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

Diyu Wu[1], Jie Liu[1], Yulei Sui[2], Shiping Chen[3], and Jingling Xue[1]

[1]School of Computer Science and Engineering, UNSW, Australia
[2]Faculty of Engineering and Information Technology, UTS, Australia
[3]Commonwealth Scientific and Industrial Research Organisation, Data61, Australia

***Abstract*—Unlike Java, Android provides a rich set of APIs to support a hybrid concurrency system, which consists 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 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 analysis 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 valid object, 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 with NADROID, a state-of-the-art static order violation detection tool for Android. SARD 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).**

***Index Terms*—use-after-free, order violation, data race detection, pointer analysis, static analysis**

## 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 software applications running 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 concurrency bugs (caused by both events and Java threads), which are difficult to detect, in practice.

Android execution is driven by asynchronous events in event queues. Initially, 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` will fetch the event from the queue and dispatches it by executing its asynchronous method.

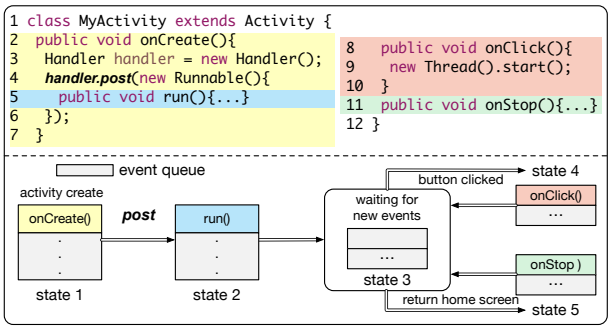



Fig. 1: Asynchronous event dispatching in Android.

***Android Event Queues***. An event queue follows the FIFO principle for dispatching events in a sequential order. However, due to the event-driven nature, both posting an event to and dispatching an event from the queue are non-deterministic, driven by user interactions. Worse still, apart from the main UI thread, Android allows Java multithreading. Every thread can create and maintain its own (unique) event queue, allowing events in different queues to be executed in parallel under an unbounded number of event interleavings. This makes it hard 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 distinguished with four different colors. The five rectangle boxes represent the five states of the event queue. An arrow between two states represents a (possible) state transition.

When an Android activity (e.g., `MyActivity` in Figure 1) starts, the `onCreate()` event is automatically posted into the 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 2-7) as highlighted in yellow. At line 5, a new user-defined event `run()` of a `Runnable` class is added into the queue via `handler.post(...)` in blue. Since every asynchronous event is dispatched in the FIFO order and executed atomically, `run()` is only dispatched after the `onCreate()` method has run to completion. The UI thread's queue is allowed to receive other events 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-

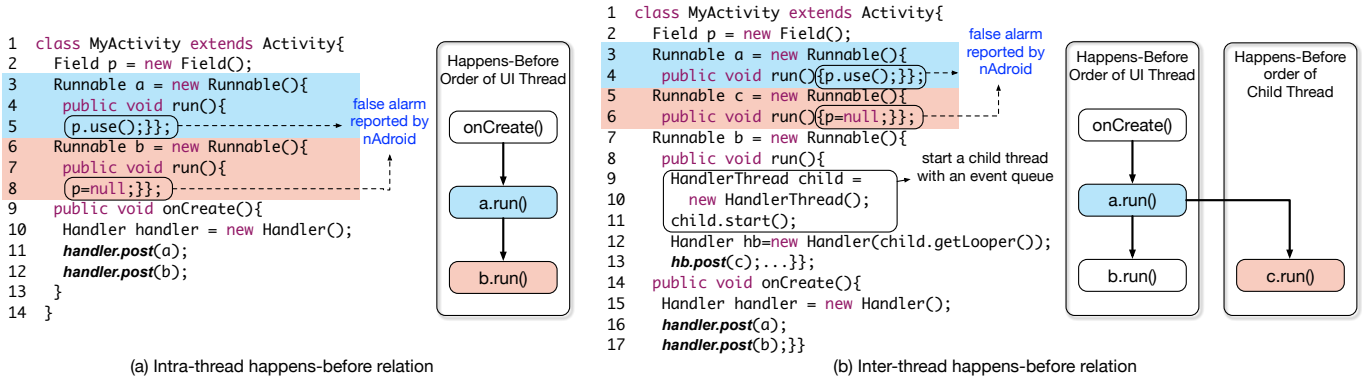



Fig. 2: Two examples illustrating happens-before relations in Android. In each example, ■ highlights an event where a field is used, ■ highlights an event where a field is freed, and ⟶ denotes the happens-before relation between the two events.

deterministic. `onClick()` will be first executed if a user clicks the corresponding GUI button. On the other hand, `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 in the Java thread-based concurrency model, Android concurrency bugs often manifest themselves in the form of order violations [1, 2]. An order violation occurs 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 dereference. UAF order violations in Android severely affect the user experience of an app, e.g., unexpectedly terminating an app or being leveraged by an attacker to launch a security attack [3].

***Challenges***. It is challenging to find order violations in Android due to its complex concurrency mechanism involving both Java threads and event queues. In addition to the normal order violations caused by Java threads, where the two statements $L1$ and $L2$ mentioned above reside in two different 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 the two distinct queues associated with two different 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 other threads. Furthermore, a thread can also be created via an asynchronous event, which can also significantly complicate the analysis of event interleavings.

***Existing Work and Limitations***. Most of the existing Java-based concurrency bug detection tools [4–22] are unaware of Android events. Simply applying these tools for detecting order violations in Android works poorly due to the complex concurrency model used for dispatching non-deterministic events. Existing efforts in detecting UAF order violations mostly focus on dynamic analysis [23, 24, 1], which first collects execution traces by exercising an app at runtime through fuzzing [23, 24] or manual exploration [1]. Then an off-line detection on the collected traces is performed. Due to the nature of dynamic analysis, dynamic race detection approaches suffer from limited code coverage in the presence of an unbounded number of event interleavings. In addition, these tools usually require multiple runs in order to generate more traces to enable an effective bug detection, resulting potentially in large runtime overheads.

Static detection of order violations will not suffer from the above mentioned limitations. However, static techniques for UAF detection [2, 25] are relatively unexplored due to the difficulty in modeling abstract asynchronous events under infinite event interleavings. SIERRA [25] includes an event-based race detection, but does not consider inter-thread order violations. NADROID [2] represents a recent 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 may miss some happens-before (HB) relations among the events in one event queue or in different event queues residing in multiple threads, causing a large number of false alarms to be issued. In addition, NADROID may also introduce spuriously HB relations unsoundly (due to its one event queue assumption), causing some bugs to be missed (Section II).

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 nullifying statement $p = null$, 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 11 and 12) 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 among `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 that `c.run()` can happen in parallel with `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 afore-mentioned limitations, this paper presents SARD, a static approach to detecting 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 the 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., yielding spurious HB relations. In contrast, SARD's modeling is more sound as it can discover more UAFs than NADROID.

We have evaluated SARD using 27 real-world Android apps. Our 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 three more true alarms given the same set of apps used by NADROID) and efficiency (by running two orders of magnitude faster).

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 apps.
- We introduce a new flow- and context-sensitive approach to modeling Android events in single and multiple event queues to reason about event interleavings.
- We evaluate our tool SARD with a set of 27 real-world Android apps. SARD significantly outperforms NADROID, a state-of-art static analysis tool for detecting UAF order violations, in terms of both precision (by reporting 1058 fewer false positives and 3 more true alarms) and efficiency (by running 175 times faster).

## 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 conservatively or inferred unsoundly (i.e., incorrectly) by NADROID [2]. As shown in Figure 3(b), SARD can precisely identify the five HB relations, $onCreate() \prec a.run()$, $onCreate() \prec b.run()$, $onCreate() \prec c.run()$, $a.run() \prec b.run()$, and $a.run() \prec c.run()$, among which $a.run() \prec b.run()$ and $a.run() \prec c.run()$ are missed by NADROID (in yellow) and $b.run() \prec c.run()$ is incorrectly introduced (in green).

### A. Existing Work

NADROID may miss some HB relations conservatively due to its imprecise flow- and context-insensitive modeling for the event interleavings and introduce some spurious HB relations unsoundly due to an unsound assumption about the existence of only one event queue for all events across all the threads.

NADROID first converts all the asynchronous events into traditional threads in order to leverage CHORD, a traditional 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. After running CHORD, NADROID then prunes the results obtained with its own modeling of HB relations, which is conservative as it assumes only that `onCreate()` happens before all other Android events (including callbacks and runnable events) of an activity and `onDestroy()` happens after all other callbacks. As shown in Figure 3(b), `onCreate()` happens before `a.run()`, `b.run()` and `c.run()` as expected but the two HB relations (in yellow) are missed.

In addition, as highlighted in green, NADROID is also unsound as it may introduce spurious HB relations between two events in two different queues of two parallel threads. NADROID infers an incorrect HB relation between `b.run()` and `c.run()`, since NADROID assumes a single unique event queue. However, the two events actually reside in the parallel event queues in two different threads, i.e., UI and child threads.

### B. SARD

SARD performs a 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 call statements leading to that event. 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 creating site (e.g., $l_9$ - $l_{10}$) are modeled. SARD determines

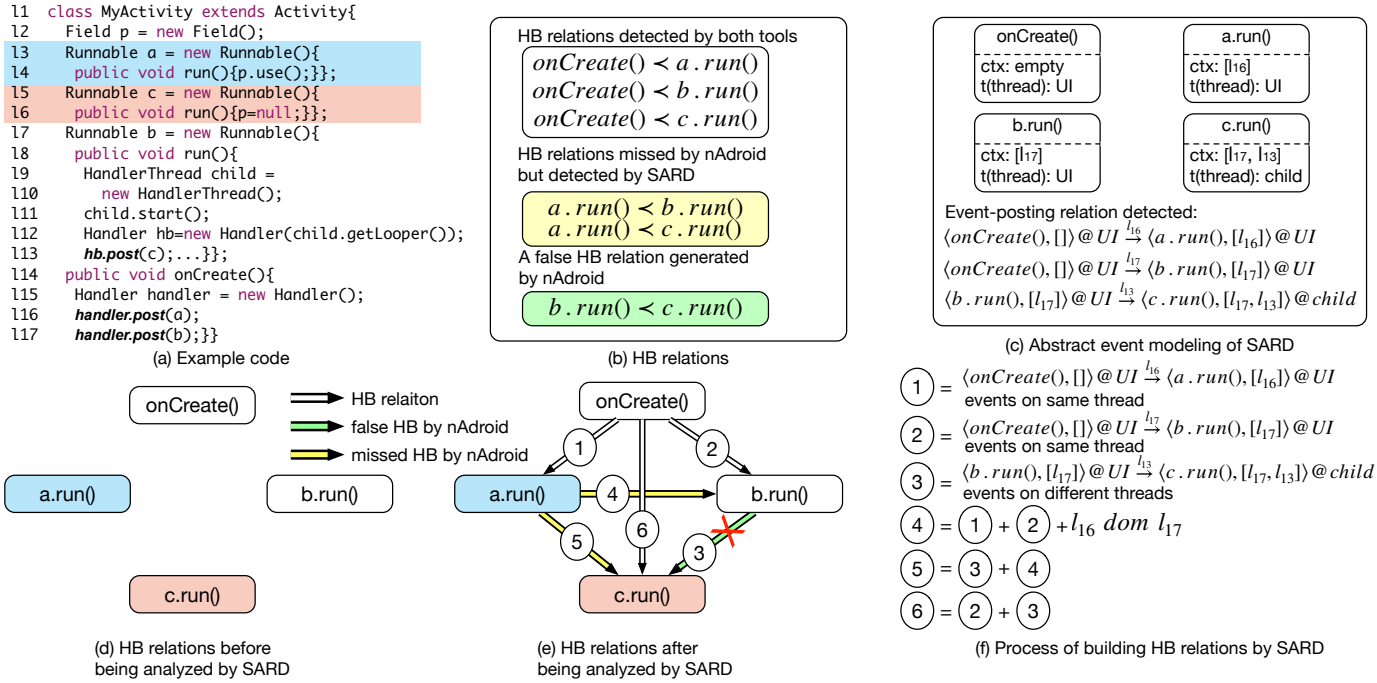



Fig. 3: A motivating example. $\langle e_x, c_x\rangle@t_x \xrightarrow{s} \langle e_y, c_y\rangle@t_y$ means that event $x$ with context $c_x$ in the event queue of thread $t_x$ posts event $y$ with context $c_y$ to the event queue at thread $t_y$ with statement $s$. If $s$ and $s'$ are two statements that appear in the inter-procedural CFG starting from a common method, then $s\ dom\ s'$ signifies that $s$ dominates $s'$ in the CFG.

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-posting relations. We write $\langle onCreate(), []\rangle@UI \xrightarrow{l_{16}} \langle a.run(), [l_{16}]\rangle@UI$ to indicate that `onCreate()` in the event queue of the UI thread posts `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 to infer the 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 in Figure 3(c), we obtain $\langle onCreate(), []\rangle \prec \langle a.run(), [l_{16}]\rangle$ and $\langle onCreate(), []\rangle \prec \langle b.run(), [l_{17}]\rangle$, which are ① and ② respectively. Because SARD's modeling can distinguish the event queues of different threads (e.g., `b.run()` and `c.run()` associated with two different threads $UI$ and $child$ as identified by ③ in Figure 3(f)), SARD can infer 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, SARD determines that `a.run()` and `b.run()` are posted in the order as denoted by the dominance relation $l_{16}\ dom\ l_{17}$. With this flow-sensitive information together with ① and ②, SARD can easily infer


Fig. 4: An overview of SARD.

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 `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 use 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.

## III. Approach

In this section, we introduce the approach used in SARD, as shown in Figure 4. 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 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 light-weight feasible path analysis to further prune out false alarms by identifying spurious UAF pairs.

| Three Event-Creating APIs | Abstract Event $\langle e, c\rangle @ t$ |
|---|---|
| **handler-based**<br>`1 Runnable a = new Runnable(){`<br>`2 public void run(){...}};`<br>`3 HandlerThread child =`<br>`new HandlerThread();`<br>`4 child.start();`<br>`5 Handler handler1 =`<br>`new Handler(child.getLooper());`<br>`6 handler1.post(a);`<br>`7 Handler handler2 = new Handler();`<br>`8 handler2.post(a);` | `e: a.run()`<br>`c: [line 6]`<br>`t: child(line 3)`<br>$\langle a.run(), [l_6]\rangle @ child$<br><br>`e: a.run()`<br>`c: [line 8]`<br>`t: UI thread(null)`<br>$\langle a.run(), [l_8]\rangle @ UI$ |

(a) A handler-based API

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

(b) An activity-based API

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

(c) A thread-based API

Fig. 5: Modeling abstract events based on API calls.

### 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 a thread where an abstract event resides 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 write $\langle e, c\rangle @ t$ to indicate 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-creating API calls $s \in \mathbb{I}$. We have systematically modeled 22 APIs falling into three categories: (1) `handler-based` APIs, (2) `activity-based` APIs, and (3) `thread-based` APIs. In Figure 5, the left column gives the code examples for the three categories.

The API methods in the first category are the methods in class `Handler`, an internal class in Android. Figure 5(a) gives an example to demonstrate that two abstract events created under different contexts running in their corresponding threads via a `handler-based` API. At line 6 (line 8), `handler1.post(a)` (`handler2.post(a)`) uses a handler object to post the runnable event `a.run()` to the event queue of the child (UI) thread.

For the second category, one can create an event via `runOnUiThread(runnable)` provided by Android's `Activity` class and its subclasses (e.g., `ListActivity`).

$m$: the containing method of a call statement $s$

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

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

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

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

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

Fig. 6: Rules for modeling abstract events.

All events created are posted only to the event queue of the UI thread. For example, at line 4 in Figure 5(b), `activity.runOnUiThread(a)` posts a runnable object to UI's event queue.

The last 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 calls of these three types of APIs. 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 some event-creating APIs in the application code.

If a call statement $s$ invokes a `handler-based` API, we apply [C-POST] to model its corresponding abstract event. $\texttt{getTgt}(s)$ returns the corresponding event posted by $s$ and $\texttt{getThread}(s)$ retrieves the abstract thread where the event runs. 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 example, if $s$ is $handler1.post(a)$ in Figure 5(a), then $\texttt{getTgt}(s)$ returns `a.run()` and $\texttt{getThread}(s)$ returns `child`.

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(s)`, which finds the thread allocation site, where the thread object at $s$ is created. By applying this rule, we extract a context-sensitive event $\langle a.run(), [l_5]\rangle @ child$ 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 for method $m'$ invoked by $s$. Based on the context $c$ of method $m$, which is the containing method of $s$, a new context $c'$ is created by appending $s$ to $c$.

### B. Static Happens-Before Analysis

Once all the abstract events have been modeled, we construct an event-posting relation between different events. For an invoking statement $s$ of event method $e'$, if its containing method can be reached in the call graph from event method $e$ without passing through other event methods, we write $\langle e, c\rangle@t \xrightarrow{s} \langle e', c'\rangle@t'$ to represent the fact that event $e$ in abstract thread $t$ under context $c$ posts $e'$ to the event queue of thread $t'$ under context $c'$ at statement $s$.

Given all the abstract events, SARD builds their HB relations to identify all safe use-free pairs $\langle s_{use}, s_{free}\rangle$. This ensures that a field use statement $s_{use}$ in one event always happens before a 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, `onCreate()` is invoked before any other callback events. Therefore, we create the HB relations from `onCreate()` to every other callback event. Similarly, `onDestroy()` happens after all other callback events, 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 HB relations between two events that have event-posting relations. For an event $\langle e_y, c_y\rangle$ in thread $t_y$ posted by event $\langle e_x, c_x\rangle$ in thread $t_x$, where $t_x = t_y$, we have $\langle e_x, c_x\rangle \prec \langle e_y, c_y\rangle$. Let us revisit the example in Figure 3, `onCreate()` happens before both `a.run()` and `b.run()` based on this rule. Due to the precision in our model, `b.run()` does not happen before `c.run()`, because these two events will be running in two 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 $\langle e_x, c_x\rangle$ at statements $s$ and $s'$, respectively. We write $s\ dom\ s'$ to indicate that $s$ dominates $s'$ in the inter-procedural CFG starting from $\langle e_x, c_x\rangle$, 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.

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

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

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

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

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

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

[INTRA-DIFF] 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 have 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 even queue during its execution. Thus, $\langle e_y, c_y\rangle$ 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 establish $\langle e_x, c_x\rangle \prec \langle e_z, c_z\rangle$ transitively if $\langle e_z, c_z\rangle$ is posted by $\langle e_y, c_y\rangle$. Therefore, [COMBO] can yield intra-thread or inter-thread HB relations, where $\langle e_x, c_x\rangle$ and $\langle e_z, c_z\rangle$ can be in either the same thread or two different threads. In Figure 3, `onCreate()` and `a.run()` both happen before `c.run()` based on this rule.

### C. Static Order Violation Detection

After having obtained the HB relations for all the abstract events, we can 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 a statement that sets a field to null as a field free $s_{free}$. With the help of the alias analysis in [26], we collect the set $\mathbb{P}_{\text{all}}$ of all candidate pairs $\langle s_{use}, s_{free} \rangle$, where two operations that access the same object and the two field usage statements $s_{use}$ and $s_{free}$ are from different events. Let $\mathbb{P}$ be the set of all the UAF pairs detected by SARD:

$$\mathbb{P} = \{\langle s_{use}, s_{free} \rangle \in \mathbb{P}_{\text{all}} \mid s_{use} \not\prec s_{free}, s_{free} \Rightarrow s_{use}\}$$

where $s_{use}$ does not happen before $s_{free}$, and $s_{free} \Rightarrow s_{use}$ denotes a *UAF-feasible* path from $s_{free}$ to $s_{use}$, such that (1) the path is control-flow feasible from $s_{free}$ to $s_{use}$, and (2) there is no assignment to initialize the underlying 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 $s_{use}$ and $s_{free}$, where the two events $s_{use}$ and $s_{free}$ run on the same thread.

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

```
1 public void onCreate(){
2  field = new Field();
3  field.use();}
```
(b)

```
1 public void onCreate(){
2  field = getField();
3  field.use();}
```
(c)

```
1 public void onCreate(){
2  field = null;
3  field = new Field();}
```
(d)

```
1 public void onCreate(){
2  field = null;
3  field = getField();
```
(e)

```
1 public void onCreate(){
2  field = null;
3  finish();}
```
(f)

Fig. 8: Commonly occurring infeasible paths.

Figure 8 gives six common 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.

Figures 8(b) and (c) show the scenarios when a field assignment statement is executed before a field use statement. 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. Figures 8(d) and (e) depict the occurrence of an assignment statement immediately after a free statement. 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 of `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 used, SARD significantly outperforms NADROID [2], a state-of-the-art static analysis tool, in terms of both efficiency (by running 175 times faster) and precision (by reporting three times fewer false alarms and identifying 3 more true alarms). In addition, SARD also achieves a false negative rate that is two times lower than NADROID on the apps with the ground truth about the UAF order violations present.

### A. Implementation

SARD is built upon FLOWDROID [27], 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 [28] pointer analysis in SOOT [26] 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 analysis.

### B. Experimental Setup and Methodology

In order to fairly compare SARD with NADROID, we use all the 27 real-world Android apps also used in NADROID. These applications exhibit a wide range of event usage patterns through a wide variety of event-creating APIs. Since our approach adopts a more precise flow- and context-sensitive modeling of the Android concurrency system, SARD can successfully identify HB 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 have 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 state-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

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 HB relations | | | | | | | #Violations Reported | Time (secs) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | [CALLBACK RELATION] | [INTRA-POST] | [INTRA-SAME] | [INTRA-DIFF] | [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 |

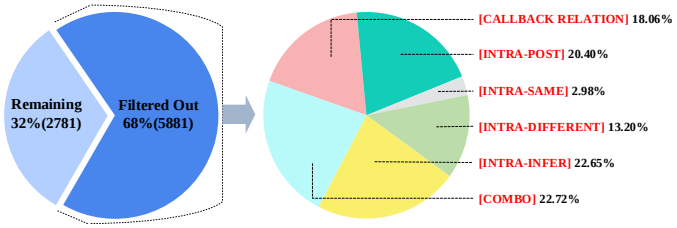
(separated by "||"). The first part gives the information about the apps used in our evaluation, including their names and lines of code (LOC).

The second part of Table I gives the number of *raw* potential order violations generated by SARD, which will be gradually scrutinized by our precise HB analysis. A candidate violation 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-DIFF], [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 numbers in brackets are the true violations checked manually. In total, 2781 violations are reported by SARD and 147 of them are true UAF order violations based on manual inspection.




(a) (b)

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

Figure 9 visualizes the data in Table I. The deep blue slice in Figure 9(a) represents 68% of false positives removed by SARD. This is further decomposed into six slices in the pie chart in Figure 9(b) with each slice representing the percentage of false positives removed by each rule in our HB analysis.

Table I also 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. SARD *vs.* NADROID: *Effectiveness and Efficiency*

In this section, we compare SARD with NADROID in finding UAF order violations in real-world Android apps. Since both do not find any violations in the same 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

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

the contexts of events among our reported violation pairs in Table I and then compare with NADROID. For effectiveness, we compare the order violations that are detected as potential violations by both tools. For efficiency, we compare the analysis times of SARD and NADROID.

Table II compares both 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. Among the violations detected, 350 are reported by both tools and this set of violations contains all the 88 true violations mentioned in NADROID [2]. Note that each tool reports some UAF order violations missed by the other. We break down the remaining 205 (1260) violations reported by SARD (NADROID) and manually check them to see whether they are false positives or not in order to understand their precision.

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 has missed the violation pair between line 11 and line 14 due to its aggressive null check filter applied at line 10. However, this check does not guarantee the safety of the field use at line 11.

Meanwhile, we have also checked the 1260 violation pairs detected by NADROID only, shown in the eighth to tenth columns of Table II. These are found to be all 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: A true order violation in *MyTracks_2*.

SARD is much more efficient than NADROID, the analysis times of SARD and NADROID for analyzing each app are given 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. NADROID is slower since it converts all Android's asynchronous events into native Java threads and then applies a heavyweight race detector (e.g., CHORD [4]) to detect potential UAF order violations in a large number of converted threads.

### *E. RQ3.* SARD *vs.* NADROID*: False Negatives*

In this section, we compare the false negative rates of SARD and NADROID in finding UAF order violations of Android apps. We use eight Android apps also used by NADROID. These apps were manually injected with 28 UAF order violations detected by a dynamic approach [24], which can be seen as the ground truth. Table III illustrates the number of violations that are recalled by SARD and NADROID.

TABLE III: Comparing SARD and 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 lower 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 the fact that NADROID incorrectly filters them out by its unsound filters ([2], §6.2).

### F. Discussion

Despite the removal of the majority of false positives, the precision of SARD depends on the precision of its underlying pointer analysis used and the precision in reasoning about the feasible control-flow paths between a UAF pair.

***Pointer Analysis***. SARD leverages the pointer analysis SPARK [28] in SOOT [26]. SPARK can only perform a conservative flow- and context-insensitive may-alias analysis, which can affect the precision of our field usage analysis and the call graph constructed for an app. Some sophistiated pointer analyses [29–34] can be used in future work.

***Implicit Control-Flow Path***. Another cause of imprecision is the implicit UAF-infeasible paths between two field use statements. Although our light-weight path-sensitive analysis has successfully pruned away a lot of false alarms by analyzing their immediate enclosing branch conditions, there are still situations where SARD is ineffective. For example, a branch condition that represents a null check may involve some complex data dependences before a field use 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 that is the most related to our static tool SARD is NADROID [2]. As discussed earlier, NADROID 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 [25], 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 handles only the *event-based* races that happen within the same thread. In addition, 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 [35] 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 [36] is another static tool for detecting races in Android apps. This work is also limited for only detecting races between two callbacks without modeling their HB relations. This limitation causes the tool to suffer from significant false positives and false negatives.

***Dynamic Analysis for Android***. Dynamic tools [1, 24] are developed to detect races in Android at runtime. Their approaches first collect execution traces, which are generated by running Android apps on devices with their customized ROM. They then perform an off-line HB relations analysis to detect races on the collected traces. EventRacer [23] uses an off-line analysis algorithm to improve the scalability and precision of the previous approaches. ERVA [37] represents an approach used to verify the results of dynamic tools. Recently, the authors of [38, 39] also introduce new approaches to building the HB relations for Android apps based on execution traces. 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, 14, 13, 15–22]. 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.

## ACKNOWLEDGEMENTS

This research is supported by an Australian Research Council Grant DP170103956.

## 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] X. Xie, J. Xue, and J. Zhang, "Acculock: accurate and efficient detection of data races," *Softw., Pract. Exper.*, vol. 43, no. 5, pp. 543–576, 2013.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] T. Tan, Y. Li, and J. Xue, "Efficient and precise points-to analysis: modeling the heap by merging equivalent automata," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 278–291, 2017.

[30] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-driven context-sensitivity for points-to analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 100, 2017.

[31] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: understanding object-sensitivity," in *ACM SIGPLAN Notices*, vol. 46, no. 1. ACM, 2011, pp. 17–30.

[32] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 1, pp. 1–41, 2005.

[33] T. Tan, Y. Li, and J. Xue, "Making k-object-sensitive pointer analysis more precise with still k-limiting," in *International Static Analysis Symposium*. Springer, 2016, pp. 489–510.

[34] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, 2016, pp. 265–266.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.