and manually check them to see whether they are false positives or not to illustrate the precision of SARD and NADROID.

SARD reports fewer false alarms than NADROID with a good precision. The fifth to the seventh columns of Table II illustrate the number of true positives and false positives that are reported by SARD alone in the 16 apps. After manual inspection, we found that SARD detects 202 false and 3 true violations, including 2 in *MyTracks_1* and 1 in *MyTracks_2*. All the three true UAFs are missed by NADROID.

We give an example in *MyTrack_2* in Figure 10 to illustrate a typical true UAF case missed by NADROID. NADROID fails to detect that the event `run()` executes in a different thread, since NADROID relies on some filters to remove excessive false alarms heuristically. It missed the violation pair between line 11 and line 14 due to its aggressive null check filter at line 10. However, this check does not guarantee the safety of field use at line 11.

Meanwhile, we have also checked the 1260 violation pairs that are detected by NADROID only, which are shown in the eighth to tenth columns of Table II. We found that all the 1260 ones are false alarms.

In total, SARD reports 555 violations with 464 false alarms (and 91 true errors) and NADROID reports 1610 violations with 1522 false alarms (and 88 true errors). We have found 3 more true pairs by issuing three times fewer false positives.

```
1   DataSourceManager dataSourceManager;
2   Handler handler;
3   public void onStart(){
4    dataSourceManager = new DataSourceManager();
5    HandlerThread ht = new HandlerThread();
6    ht.start();
7    handler = new Handler(myThread.getLooper());
8    handler.post(new Runnable(){
9     public void run(){
10     if(dataSourceManager!=null){
11      dataSourceManager.update();}
12    }});}
13   public void onStop(){
14    dataSourceManager = null;}
```

Fig. 10: True order violation example in *MyTracks_2*.

SARD is much more efficient than NADROID, the analysis times of SARD and NADROID for analyzing each app are shown in the last two columns of Table II. For the 16 apps used, NADROID takes 88303.51 seconds while SARD is 175 times faster using only 502.37 seconds to finish the analysis. The longer analysis time of NADROID is due to that it converts all Android's asynchronous events into native Java threads, and then applies a heavyweight race detector (e.g., CHORD) to detect potential UAF order violations in a large number of converted threads which takes a significantly long time.

E. RQ3: Comparing SARD with NADROID in False Negatives

In this section, we compare the false negative rate of SARD with NADROID in finding UAF order violations of Android apps. We use eight Android apps also used by NADROID, These apps were manually injected 28 UAF order violations, which can be seen as the ground truth of this experiment. Table III illustrates the number of violations that are recalled by SARD and NADROID.

TABLE III: Comparing SARD with NADROID in false negatives with regard to the manually injected 28 UAF order violations used in [2]. The numbers **in bold** indicate that SARD recalls more real violations than NADROID does.

App Name	#Manually Injected Ordering Violations	#Ordering Violations Detected by NADROID	#Ordering Violations Detected SARD
Aard	1	1	1
Browser	3	1	**3**
K9 Mail	1	1	1
Mms	6	4	**5**
Music	6	5	5
MyTracks_2	1	1	1
SGT Puzzles	9	8	**9**
Tomdroid	1	0	0
Total	28	21	25

For the eight apps, SARD recalls 25 real UAF order violations in total while NADROID recalls only 21 violations. For the app *Browser*, SARD finds two more order violations. For *Mms* and *SGT Puzzles*, SARD finds one more violation in each app. In total we recall 4 more order violations than NADROID, which shows that SARD has a low false negative rate (10.7%) than NADROID (25.0%) in finding UAF order violations for Android apps. We have manually checked the app code to see why SARD outperforms NADROID. For *Mms*, NADROID miss 1 more order violation than SARD because the containing method of field access statements is not reachable in its call graph. The other 3 order violations missed by NADROID in *Browser* and *SGT Puzzles* are due to that NADROID incorrectly filters them out by its unsound filters ([2], §6.2).

F. Discussion

Despite that a majority of false positives have been removed, the precision of SARD depends on its underlying pointer analysis and the feasible control-flow paths between a UAF pair.

Pointer Analysis. SARD leverages the pointer analysis SPARK [27] in the SOOT [25] framework. SPARK could only perform a conservative flow- and context-insensitive may-alias analysis, which can affect the precision of both our field usage analysis (two fields aliased in a UAF pair reported by SPARK are actually not aliased) and the interprocedural call graph (a use which is reachable from a free determined by the imprecise call graph).

Implicit Control-Flow Path. Another cause of imprecision is the implicit UAF-infeasible paths between two field usage statements. Although our light-weight path-sensitive analysis successfully prunes out a lot of false alarms by analyzing their immediate contradict branch conditions, there are still situations that SARD is not able the detect. For example, a branch condition has complicated data flow dependence to perform the null check before a field usage statement.

V. RELATED WORK

Detecting use-after-free races in Android is a new research area relative to traditional Java race detection. There are both static and dynamic approaches proposed.

Static Analysis for Android. The work most related to our static tool is NADROID [2]. It converts asynchronous events into native Java threads and leverages the Java race detector CHORD [4] to perform race detection. Section IV shows that SARD outperforms NADROID in both effectiveness and efficiency.

Recently, a static tool, SIERRA [24], has been introduced for detecting *event-based* races in Android applications. SIERRA applies an *action-sensitive* pointer analysis and builds happens-before relations between asynchronous events for *event-based* race detection. However, unlike SARD, SIERRA only includes the situation that *event-base* races happen within the same thread and SIERRA also ignores the contexts of events, which will induce different running threads and HB relations with others. Currently, the source code of SIERRA is not available yet, so we are not able to compare our tool with it.

Some other static tools also have been proposed to detect races in Android. ASYNCHRONIZER [28] is a static refactoring tool to extract long-running operations in AsyncTask (an encapsulated thread class). However, this tool focuses on AsyncTask only and is not able to detect the races caused by asynchronous events. DEVA [29] is another static tool that detects races in Android. This work is also limited for only detecting races between two callbacks without modeling of their HB relations. The limitation makes the tool suffer from significant false positives and false negatives.

Dynamic Analysis for Android. Dynamic tools [1, 23] are developed to detect races in Android at runtime. Their approaches first collect execution traces, which are generated by running Android applications on devices with their customized ROM. They then perform an off-line HB relations analysis to detect races on the collected traces. EventRacer [22] uses a novel off-line analysis algorithm to improve the scalability and precision of the previous approaches. There is also an approach ERVA [30] that is only used to verify the results of dynamic tools. Recently, the authors of [31, 32] also introduce new approaches to build the HB relations for Android apps based on execution traces. The results of the tools developed can also be leveraged to detect races. While reporting fewer false positives, dynamic tools suffers from limited code coverage and extra runtime overheads.

Race Detection for Java. There are quite a few existing approaches that can detect data races in traditional Java programs. There are static tools, based on, for example, locksets [4–7], type systems [8–10] and model checking [11]. There are also dynamic analysis tools [12–21]. These tools are not aware of Android's asynchronous events, making them ineffective in detecting UAF races in Android.

VI. CONCLUSION

In this paper, we have presented a new static tool, SARD, for detecting UAF order violations in Android apps. In SARD, we have systematically modeled the asynchronous events in Android and introduced a flow- and context-sensitive analysis to build precise happens-before relations between two events. According to our evaluation, SARD outperforms NADROID by removing its false alarms substantially and discovering its missed true violations with significantly less analysis times.

References

[1] 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*, 2014.

[2] X. Fu, D. Lee, and C. Jung, "nAdroid: Statically Detecting Ordering Violations in Android Applications," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018.

[3] *CVE-2017-0780*, https://www.cvedetails.com/cve/CVE-2017-0780/.

[4] M. Naik, A. Aiken, and J. Whaley, "Effective Static Race Detection for Java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[5] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, 2003.

[6] P. Pratikakis, J. S. Foster, and M. Hicks, "LOCKSMITH: Context-sensitive Correlation Analysis for Race Detection," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[7] 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*, 2007.

[8] C. Flanagan and S. N. Freund, "Type-based Race Detection for Java," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[9] C. Boyapati, R. Lee, and M. Rinard, "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks," in *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2002.

[10] P. Pratikakis, J. S. Foster, and M. Hicks, "LOCKSMITH: Practical Static Race Detection for C," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 1, pp. 1–55, 2011.

[11] S. Qadeer and D. Wu, "KISS: Keep It Simple and Sequential," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.

[12] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng, "RACEZ: A Lightweight and Non-invasive Race Detection Tool for Production Applications," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[13] X. Xie and J. Xue, "Acculock: Accurate and Efficient Detection of Data Races," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2011.

[14] C. Flanagan and S. N. Freund, "FastTrack: Efficient and Precise Dynamic Race Detection," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[15] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[16] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, 2005.

[17] T. Zhang, C. Jung, and D. Lee, "ProRace: Practical Data Race Detection for Production Use," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[18] T. Zhang, D. Lee, and C. Jung, "TxRace: Efficient Data Race Detection Using Commodity Hardware Transactional Memory," *SIGOPS Oper. Syst. Rev.*, vol. 50, no. 2, pp. 159–173, 2016.

[19] B. Lucia and L. Ceze, "Cooperative Empirical Failure Avoidance for Multithreaded Programs," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[20] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective Data-race Detection for the Kernel," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[21] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy, "Detecting and Surviving Data Races Using Complementary Schedules," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[22] 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*, 2015.

[23] 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*, 2014.

[24] Y. Hu and I. Neamtiu, "Static detection of event-based races in android apps," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, 2018, pp. 257–270.

[25] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework," in *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research*, 1999.

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel,

"FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[27] O. Lhoták and L. Hendren, "Scaling Java Points-to Analysis Using SPARK," in *Proceedings of the International Conference on Compiler Construction*, 2003.

[28] Y. Lin, C. Radoi, and D. Dig, "Retrofitting Concurrency for Android Applications Through Refactoring," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[29] G. Safi, A. Shahbazian, W. G. J. Halfond, and N. Medvidovic, "Detecting Event Anomalies in Event-based Systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[30] Y. Hu, I. Neamtiu, and A. Alavi, "Automatically verifying and reproducing event-based races in android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, 2016, pp. 377–388.

[31] P. Maiya and A. Kanade, "Efficient computation of happens-before relation for event-driven programs," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017, 2017, pp. 102–112.

[32] C.-H. Hsiao, S. Narayanasamy, E. M. I. Khan, C. L. Pereira, and G. A. Pokam, "Asyncclock: Scalable inference of asynchronous event causality," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, 2017, pp. 193–205.