Exposing Android Event-Based Races by Selective Branch Instrumentation

Diyu Wu[1], Dongjie He[1], Shiping Chen[2], and Jingling Xue[1]
[1]School of Computer Science and Engineering, UNSW Sydney, Australia
[2]Commonwealth Scientific and Industrial Research Organisation, Data61, Australia

***Abstract*—Android supports an event dispatching system that reacts to system and user actions by generating events. However, lack of synchronization between events can lead to event-based races in Android apps. Such event-based races are difficult to detect dynamically due to the challenges faced in generating the right events to satisfy the right event-dependent conditional branches, so that their guarded racy statements can be reached. As a result, existing dynamic tools, which try to find and reschedule some race-triggering events heuristically, are often ineffective.**

We introduce SIEVE, a tool for exposing event-based races in Android apps dynamically by leveraging a new selective branch instrumentation technique. For the conditionals potentially affecting a race (detected, say, by a static tool), SIEVE fixes the true/false outcomes of some of these conditionals based on a systematic branch analysis, which analyzes the satisfiability of all the conditionals guarding the given racy statements and their safeness for instrumentation. By instrumenting certain branches selectively this way, we can not only expose effectively event-based races but also reduce substantially the negative ramifications of instrumentation (e.g., reporting non-existent races and introducing unexpected crashes during dynamic execution). An evaluation of SIEVE with 25 Android apps shows that our tool can expose event-based races more effectively than the state of the art.

***Index Terms*—event-based races, static analysis, instrumentation**

I. INTRODUCTION

As the most popular mobile operating system [1], Android includes both traditional Java multi-threading and asynchronous event dispatching in its concurrency model in order to support sophisticated tasks performed by Android apps. However, this introduces complex concurrency bugs, which are prevalent in Android apps [2].

Android's event dispatching system makes the Android operating system react to system and user actions by generating events. An event will be dispatched in order to execute a responsible (callback) method. Due to the lack of synchronization between such callback methods invoked (called *event methods*), where their execution order is non-deterministic, *event-based races* will occur [3]. In this study, we limit our focus to the races that lead to use-after-free violations, which can cause asynchronous vulnerabilities [4]. An event-based race that causes a use-after-free violation happens when a pointer is dereferenced, i.e., used after it has been freed, where its use and free operations can be reached from two event methods. Due to the non-deterministic execution order between the two event methods, the use operation may be potentially executed after the free operation. Then a null pointer exception will be thrown, causing an unexpected termination for the app. According to [5], event-based races have become a main source of concurrency bugs in Android apps, by surpassing the traditional data races between Java threads.

In order to detect the event-based races in Android apps, researchers have developed both static [3, 4, 6] and dynamic tools [5, 7, 8]. Static tools have better code coverage but higher false positive rates (due to the challenges faced in inferring precisely the pointer/alias information and happens-before relations in Android apps), while dynamic tools usually exhibit the opposite tradeoffs.

To reap the best of both worlds, one may apply first static analysis to find all the suspicious event-based races on some *racy statements* in an app and then expose as many true races as possible by performing dynamic analysis. APEChecker [9] focuses on detecting asynchronous programming errors (including some event-based races as a special case), by first detecting statically suspicious racy statements in an app, then generating program paths from the corresponding event methods to these racy statements and mapping these paths to appropriate events, and finally, dispatching these events during dynamic program execution under an appropriate environment (e.g., with a certain network connection or camera status). During their dynamic analysis, APEChecker tries to expose a suspicious event-based race as a true race by generating appropriate events to reach, i.e., exercise its corresponding racy statements. Unfortunately, racy statements, which are usually guarded by conditionals affected by some specific events, cannot be easily reached this way (due the exponential number of events that must be tried for an app).

In this paper, we focus on detecting event-based races in Android apps dynamically. To overcome the challenges faced in generating the right events to satisfy some complex event-dependent conditionals guarding racy statements, we rely on a new selective branch instrumentation technique. Given a suspicious event-based race detected by a static tool, we first collect all the conditionals that can affect the dynamic execution of its racy statements. Then, we analyze systematically each branch to decide whether it should be instrumented or not (i.e., specialized to be always taken or not). We will instrument every selected branch in order to steer the program execution towards the racy statements under consideration. Finally, we execute such an instrumented app dynamically to expose the given suspicious race by generating events appropriately.

The key novelty behind our approach is that we apply

2332-6549/20/$31.00 ©2020 IEEE
DOI 10.1109/ISSRE5003.2020.00033

a systematic branch analysis to instrument selectively some conditionals guarding a suspicious event-based race. For every such a conditional, analyzing the possible values of the variables used can reveal whether the conditional may be satisfied or not (during some program execution). This allows us to avoid introducing non-existent, i.e., infeasible execution paths caused by instrumenting unsatisfiable conditionals. In addition, we also analyze the safeness of instrumentation on every conditional, reducing unexpected crashes during dynamic execution. For some conditionals that are not suitable for instrumentation (due to crashes), our branch analysis will identify the definition statements for all the variables used and dispatch appropriate events to reach these definition statements so that these conditionals can be evaluated dynamically.

We have implemented our approach for detecting event-based races in a tool, called SIEVE (**S**elective **I**nstrumentation for **Eve**nt-based races). As the closely related tool APEChecker [9], is not open-sourced, we have evaluated SIEVE against a baseline tool (implemented by ourselves) that simply generates events to trigger the event methods in a suspicious event-based race by mimicking how APEChecker works. Our set of benchmarks consists of 25 real-world Android apps containing 190 suspicious event-based races reported by Sard, a static tool for detecting event-based races in Android apps [6], of which 18 are true races. Among all the suspicious races, SIEVE has successfully reached 90 races (47.3%) during the dynamic execution and exposed 16 true races without any false alarm, while the baseline tool has reached only 56 (29.4%) races and found only 14 true races.

In summary, this paper makes the following contributions:

- We present a novel selective branch instrumentation technique that can expose event-based races more effectively than the state of the art, with a low false positive rate.
- We introduce an effective race detection tool, SIEVE, for finding event-based races in Android apps dynamically, based on selective branch instrumentation.
- We evaluate SIEVE with 25 real-world Android apps, by demonstrating that our approach is more effective than the state of the art in detecting event-based races.

The rest of the paper is organized as follows. Section II gives some background information. Section III motivates our approach. We introduce SIEVE in Section IV and evaluate it in Section V. Section VI discusses the related work. Finally, Section VII concludes the paper.

II. BACKGROUND

First, we introduce the Android components. Then, we provide some background information about Android's event dispatching system. Finally, we describe how Android's concurrency model induces event-based races in Android apps.

A. Android Components

An Android app consists of four types of components: (1) an *Activity* lies at the heart of the Android programming framework, showing all the UI widgets in the screen. An app starts from the entry activity when opened and then reacts to external user or system actions to proceed; (2) a *Service* performs some background operations (without containing any UI widgets); (3) a *Content Provider* manages the data shared with other apps; and (4) a *Broadcast Receiver* can respond to system or user events.

B. Android's Event Dispatching System

As an event-driven operating system, Android leverages an event dispatching system to orchestrate the execution of Android apps. An *event* can be generated externally by a user or system action or internally by a thread or event executed in the app. Once generated, an event will be stored in an *event queue* waiting for event dispatching. Each event queue is maintained by a thread in the application process, which is called a *looper thread*. As long as a looper thread is running, it will continuously monitor its event queue and dispatch one event at a time to invoke the corresponding method (referred to here as *event method*). If an event is generated externally by a user or system action, then the corresponding event method triggered is also known conventionally as a *callback* (method).

C. Event-Based Races

Under an event dispatching system, the control flow of an app is dictated by a sequence of events generated. However, the execution order of their event methods is non-deterministic, since (1) user and system actions are unpredictable; and (2) events can be also generated by background threads or other events through certain APIs during their program execution.

Unless carefully handled, if two statements, reachable from two different event methods, perform use and free operations on the same object, respectively, then an event-based race may happen due to their non-deterministic execution order.

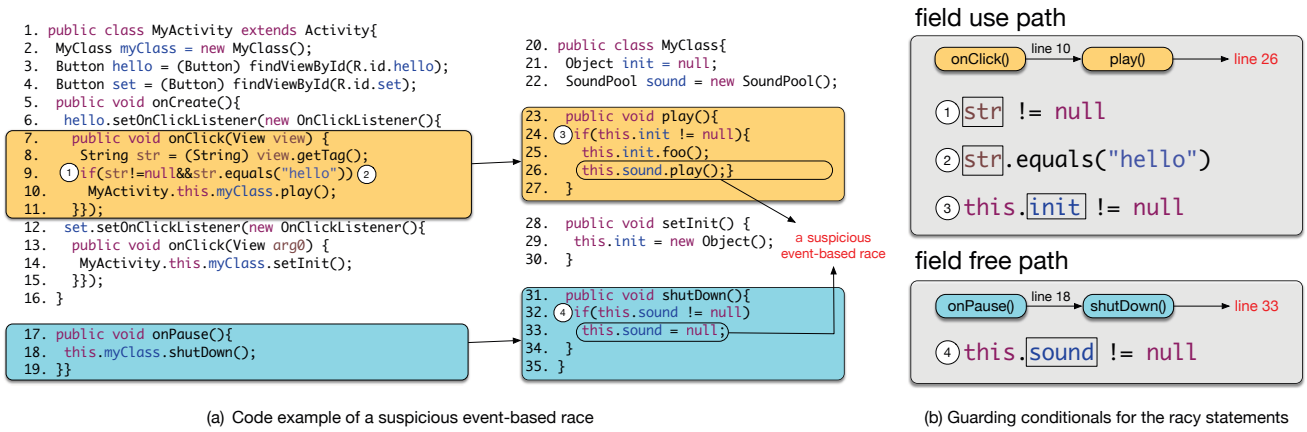
D. Happens-Before Relations

While most event methods are executed non-deterministically, some have a happens-before relation with others. If A has a happens-before relation with B, then A will always be executed before B. Our happens-before relations are built based on the rules from [3, 4, 6].

III. MOTIVATING EXAMPLE

Figure 1 depicts our motivating example. Figure 1(a) is abstracted from two real apps, PhoneFoneFun [10] and VitoshaDM [11], with a few modifications for illustration purposes. Existing static tools for detecting event-based races [3, 4, 6] may report a suspicious race between line 26 and line 33 on the field `sound` of a `MyClass` object pointing to a `SoundPool` object. However, due to their high false positive rates, static tools can only flag this as a race warning.

By inspecting the code manually, we found that this race warning is indeed a true race, since there exists two dynamic execution paths from its two corresponding event methods to the racy statements at lines 26 and 33, as shown in Figure 1(b), so that all their guarding conditionals can be satisfied. When the user clicks the `hello` button on the screen (line 3), its associated callback `onClick()` (lines 7-11) will be invoked. Under certain circumstances, `onClick()` will invoke



(a) Code example of a suspicious event-based race

(b) Guarding conditionals for the racy statements

Fig. 1: A motivating example.

`play()`, reaching the field use statement at line 26. On the other hand, an incoming call can trigger `onPause()`, which will invoke `shutDown()`, reaching the field free statement at line 33. Since the click event and incoming call event are generated asynchronously, `onPause()` can be possibly executed before `onClick()`. In this case, the field `sound` pointing to a `SoundPool` object would be first freed and then used later, causing an event-based race, and subsequently, a null pointer exception being thrown.

A. Existing Dynamic Techniques

The dynamic tool APEChecker [9], will usually fail to expose the race warning for the two racy statements at lines 26 and 33 as a true race. The main limitation is that it cannot systematically generate the events required to satisfy all the guarding conditionals in Figure 1(b) so that the two racy statements can be exercised.

APEChecker [9] will first perform static analysis to construct the two paths from `onClick()` and `onPause()` to the two racy statements at lines 26 and 33, as shown in Figure 1(b). APEChecker will then generate and execute the corresponding events so that `onClick()` and `onPause()` are executed. Targeting the conditionals, APEChecker tries to alleviate this problem by supporting lightweight modeling of system setting (e.g., network connection, camera condition) or user inputs (e.g., email, phone number). For example, if `WifiManager.isWifiEnabled()` appears in a relevant conditional, APEChecker will set the WiFi to a specific status before the app executes. However, these strategies could only affect a small part of the conditionals. In this example, APEChecker will fail to reach line 26, as its guarding conditional `this.init != null` will always evaluate to false, unless a click event is generated for the `set` button (to initialize the `init` field of a `MyClass` object).

B. Our Approach

To expose event-based races more effectively than the prior art, SIEVE will selectively instrument, i.e., specialize some guarding conditionals for racy statements (as being either always taken or always not taken) so that more racy statements can be exercised. To reduce false positives and unexpected crashes, SIEVE selects only certain conditionals for instrumentation based on a systematic branch analysis. For our example, SIEVE can successfully reach the racy statements at lines 26 and 33 and expose a true race in between.

Given the suspicious race at lines 26 and 33, SIEVE first collects their guarding conditionals listed in Figure 1(b) for the use path to line 26 and the free path to line 33. Then SIEVE performs a systematic branch analysis on each conditional to determine whether it should be instrumented or not. To avoid introducing infeasible execution paths, we need to check whether a conditional can be satisfied along its path by analyzing the possible values of all the variables appearing in the conditional. In our example, the variables that need to be analyzed are boxed in Figure 1(b).

For the two conditionals, ① and ②, the variable `str` used therein is defined as a return value of `getTag()` of a string type. We can reasonably assume that `str` may be initialized with "`hello`" so that both ① and ② can be satisfied during program execution. As for the conditional ④, `this.sound` used therein is also used in the racy statement at line 33. In that case, ④ is left unchanged in order to ensure the correct execution of the racy statement at line 33.

Determining whether the conditional ③ at line 24 can be satisfied or not during program execution is more complex, since we need to reason about the possible values taken by the field `init` of a `MyClass` object. SIEVE first finds all the definition statements of `init` to see if `init` may be assigned with a value that can make ③ satisfied. In the code snippet, `init` is initialized with a new object at line 29 that meets this requirement. Before drawing the conclusion, SIEVE will also need to inspect whether the statement at line 29 can be executed before ③. Indeed, this statement can be reached from the callback `onClick()` (lines 13-15) associated with the `set` button (line 4). Due to the lack of any happens-before relation between the two `onClick()` methods, the `onClick()` (lines 13-15) can be possibly executed before the `onClick()` (lines 7-11). However, if the latter always

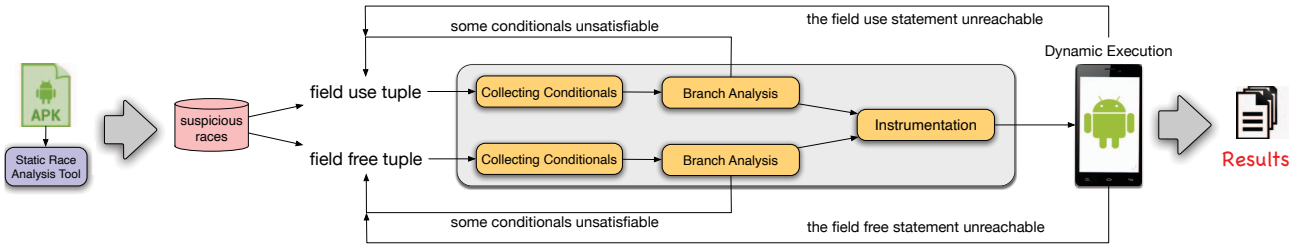


Fig. 2: Workflow of SIEVE.

happens-before the former instead, then ③ can never be satisfied, making the reported race a false positive.

After having checked the satisfiability of all the four conditionals, SIEVE will proceed to check the safeness for selective instrumentation. Regarding ③ at line 24 as always satisfiable is not safe since the statement at line 25 is data-dependent on it. Without this check, a null pointer exception may be thrown at line 25. Therefore, instead of instrumenting ③, SIEVE will opt to handle it during program execution as discussed below. The other two conditionals, ① and ②, will be specialized to true so that line 10 is always executed. As for ④, it will not be instrumented as discussed earlier.

SIEVE now executes the instrumented app to see if the given suspicious race can be exposed as a true race. For a `MyActivity` object, SIEVE generates an incoming call event to cause `onPause()` to be invoked, so that the field free statement at line 33 can be reached. To reach the field use statement at line 26, line 10 will be reached initially since ① and ② are instrumented to be always taken. To satisfy ③, SIEVE will first click the `set` button to invoke `onClick()` (lines 13-15) so that the `init` field is initialized at line 29. SIEVE will then click the `hello` button to invoke `onClick()` (lines 7-11), so that the statement at line 26 is reached.

Finally, SIEVE reports the suspicious race on the field `sound` of a `MyClass` object as a dangerous race, since a null pointer exception will be thrown if line 33 happens to be executed before line 26.

IV. SIEVE

We first give an overview of SIEVE (Section IV-A). We then describe its different analysis phases in more detail (Sections IV-B to IV-D). Finally, we discuss its dynamic execution phase (Section IV-E).

A. Overview

Figure 2 gives the workflow of SIEVE. Given an Android apk file, we first apply the static tool, Sard [6], to detect suspicious event-based races in the app. Each race warning consists of (1) a pair of racy statements s_u and s_f that perform the use and free operations on a shared object, respectively; (2) a pair of event (callback) methods e_u and e_f, which can react to user or system events and reach the racy statements s_u and s_f, respectively; and (3) a pair of context sets $\mathbb{C}_u$ and $\mathbb{C}_f$. A context $ctx = [I_1, I_2, \cdots, I_n]$ is essentially a list of invocation statements leading from an event method to the method where the corresponding racy statement resides in.

For a given race, SIEVE operates on a pair of triples, $\langle \tau_u, \tau_f \rangle$, such that $\tau_p = \langle s_p, e_p, ctx_p \rangle$, where $p \in \{u, f\}$. Here, s_p is a racy statement, e_p is the corresponding event method leading to s_p, and $ctx_p \in \mathbb{C}_p$.

For every such a triple $\langle s_p, e_p, ctx_p \rangle$, SIEVE first collects all the conditionals that may prevent the execution from reaching s_p from e_p under context ctx_p. SIEVE then applies a systematic branch analysis to each conditional to check whether it can be satisfied or not during some program execution. If there exists a conditional that can never be satisfied, SIEVE will pick a different context ctx'_p from $\mathbb{C}_p$ and start over again. If all the guarding conditionals for τ_u and τ_f can be satisfied under some contexts ctx_u and ctx_f, respectively, SIEVE will proceed to check the safeness for instrumenting each conditional. Then, SIEVE will selectively instrument certain branches by replacing each branch with true or false appropriately to steer the execution towards the racy statement guarded.

Finally, SIEVE will execute the instrumented app by generating the events as desired in order to expose the given suspicious race as a true race by observing any null pointer exception being thrown. If no race is found, SIEVE will try different contexts $\mathbb{C}_u$ and $\mathbb{C}_f$ until a race has been exposed or a pre-set time budget has been exhausted.

$$\frac{\tau : \langle s, e, ctx \rangle \quad t \in \{s\} \cup ctx}{\{t\} \subseteq \mathcal{W}} \qquad \text{[C-INITIALIZE]}$$

$$\frac{w \in \mathcal{W} \quad t \in stmt(\mathcal{M}_w) \quad t \text{ is a conditional statement} \quad t \xrightarrow{g} w}{\{t\} \subseteq \mathcal{C} \quad \{t\} \subseteq \mathcal{W}} \qquad \text{[C-COLLECT]}$$

Fig. 3: Rules for collecting conditionals.

B. Collecting Conditionals

For a triple $\langle s, e, ctx \rangle$, SIEVE collects all the guarding conditionals $\mathcal{C}$ that can affect the reaching racy statement s by applying the two rules given in Figure 3. By [C-INITIALIZE], we initialize the *worklist* $\mathcal{W}$ with the racy statement s and all the invocation statements in the list of context ctx. Afterwards, we apply [C-COLLECT] iteratively. For a conditional statement t that is in the same method $\mathcal{M}_w$ that contains a statement w in $\mathcal{W}$, if w is control dependent on t, we consider t as a guarding conditional of w denoted as $t \xrightarrow{g} w$. Then we include t to the worklist $\mathcal{W}$ and the conditional set $\mathcal{C}$.

$$\frac{c \in \mathcal{C} \quad v \in variables(c)}{\mathcal{D} = \mathcal{D} \cup \{\langle v, d\rangle \mid d \in DefStmt(v)\}} \quad \text{[T-DEFINITION]}$$

$$\frac{\langle v, d\rangle \in \mathcal{D} \quad d : x = l \quad l \text{ is a local variable}}{\mathcal{D} = \mathcal{D} \setminus \{\langle v, d\rangle\} \cup \{\langle v, d'\rangle \mid d' \in DefStmt(l)\}} \quad \text{[T-LOCAL]}$$

$$\frac{\begin{array}{c}\tau : \langle s, e, ctx\rangle \quad \langle v, d\rangle \in \mathcal{D} \quad I \in ctx \quad I \text{ invokes } \mathcal{M}_d \\ p_i \text{ is } \mathcal{M}_d\text{'s formal parameter} \\ d : x = p_i \quad a_i = argument(p_i, I)\end{array}}{\mathcal{D} = \mathcal{D} \setminus \{\langle v, d\rangle\} \cup \{\langle v, d'\rangle \mid d' \in DefStmt(a_i)\}} \quad \text{[T-PARAMETER]}$$

$$\frac{\begin{array}{c}\tau : \langle s, e, ctx\rangle \quad \langle v, d\rangle \in \mathcal{D} \quad d : x = \_.f \\ \_.f \text{ is not a field of an Android framework class} \\ \_.f \text{ is not the same field accessed by } s \\ \mathcal{F} = \{d' \mid d' \in fieldDefStmt(\_.f) \quad e_{d'} \in entries(d') \quad e \not\prec e_{d'}\}\end{array}}{\mathcal{D} = \mathcal{D} \setminus \{\langle v, d\rangle\} \cup \{\langle v, d'\rangle \mid d' \in \mathcal{F}\}} \quad \text{[T-FIELD]}$$

Fig. 4: Rules for definition tracing.

C. Branch Analysis

To support our selective branch instrumentation, SIEVE performs a systematic branch analysis for the collected conditionals in $\mathcal{C}$ to check whether they can be satisfied during program execution under their designated contexts. There are two steps, definition tracing and value range analysis.

1) **Definition Tracing:** For each conditional $c \in \mathcal{C}$, SIEVE finds the definition statements of the variables used in c by performing an inter-procedural backward tracing context-sensitively by applying the rules given in Figure 4. Let us consider [T-DEFINITION] first. For every variable v used in c, if d is a definition statement indicated by $d \in DefStmt(v)$, then $\langle v, d\rangle$ is recorded in $\mathcal{D}$. For each $\langle v, d\rangle \in \mathcal{D}$, the remaining three rules are applied iteratively.

For a definition statement $d : x = l$ of a variable v, if l is a local variable, then [T-LOCAL] will be applied (iteratively). This rule removes the original definition $\langle v, d\rangle$ from $\mathcal{D}$ and adds a set of new ones, $\{\langle v, d'\rangle \mid d' \in DefStmt(l)\}$, to $\mathcal{D}$, where d' is one of the definition statements of l.

[T-PARAMETER] is applied when p_i in the definition statement $d : x = p_i$ is a formal parameter of its containing method $\mathcal{M}_d$. Based on p_i and the invocation statement I under the given context ctx, we obtain the corresponding actual argument a_i, which is used to find its definition statement d'.

[T-FIELD] is applied when the underlying definition statement is a load $d : x = \_.f$, where f can be either a static field or an instance field. First of all, f cannot be a field of any Android framework class since it would be otherwise hard to know where its definitions are. In addition, f cannot be the same field accessed by the racy statement s, since instrumenting any conditional involving $\_.f$ will not be safe (as demonstrated with ④ in our motivating example). The function $fieldDefStmt()$ returns the set of definition statements for the field $\_.f$. In the case of an instance field $\_.f$, $x.f = \cdots$ can be potentially a definition statement if x may be aliased with the base pointer _. Given a field definition statement d', the function $entries$ returns the set of the callback methods reaching d'. To ensure that the definition statement d' can be executed before the conditional c where v is used, e must not happen-before $e_{d'}$, denoted by $e \not\prec e_{d'}$.

2) **Value Range Analysis:** Let $\mathcal{V} \cup \{\mathbf{0}\}$ be the set of values for all the variables, where $\mathbf{0} \notin \mathcal{V}$ is a special value. Conditionals containing variables that may have the special value $\mathbf{0}$ will not be instrumented later. In this value range analysis phase, SIEVE will analyze the possible values in $\mathbb{V} \in 2^{\mathcal{V} \cup \{\mathbf{0}\}}$ taken by each variable that appears in a conditional, by applying the rules given in Figure 5.

$$\frac{\langle v, d\rangle \in \mathcal{D} \quad d : x = p_i \quad p_i \text{ is a formal parameter of a callback method}}{\mathcal{V} \subseteq \mathbb{V}_v} \quad \text{[V-PARAMETER]}$$

$$\frac{\langle v, d\rangle \in \mathcal{D} \quad d : x = foo(\_)}{\mathcal{V} \subseteq \mathbb{V}_v} \quad \text{[V-METHOD]}$$

$$\frac{\langle v, d\rangle \in \mathcal{D} \quad d : x = \_.f \quad \_.f \text{ is a field of an Android framework class}}{\mathcal{V} \subseteq \mathbb{V}_v} \quad \text{[V-FRAMEWORK]}$$

$$\frac{\langle v, d\rangle \in \mathcal{D} \quad d : x = \textbf{new } T()}{\{\textbf{new } T()\} \subseteq \mathbb{V}_v} \quad \text{[V-VALUE-NEW]}$$

$$\frac{\langle v, d\rangle \in \mathcal{D} \quad d : x = constant}{\{constant\} \subseteq \mathbb{V}_v} \quad \text{[V-VALUE-CONST]}$$

$$\frac{\tau : \langle s, e, ctx\rangle \quad \langle v, d\rangle \in \mathcal{D} \quad d : x = \_.f \quad \_.f \text{ is possibly the same field accessed by } s}{\{\mathbf{0}\} \subseteq \mathbb{V}_v} \quad \text{[V-SAME]}$$

$$\frac{\langle v, d\rangle \in \mathcal{D} \quad d : x = b \in otherForm(a)}{\{\mathbf{0}\} \subseteq \mathbb{V}_v} \quad \text{[V-RESERVE]}$$

Fig. 5: Rules for value range analysis.

The rule [V-PARAMETER] is applied if the variable v is defined in terms of a parameter of a callback method. Since the values of a callback parameter can vary widely depending on system or user actions, we assume reasonably that v can take any value from $\mathcal{V}$. For similar reasons, we have also made the same assumption as when v is defined in terms of a field of an Android framework class ([V-FRAMEWORK]). As for the situation when v is defined with a return value of a method ([V-METHOD]), in most of the cases, the possible return values of this method can satisfy the conditional containing v, even if we apply heavyweight context- and path-sensitive analysis. Therefore, we apply the same assumption in this rule.

The rules [V-VALUE-NEW] and [V-VALUE-CONST] for constant initializations in both cases are self-explanatory.

The rule [V-SAME] handles the case when v is defined in terms of a field $\_.f$, where the base pointer _ is aliased with the base pointer of the field accessed by the racy statement s (as demonstrated by ④ in our motivating example). To avoid crashing the app, any conditional involving this field access will not be instrumented.

The rule [V-RESERVE] is applied if the variable v is defined in a way that is not captured by the other rules. We have observed that v is often defined locally in this case. Thus, we assume that its possible values can be obtained during program execution. No instrumentation for v is actually needed.

D. Selective Branch Instrumentation

For the set of conditionals $\mathcal{C}$, collected for $\tau : \langle s, e, ctx\rangle$ (Section IV-B), SIEVE will select appropriate conditionals to

instrument based on the value ranges collected for the variables appearing in these conditionals (Section IV-C), by applying the rules given in Figure 6.

We instrument a conditional only if it can be satisfied during some program execution. Otherwise, a race exposed along some infeasible program paths represents a false positive only.

The two rules, [S-CONST] and [S-VARIABLE], are used to check the satisfiability of a conditional c. Here, $\trianglerighteq$ denotes an abstract relational operator that represents a concrete relational operator such as $>, \neq$ and $\leq$. Given two possible value sets, $\mathbb{V}_1$ and $\mathbb{V}_2$ (with neither containing $\mathbf{0}$), $satisfy(\mathbb{V}_1, \mathbb{V}_2, \trianglerighteq)$ returns true if and only if v_1 and v_2, where $v_1 \times v_2 \in \mathbb{V}_1 \times \mathbb{V}_2$, are type-compatible and their possible values are satisfied with respect to $\trianglerighteq$. As mentioned above, if the set of possible values of any variable that appears in c includes $\mathbf{0}$, then c will not be instrumented. If a conditional c can be satisfied during program execution, it will be recorded in $\mathcal{S}$.

The rule [S-SAFENESS] ensures the safeness for instrumenting a conditional. Replacing a conditional by a fixed true/false outcome may lead to unexpected crashes or infinite loops. In addition, a guarding conditional c for $\tau : \langle s, e, ctx \rangle$ can be reached from not only e but also some other callback methods. Instrumenting c may falsely introduce some execution paths of other callbacks that should otherwise not be executed, causing unexpected crashes. Therefore, before performing instrumentation on a satisfiable conditional $c_s \in \mathcal{S}$, SIEVE also checks the safeness for doing so using this rule.

Let us now examine [S-SAFENESS] in more detail. For a statement i, whose guarding conditional is c_s, denoted as $c_s \xrightarrow{g} i$, i should not be use-dependent on c_s, denoted $c \overset{u}{\not\rightarrow} i$ (to reduce program crashes or infinite loops introduced by instrumenting c_s). In this paper, i is considered to be *use-dependent* on c_s if it uses a variable in c_s without any re-definition for the used variable in between. In addition, for every other callback e_o that can reach c_s, e_o should not happen-before e, denoted $e_o \not\prec e$ (for the purposes of reducing crashes and infinite loops introduced again).

It is important to point out that it is impossible to avoid introducing crashes or infinite loops unless we can reason about the precise intention that programmers have in mind in writing their code. The objective of [S-SAFENESS] is to reduce their occurrences in order to maximize the effectiveness of SIEVE. As evaluated in Section V, SIEVE induces a few crashes and no infinite loops for the 25 apps evaluated. Interestingly, an infinite loop introduced may still allow a race to be exposed due to the null pointer exception thrown inside the loop. Due to the potential existence of an infinite loop, a time budget is set when exercising an event (Section IV-E).

By applying the rules in Figure 6, we will obtain the set of conditionals to be instrumented in $\mathcal{I}$. Based on the branch guarding a racy statement, we instrument its outcome as true or false whichever is appropriate to ensure that the racy statement can be reached during the instrumented program execution. For the conditionals that are satisfiable but unsafe for instrumentation, we record the callback methods that make these conditionals satisfied. Those callbacks are referred to

$$\frac{c : v \trianglerighteq constant \in \mathcal{C} \quad \mathbf{0} \notin \mathbb{V}_v \quad satisfy(\mathbb{V}_v, \{constant\}, \trianglerighteq)}{\{c\} \subseteq \mathcal{S}} \quad \text{[S-CONST]}$$

$$\frac{c : v_i \trianglerighteq v_j \in \mathcal{C} \quad \mathbf{0} \notin \mathbb{V}_{v_i} \quad \mathbf{0} \notin \mathbb{V}_{v_j} \quad satisfy(\mathbb{V}_{v_i}, \mathbb{V}_{v_j}, \trianglerighteq)}{\{c\} \subseteq \mathcal{S}} \quad \text{[S-VARIABLE]}$$

$$\frac{\tau : \langle s, e, ctx \rangle \quad c_s \in \mathcal{S} \quad e_o \in entries(c_s) \quad i \in stmt(\mathcal{M}_{c_s}) \quad c_s \xrightarrow{g} i \quad c_s \overset{u}{\not\rightarrow} i \quad e_o \neq e \quad e_o \not\prec e}{\{c\} \subseteq \mathcal{I}} \quad \text{[S-SAFENESS]}$$

Fig. 6: Rules for selective branch instrumentation.

as *settable callbacks* and will be activated by generating appropriate events during dynamic execution (as described below).

Finally, there is a caveat on a semaphore mechanism used in detecting event-based races dynamically. In order to expose the race between τ_u and τ_f, we must ensure that s_f is executed before s_u. If both statements are executed in the same thread, simply invoking e_f before e_u will ensure this correct execution order as desired. However, if both statements are executed in two different threads, the desired execution order for s_f and s_u cannot be guaranteed this way. Instead, we will proceed similarly as in APEChecker [9] by using a semaphore mechanism. SIEVE inserts a P operation before the use statement s_u and a V operation after the free statement s_f. As a result, we will not move past s_u until s_f has been executed. The waiting operation here simulates a time-consuming task, without changing the behaviors of the app.

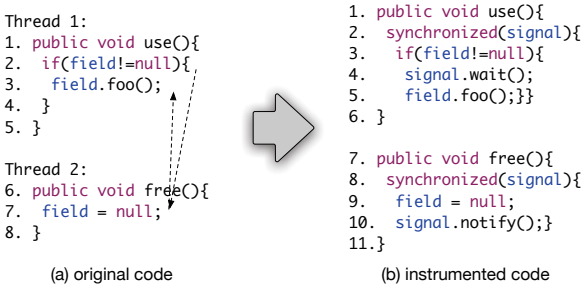


Fig. 7: An inter-thread event-based race.

Figure 7 illustrates an inter-thread event-based race. In the original code given in Figure 7(a), the statements at lines 3 and 7 are supposed to run in two different threads. In order to expose the race, the correct execution order should be line 2 $\rightarrow$ line 7 $\rightarrow$ line 3. It is not easy to achieve this order by executing sequentially `free()` and `use()` in that order. By inserting the semaphore operations as shown in Figure 7(b), `use()` should be executed before `free()` in order to maximize the chances for exposing the race at run time.

E. Dynamic Execution

SIEVE tries to expose the event-based race (if any) between s_u and s_f by executing dynamically the instrumented app on

an Android emulator. It maps the callback methods to corresponding user or system actions and simulates these actions by `Android Debugging Bridge(adb)` [12]. Currently, SIEVE focuses on modeling two main types of callback methods: (1) callback methods that react to user actions on UI widgets (e.g., `onClick()`) and (2) lifecycle callbacks of components (e.g., `onPause()`). In addition, SIEVE also inspects the information from `Android Logcat` [13] to monitor any exception raised due to an exposed race. SIEVE reports an event-based race if a null pointer exception is thrown when s_f is executed before s_u.

For each tuple $\tau_p : \langle s_p, e_p, ctx_p \rangle$, where $p \in \{f, u\}$, SIEVE generates a sequence of events $\mathcal{Q}_p$ consisting of some *navigation events*, some *settable events* and a *race event*. Navigation events are used to steer the program to the *target components* that contains e_p. SIEVE finds the callbacks that can reach the statements starting target components (e.g. `startActivity()`) and then generates the corresponding navigation events. After having reached the target component, SIEVE will send the settable events to invoke the settable callbacks described in Section IV-D and then the race event to invoke e_p. For each event in $\mathcal{Q}_p$, SIEVE currently allows their corresponding callbacks to execute for up to 2 seconds (to avoid getting stuck in an infinite loop that may be introduced, as discussed earlier in Section IV-D).

To expose the race between τ_u and τ_f, the execution order of $\mathcal{Q}_u$ and $\mathcal{Q}_f$ is determined as follows. If there is no semaphore injected during the instrumentation, $\mathcal{Q}_f$ will be executed first. Otherwise, $\mathcal{Q}_u$ will be executed first, as illustrated in Figure 7.

V. EVALUATION

Our evaluation aims to show that SIEVE can advance the state-of-the-art dynamic techniques for finding event-based races in Android apps. We address three research questions:

- **RQ1.** Can SIEVE be used as a practical tool for exposing event-based races dynamically in Android apps?
- **RQ2.** Is SIEVE more effective than existing dynamic tools?
- **RQ3.** Can SIEVE generate fewer crashes and fewer false positives by instrumenting branches selectively than fully?

A. Implementation

We have implemented SIEVE in Java (in 8K LOC) on top of several open-source tools. We rely on FlowDroid [14] to decompile an Android apk file and construct its call graph. For a given app, SIEVE operates on its Jimple intermediate representation (IR) supported by Soot [15] for performing its instrumentation and repackaging the instrumented IR into an apk file. During the execution phase, SIEVE sends commands through Android Debugging Bridge (adb) [12] to generate events (e.g., clicks and incoming calls), which can invoke the corresponding callbacks. For some of the callbacks that are triggered by interacting with some UI widgets on the screen, SIEVE utilizes Gator [16] to map the callbacks to the ids of the UI widgets. This allows us to select the correct UI widgets to perform appropriate operations (e.g., clicks) as desired. Finally, we make use of Logcat [13] to monitor the exceptions thrown during the execution in order to expose the exception-causing races in an Android app.

B. Methodology

We have selected a set of 25 real-world Android apps from F-Droid [17], a repository of open-source Android apps. These apps are obtained based on two selection criteria: (1) no user credentials are required, since the human efforts will otherwise be needed during the dynamic execution; and (2) no unsupported events are required to reach the target components or racy statements, since SIEVE currently supports for mapping parts of the callback methods to user or system events only. There are 190 suspicious races reported in the 25 apps by Sard [6]. Of the 190 race warnings reported, 18 are true races and 172 are false positives, found by manual code inspection.

We address RQ1 – RQ3 as follows. To evaluate RQ1, we analyze SIEVE's recall and false positive rates, together with its analysis times. To evaluate RQ2, we compare SIEVE with APEChecker, a state-of-the-art dynamic tool [9], which does not perform any branch instrumentation. However, as APEChecker is not open-sourced, we will evaluate SIEVE against a version of SIEVE, named SIEVE-ZB, that simply generates events to trigger the event methods in a suspicious event-based race with the aforementioned semaphore mechanism by mimicking how APEChecker works. By inspecting manually the 190 suspicious races in the 25 apps, we found that APEChecker does not perform any instrumentation on any of their guarding conditionals as these conditionals cannot be handled by its lightweight model (Section III-A). Thus, SIEVE-ZB is expected to behave similarly as APEChecker for the 25 apps considered. To evaluate RQ3, we compare SIEVE with a version, named SIEVE-FB, that instruments all the guarding conditionals for a given pair of racy statements.

Our evaluation is conducted on a quad-core i5-6500 3.2GHz machine with 16GB RAM running Ubuntu 16.04 LTS. SIEVE executes an app on a Nexus 4 emulator with API level 28. The analysis time for each app is the average of three runs.

C. Results and Analysis

Table I presents our results, divided into five parts separated by "||". Part 1 gives the information about each app, including its name, its LOC and the number of suspicious races reported statically. In Column 3, the number x inside (x) for an app is the number of true races in the app. Parts 2 – 4 contain the information for RQ1 – RQ3, respectively. Part 5 gives the time taken by SIEVE for analyzing each app.

1) **RQ1. Practicality:** SIEVE has succeeded in exposing 16 out of 18 true races in the 25 apps (achieving a recall rate of 88.9%) without producing any false positive. SIEVE has spent a total of 9435.6 seconds (about 2.6 hours) on analyzing all the 25 apps, with an average of 377.4 seconds (6.3 minutes) per app. Below we analyze the results given in Column "RQ1" to demonstrate the practicality of SIEVE.

TABLE I: Comparing SIEVE (selective branch instrumentation) with SIEVE-ZB (no branch instrumentation) and SIEVE-FB (full/all branch instrumentation) in analyzing 25 Android apps containing 190 race warnings (including 18 true races).

App	LOC	#Suspicious Races Found Statically (True Races)	RQ1 SIEVE: Selective Instrumentation				RQ2 SIEVE-ZB: Non-Instrumentation			RQ3 SIEVE-FB: Full-Instrumentation			Time for SIEVE (secs)
			#Infeasible Races	#Unreached Races	#Crashes	#Reached Races (Reported Races)	#Unreached Races	#Crashes	#Reached Races (Reported Races)	#Unreached Races	#Crashes	#Reached Races (Reported Races)	
AsciiCam	2808	2	0	1	0	1	1	0	1	1	1	1(1)	66.2
CMIS Browser	12356	1	0	1	0	0	1	0	0	1	0	0	58.7
Cool Mic	3083	11	0	11	0	0	11	0	0	11	2	0	423.0
Duorem	3728	2	0	0	0	2	2	0	0	0	0	2	98.9
GTFSOffline	4659	2	0	2	0	0	2	0	0	1	1	1(1)	78.1
Ithaka Game	2187	1	0	0	0	1	1	0	0	1	1	0	41.1
KIBA	2069	1(1)	0	0	0	1(1)	0	0	1(1)	1	1	0	39.5
KitchenTimer	1236	5	0	5	0	0	5	0	0	5	2	0	218.3
Mitzuli	9058	13	2	11	0	0	13	0	0	13	10	0	958.5
MouseApp	5856	4	0	0	0	4	0	0	4	0	0	4	150.8
Nounours	5814	2	0	2	0	0	2	0	0	2	1	0	118.5
Nusic	8520	6	0	0	0	6	0	0	6	0	0	6	434.2
Patolli	1886	13	0	13	2	0	13	0	0	13	7	0	419.1
PhoneFoneFun	623	4	0	1	0	3	2	0	2	1	0	3	215.3
Rick App	1010	6	0	5	0	1	6	0	0	5	0	1	232.2
Ringdroid	7819	26	0	6	2	20	25	0	1	5	5	18(5)	1202.7
SicMu Player	5568	18(2)	15	3	0	0	18	0	0	11	8	7(7)	382.1
SliderSynth	3015	5	0	4	0	1	4	0	1	5	1	0	521.6
Speech Trainer	1851	12	0	6	0	6	6	0	6	12	0	0	984.3
Tallyphant	1303	22	2	0	0	20	8	0	14	2	2	20	892.2
Vector Pinball	9094	23(13)	0	8	3	15(13)	8	0	15(13)	8	6	15(13)	1237.7
Vitosha DM	390	2(2)	0	0	0	2(2)	2	0	0	0	0	2(2)	62.7
Weechat	9264	5	0	1	0	4	3	0	2	1	0	4	318.2
YucataEnvoy	3641	1	0	1	0	0	1	0	0	1	0	0	35.6
Zxing	29204	3	0	0	0	3	0	0	3	1	1	2	246.1
Total	136042	190(18)	19	81	7	90(16)	134	0	56(14)	104	49	86(29)	9435.6

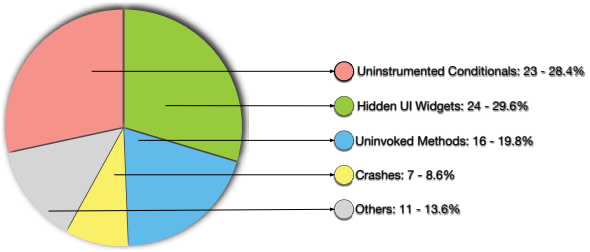


Fig. 8: Different causes for unreached races.

In Column "#Infeasible Races" for RQ1, we give the number of races that are doomed to be infeasible for each app by our branch analysis. A suspicious race is *infeasible* if one of its guarding conditionals can never be satisfied under any given context ([S-VARIABLE] and [S-CONST]). For infeasible races, no further dynamic analysis is necessary. In total, 19 infeasible races are found.

In Column "#Unreached Races" and Column "#Crahses" for RQ1, we give the number of unreached races and crashes. We have manually inspected the source code of the 17 apps containing unreached races (totaling 81). The reasons behind and their percentage contributions are depicted in Figure 8.

- **Uninstrumented Conditionals.** SIEVE instruments selectively some guarding conditionals for a suspicious race conservatively in order to reduce crashes and false positives. A total of 23 unreached races fall into this category, since some of their guarding conditionals (not instrumented by our rules in Figure 6) are not satisfied dynamically.
- **Hidden UI Widgets.** For an event method reaching the racy statement of a suspicious race, its corresponding UI widget can only be shown under certain conditions. Currently, SIEVE models this only partially (e.g., a button in a popup-dialog), resulting in 24 suspicious races not to be reached. Within this category, we have missed 2 true races in `SicMu Player`, which are the only two missed in the 25 apps.
- **Uninvoked Methods.** Some of system methods can only be triggered by the Android system. Specifically, if the statement `unbindService()` is called within callback methods, `onServiceDisconnected()` can be invoked by the system under the condition that the process hosting the service has crashed or has been killed by the Android system. Currently, SIEVE cannot control the invocation of `onServiceDisconnected()`. As a result, there are 16 unreached races caused due to this reason.
- **Crashes.** There are 7 crashes listed in Column "#Crashes", causing 7 unreached races. This can happen, as described in Section IV-D, since SIEVE makes its decisions to instrument a conditional by reasoning about its related control and data dependences, without understanding the programmers' intended correlation between a conditional and its guarded racy statement(s). How to reason about implicit information flow [18] remains to be a challenging problem. Note that we have not observed any infinite loops introduced by SIEVE.
- **Others.** The remaining 11 unreached races happen (as desired), due to the correct handling of these races in the code by having either avoided a race by using synchronization or handled a race by catching the run-time exception thrown.

In Column "#Reached Races" for RQ1, we give the number of reached races, together with the number of dangerous races reported by SIEVE (in parentheses), for each app. If SIEVE can successfully reach the two racy statements of a suspicious race in a correct order (with the free statement preceding the use statement), then the race is considered to be reached. For a reached race, if the app throws a null pointer exception, SIEVE will report a dangerous race.

In general, for the 190 suspicious event-based races reported statically in the 25 apps, SIEVE has successfully reached 90 (47.3%) of them and reported 16 dangerous races, which are all true races. Among the remaining 100 races, which consist of 19 infeasible races and 81 unreached races, we only miss 2 true races due to hidden UI Widgets as described earlier. This

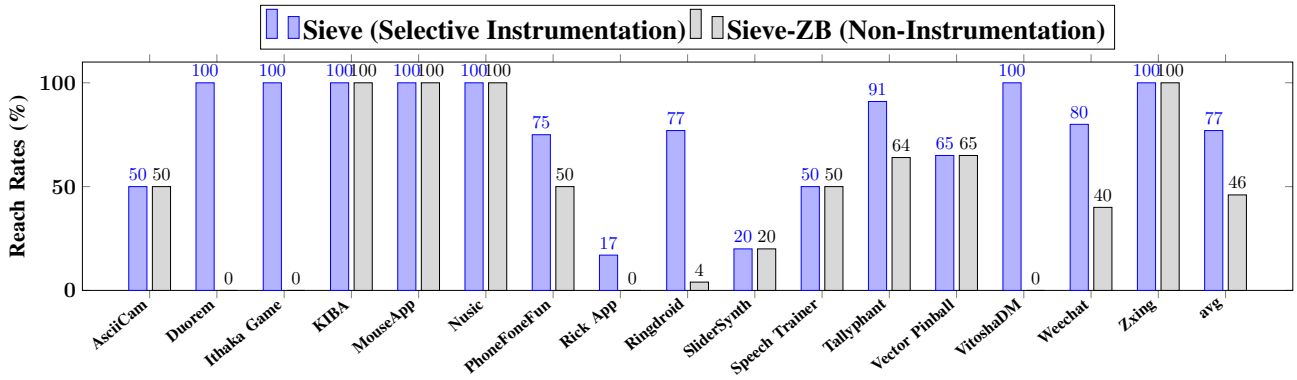


Fig. 9: Comparing SIEVE and SIEVE-ZB in terms of their reach rates for 16 apps.

VitoshaDM

```
1.  public void onCreate(){
2.   this.rollMode = false;
3.   this.sounds = new SoundPool();
4.   Timer timer = new Timer();
5.   timer.schedule(new TimerTask(){
6.    public void run(){
7.     roll();
8.    }}, 0, 100);}

9.  public void roll(){
10.  if(rollMode){
11.   this.sounds.play();
12.   this.rollMode = false;}}

13. public void onClick(){
14.  this.rollMode = true;}
15. public void onDestroy(){
16.  this.sounds = null;}
```

Fig. 10: A true race missed in `VitoshaDM` by SIEVE-ZB but exposed by SIEVE.

Ringdroid

```
1.  public void onCreate(){
2.   this.mPlayer = null;
3.   this.mIsPlaying = false;
4.   load();}

5.  public void load(){
      ...
6.   update();
7.   mPlayer = new SamplePlayer();}

8.  public void update(){
9.   if(mIsPlaying) → if(true)
10.   mPlayer.getCurrentPosition();}

11. public void onClick(){
      ...
12.  update();}
```

Fig. 11: A false positive in `Ringdroid` reported by SIEVE-FB but avoided by SIEVE.

demonstrates that SIEVE with selective branch instrumentation can effectively expose event-based races.

2) RQ2. Improvement: We assess how SIEVE advances the state of the art [9] in detecting event-based races dynamically by comparing SIEVE with SIEVE-ZB (a version of SIEVE without performing any branch instrumentation).

Let us now analyze our results in Column "RQ2" to see how much more effective that SIEVE is than SIEVE-ZB. Due to the lack of branch analysis, SIEVE-ZB reports no infeasible races at all. By performing no branch instrumentation, SIEVE-ZB reaches only 56 (29.4%) of the 190 suspicious races in the 25 apps. With selective branch instrumentation, SIEVE has reached 90 suspicious races, representing an increase of 60.7% and reports 2 more true races in `Vitosha DM`.

Figure 10 illustrates a true race in `VitoshaDM` missed by SIEVE-ZB but exposed by SIEVE. This race happens between lines 11 and 16. On executing `onCreate()`, a `TimerTask` will be scheduled to run in a background thread (lines 5-8), which invokes `roll()` every 100 milliseconds. During the iterative execution of `roll()`, if `onDestroy()` is invoked, `this.sounds` will be freed at line 16, causing a null pointer exception to be thrown at line 11. SIEVE-ZB has missed this race, since it can only reach line 11 after `this.rollMode` has been set to true earlier by `onClick()`. However, this did not happen as it failed to generate an event to invoke `onClick()` that way. In contrast, SIEVE has exposed this race successfully. Due to our branch analysis, the conditional at line 10 will be instrumented (as always taken). During our dynamic execution, assisted by the semaphore mechanism discussed in Section IV-E, SIEVE has exposed this race as a true race (due to a null pointer exception observed).

Figure 9 also compares SIEVE and SIEVE-ZB in terms of a so-called reach rate for 16 apps (excluding the 9 apps without any reached races found by both tools). For each tool, its *reach rate* for an app is the percentage of the reached races found by the tool over the total number of suspicious races in the app. For the 16 apps compared, SIEVE-ZB always achieves no better reach rates, obtaining strictly lower reach rates than SIEVE in 8 apps with 0 reach rates in `Duerem`, `Ithaka Game`, `Risk App` and `VitoshaDM`. As a result, SIEVE has found 2 true races in `VitoshaDM` that are missed by SIEVE-ZB. On average, SIEVE's reach rate is 77% but SIEVE-ZB's is only 46% per app. This has allowed SIEVE to expose more true races than SIEVE-ZB successfully.

3) RQ3. Selective Instrumentation: We demonstrate the advantages of SIEVE by comparing it with SIEVE-FB, a modified version of SIEVE. Like SIEVE, SIEVE-FB also performs instrumentation on conditionals to steer to execution towards desired branches and racy statements. Unlike SIEVE, however, SIEVE-FB instruments all the guarding conditionals for a suspicious race (i.e., with full branch instrumentation). With selective branch instrumentation, SIEVE exposes the 16 true races with no false positives reported. With full branch instrumentation, SIEVE-FB has only exposed 15 true races but issued 14 false alarms (with a false positive rate of 48.3%).

Let us now analyze our results given in Column "RQ3". Like SIEVE-ZB, SIEVE-FB also does not discover any infeasible races at all (due to the lack of branch analysis). There are two reasons why SIEVE is more effective than SIEVE-FB:

- **False Positives.** SIEVE-FB is significantly less precise than SIEVE in terms of the percentage of reached races established through infeasible paths. Among the 86 reached races found by SIEVE-FB (4 fewer than that found by SIEVE), the program paths leading to many reached races are infeasible. This is reflected by the fact that SIEVE has reported 16 races

with no false positives at all but SIEVE-FB has reported 29 races including 14 false positives.

- **Program Crashes.** By instrumenting all guarding conditionals for a suspicious race indiscriminately without considering safeness, SIEVE-FB has induced significantly more crashes than SIEVE: 49 in 15 apps for SIEVE-FB relative to 7 in 3 apps for SIEVE. Note that SIEVE-FB has reached fewer races than SIEVE in `Ithaka Game`, `Ringdroid`, `SliderSynth`, `Zxing` and missed one true race in `KIBA` due to more exceptions thrown. In `Speech Trainer`, SIEVE-FB reaches fewer races without any crash due to the infinite loop introduced by instrumentation.

Figure 11 illustrates a false positive in `Ringdroid` reported by SIEVE-FB but avoided by SIEVE, on `mPlayer` between lines 2 and 10. During the normal execution starting from `onCreate()`, `mPlayer` is assigned with a new object at line 7 after being nullified at line 2. Therefore, invoking `onClick()` later will never cause a null pointer exception at line 10. However, with full branch instrumentation, SIEVE-FB will consider the conditional at line 9 as being always taken. On executing `onCreate()`, lines 2 and 10 will be reached in that order, causing a null pointer exception at line 10. This is then reported as a race SIEVE-FB. As for SIEVE, when analyzing the conditional at line 9 reachable from `onClick()`, it finds that the conditional can also be reached from `onCreate()`, which happens before `onClick()`. According to [S-SAFENESS] in Figure 6, SIEVE will not instrument this conditional (since it is unsafe), avoiding the false positive reported by SIEVE-FB.

D. Limitations

SIEVE can be improved along a number of directions. First, SIEVE's branch analysis is developed on top of Spark [19], a pointer analysis framework provided in Soot [15]. Spark only performs a flow- and context-insensitive may-alias analysis conservatively and handles Java reflection partially, affecting the precision of our branch analysis. This can be improved by incorporating more powerful pointer analyses [20–22] and reflection analyses [23, 24].

Second, SIEVE is conservative in estimating the values of certain variables (e.g., those obtained from operations on primitive types as in [V-RESERVE] in Figure 5). This can be improved by leveraging an SMT solver such as Z3 [25]. Although SIEVE has achieved a high recal rate of 88.9% for the 25 apps evaluated, a more precise value range analysis will enable us to instrument more conditionals, and consequently, reach and expose more true races in more apps.

Finally, during the dynamic execution phase, SIEVE cannot yet trigger all callback methods or system methods as discussed in Section V-C1. This can be enhanced with more engineering efforts.

VI. RELATED WORK

Traditional Data Race Detection. Data race detection for multi-threaded programs (especially Java programs) has been investigated extensively. Many approaches exist, by exploring type systems [26, 27], lock-sets [28–30], May-Happen-in-Parallel (MHP) relations [31, 32], dynamic instrumentation [33–38], and runtime mitigations [39, 40]. However, these techniques cannot be directly applied to detect event-based races in Android apps effectively, as they are not developed to support Android's concurrency model.

Event-based Race Detection. A number of techniques have been proposed for finding event-based races. WebRacer [41] and EventRacer [8] detect such races in web applications. In the case of Android apps, dynamic tools, such as EventRacer (its Android version) [8], CAFA [7] and DroidRacer [5], execute an app on a modified device to collect execution traces and then apply happens-before rules to detect races. Due to their dynamic nature, these dynamic tools suffer from the classic code coverage problem. Alternatively, static tools have also been considered. nAdroid [4] performs a so-called *thread-ification* technique to transform the events in Android into traditional threads in order to leverage a traditional data race detection tool, Chord [29]. In addition, some heuristics-based filters are also used to reduce false positives. SIERRA [3] and Sard [6] model asynchronous events context-sensitively and apply their own happens-before rules for detecting races. Some static tools [42, 43] for detecting other bugs may also be able to detect some event-based races. However, static tools are susceptible to false positives due to, for example, imprecision in the alias information and happens-before relations used.

An effective approach for reducing false positives reported by static tools is to further expose the race warnings from static tools dynamically. Both ERVA [44] and RacerDroid [45] can take the results from existing tools and replay the corresponding events to invoke callbacks reaching the racy statements in a correct order. However, both tools still suffer from the code coverage problem, as ERVA can only work together with a dynamic tools (due to its design on building its event dependency graphs dynamically) and RacerDroid relies on dynamic exploration to obtain the callbacks reaching racy statements. APEChecker [9] exposes event-based races by finding and rescheduling statically the events that can invoke the callbacks reaching a pair of racy statements. However, racy statements are usually guarded by event-dependent conditionals, limiting its effectiveness (as evaluated in the paper).

VII. CONCLUSION

We have introduced a new approach, SIEVE, for exposing event-based races dynamically in Android apps based on a new selective branch instrumentation technique. In comparison with the state of the art, SIEVE can expose the event-based races in real-world Android apps more effectively, making it easily deployable as a practical bug-detection tool.

VIII. ACKNOWLEDGEMENT

We wish to thank all the reviewers for their comments and feedback. This research is supported by Australian Research Grants ((DP170103956 and DP180104069).

References

[1] *Market Share*, https://www.netmarketshare.com/operating-system-market-share.aspx?id=platformsMobile.

[2] B. Zhou, I. Neamtiu, and R. Gupta, "Experience report: How do bug characteristics differ across severity classes: A multi-platform study," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 507–517.

[3] 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.

[4] 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.

[5] 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.

[6] D. Wu, J. Liu, Y. Sui, S. Chen, and J. Xue, "Precise static happens-before analysis for detecting uaf order violations in android," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 276–287.

[7] 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.

[8] 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*, ser. OOPSLA'15, 2015, pp. 332–348.

[9] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu, "Efficiently manifesting asynchronous programming errors in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, 2018, pp. 486–497.

[10] *PhoneFoneFun: a playphone for toddlers*, https://f-droid.org/packages/com.quaap.phonefonefun/.

[11] *VitoshaDM: helps you to make decisions*, https://f-droid.org/packages/eu.veldsoft.vitoshadm/.

[12] *Android Debug Bridge: a versatile command-line tool*, https://developer.android.com/studio/command-line/adb.

[13] *Logcat*, https://developer.android.com/studio/command-line/logcat.

[14] 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.

[15] 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.

[16] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in Android applications," in *International Conference on Software Engineering*, 2015, pp. 89–99.

[17] *F-Droid: Free and Open Source Android App Repository*, https://f-droid.org/.

[18] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *International Conference on Information Systems Security*, 2008, pp. 56–70.

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

[20] T. Tan, Y. Li, and J. Xue, "Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, 2017, p. 278–291.

[21] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11, 2011, p. 17–30.

[22] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, p. 1–41, 2005.

[23] N. Grech, G. Kastrinis, and Y. Smaragdakis, "Efficient Reflection String Analysis via Graph Coloring," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), pp. 26:1–26:25.

[24] Y. Li, T. Tan, and J. Xue, "Effective soundness-guided reflection analysis," in *Static Analysis*, S. Blazy and T. Jensen, Eds., 2015, pp. 162–180.

[25] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Springer Berlin Heidelberg, 2008, pp. 337–340.

[26] 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*, ser. OOPSLA '02, 2002, pp. 211–230.

[27] 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*, ser. PLDI '00, 2000, p. 219–232.

[28] D. Engler and K. Ashcraft, "Racerx: Effective, static de-

tection of race conditions and deadlocks," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, 2003, pp. 237–252.

[29] 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*, ser. PLDI '06, 2006, pp. 308–319.

[30] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: Practical static race detection for c," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2011.

[31] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, p. 337–348.

[32] Q. Zhou, L. Li, L. Wang, J. Xue, and X. Feng, "May-happen-in-parallel analysis with static vector clocks," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018, 2018, pp. 228–240.

[33] 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.

[34] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, pp. 391–411, 1997.

[35] 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, pp. 401–410.

[36] T. Zhang, D. Lee, and C. Jung, "Txrace: Efficient data race detection using commodity hardware transactional memory," *ACM SIGARCH Computer Architecture News*, pp. 159–173, 2016.

[37] T. Zhang, C. Jung, and D. Lee, "Prorace: Practical data race detection for production use," *ACM SIGOPS Operating Systems Review*, pp. 149–162, 2017.

[38] J. Huang and A. K. Rajagopalan, "Precise and maximal race detection from incomplete traces," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016, 2016, p. 462–476.

[39] 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*. ACM, 2011, pp. 369–384.

[40] J. Wu, H. Cui, and J. Yang, "Bypassing races in live applications with execution filters." in *OSDI*, vol. 10, 2010, pp. 1–13.

[41] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race detection for web applications," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, 2012, pp. 251–262.

[42] 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.

[43] 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.

[44] 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.

[45] H. Tang, G. Wu, J. Wei, and H. Zhong, "Generating test cases to expose concurrency bugs in android applications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016, 2016, pp. 648–653.