Detecting Event Anomalies in Event-Based Systems

Gholamreza Safi Arman Shahbazian William G.J. Halfond Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA, USA
{gsafi,armansha,halfond,neno}@usc.edu

ABSTRACT

Event-based interaction is an attractive paradigm because its use can lead to highly flexible and adaptable systems. One problem in this paradigm is that events are sent, received, and processed nondeterministically, due to the systems' reliance on implicit invocation and implicit concurrency. This nondeterminism can lead to event anomalies, which occur when an event-based system receives multiple events that lead to the write of a shared field or memory location. Event anomalies can lead to unreliable, error-prone, and hard to debug behavior in an event-based system. To detect these anomalies, this paper presents a new static analysis technique, *DEvA*, for automatically *d*etecting *ev*ent *a*nomalies. *DEvA* has been evaluated on a set of open-source event-based systems against a state-of-the-art technique for detecting data races in multi-threaded systems, and a recent technique for solving a similar problem with event processing in Android applications. *DEvA* exhibited high precision with respect to manually constructed ground truths, and was able to locate event anomalies that had not been detected by the existing solutions.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.5 [**Testing and Debugging**]: Diagnostics

General Terms

Design, Experimentation

Keywords

Event-based system, Android application, Event anomaly, Race detection

1. INTRODUCTION

The event-based paradigm allows developers to design and build systems that are highly flexible and can be easily adapted [24, 53, 28, 21]. These advantages have made event-based systems (EBSs) popular in a range of domains [16, 17]. However, the event-based paradigm can also introduce complications due to the nature of the underlying event processing. Events can be delayed, damaged, or lost because of environment problems, such as network errors or hardware deficiencies. More generally, events can be consumed nondeterministically, independent of their order of occurrence [31]. While directly enabling EBS flexibility, this nondeterminism can also lead to unpredictable system behavior.

In this paper we address one specific type of problem related to event handling, *event anomaly (EA)*. An EBS has an EA when the processing of two or more events results in accesses to the same memory location (e.g., a variable containing state or data) and at least one of those is a write access. The impact of an EA can vary based on the role the affected variable plays in the EBS, but could include undesirable behaviors ranging from inconsistent state to data corruption. Detecting EAs is hard for several reasons:

Nondeterminism: When a tester suspects that an EBS contains an EA, nondeterminism can make it difficult to execute or reproduce the EA. This reduces the efficiency and effectiveness of standard test-based detection techniques and popular spectrum-based fault localization techniques.

Implicit Invocation: EBSs rely on callbacks, which are methods registered with an event-notification facility and called when the notifier receives an appropriate event. The resulting implicit invocation makes it difficult to identify the control-flow of the code when an event is received and processed.

Ambiguous Interfaces: Event callback methods often accept a generic `Event` type. They must examine the event's attributes to determine its actual type [26] and to dispatch the event appropriately [32]. Ambiguous interfaces make it difficult to determine the event type that is responsible for an EA.

Implicit Concurrency: Different types of received events may result in different method invocations in an EBS, thereby introducing different execution paths. Each of these paths is independent of the others and may be executed in any order, depending on when the events are consumed [19]. The resulting implicit concurrency makes it difficult to identify when a variable may be accessed by two different execution paths.

Implicit invocation, implicit concurrency, and ambiguous interfaces are useful mechanisms for EBSs. Together they allow EBSs to be loosely coupled and to scale easily. At the same time, these very mechanisms make it harder to determine execution order and memory-access patterns in an EBS, whose interplay results in EAs. As a consequence, it can be very challenging for developers to detect EAs in EBSs.

Researchers have recognized the need for automatically discovering EAs. For instance, *CAFA* [29] identifies *Use-After-Free (UF)*, a common type of EA in Android applications, while WEBRACER [47] and EVENTRACER [50] focus on detecting certain types of EAs in client-side web applications. However, these techniques are based on dynamic analysis; therefore, they offer no guarantees

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy

http://dx.doi.org/10.1145/2786805.2786836

of completeness and can only identify EAs that have been exposed during an execution. Other approaches try to address the problem of EAs by introducing new programming language constructs [34, 22]. However, these approaches are not applicable to existing systems that are written in general-purpose languages, such as Java. Furthermore, they tend to consider callback handlers with explicitly typed events as the potential source of anomalies, when for many EBSs the cause of EAs is the processing of generically typed events inside the handlers, followed by dispatching to different component methods.

To address these limitations of existing techniques, we have developed a new static analysis technique that specifically targets EBSs and can handle the semantics of implicit invocation, ambiguous interfaces, and implicit concurrency. Our approach, called *DEvA* (for *D*etecting *Ev*ent *A*nomalies), builds on our recent work on static analysis to identify ambiguous interfaces in EBSs [26] and adds further analyses to identify variables that may be modified as a result of receiving an event—a potential EA. We evaluated *DEvA* on 20 open-source event-based applications and libraries from different domains. We compared *DEvA*'s performance to two state-of-the-art techniques, one targeting traditional data races in multi-threaded systems and the other targeting *UF* anomalies in Android. Our evaluation results show that *DEvA* has high precision and was able to find EAs that the other techniques were unable to detect. *DEvA* was also fast, averaging one minute per application analysis, which is significantly faster than the previously proposed techniques.

The remainder of the paper is organized as follows. Section 2 discusses a motivating example. Section 3 defines fundamental concepts underlying EBSs and provides a formal definition of EAs. Sections 4 and 5 detail *DEvA* and its evaluation. Section 6 summarizes the related work. Section 7 concludes the paper. Finally, Section 8 includes information about *DEvA*'s implementation and the replication package we have made available.

2. MOTIVATING EXAMPLE

As a simple example, consider *MyTracks*, an Android application developed by Google. *MyTracks* records the location, path, distance, and elevation of a Google Maps user. Figure 1 shows a portion of this application, specifically, the handlers for two events: *onLocationChangedAsync* (line 5) processes `LocationChanged-Async` events and *onDestroy* (line 12) processes `Destroy` events. `Destroy` is a common event used for memory management that, when it occurs, frees the memory assigned to an activity or service of a given Android application.[1]

In the case of *MyTracks*, the EA occurs when both `Destroy` and `LocationChangedAsync` events occur but `Destroy` is processed first. The *onDestroy* handler will free memory by setting the private class variable *providerUtils* to null. At some point shortly thereafter, *onLocationChangedAsync* will attempt to access *providerUtils*, thus generating a null pointer exception. The cause of this exception is an EA, since each of the two events results in an access to the same memory location and one of them is a write access.

The example in Figure 1 is an instance of an EA known as *Use-After-Free* (*UF*). Our approach, *DEvA*, was able to detect this EA, but existing techniques, such as *CAFA* [29], were unable to do so (see Section 5.) These types of EAs also occur in other real-world systems, such as *Firefox* for Android [12, 11].

[1]Note that invoking *onDestroy* is not equivalent to performing garbage collection in Android. *onDestroy* only provides an opportunity to clean things up before an activity or service is destroyed: it can change the memory locations referenced by the activity or service instance or free them by setting them to null, but it does not release the instance.

```
1  public class TrackRecordingService
2         extends Service
3         implements LocationListener {
4  ...
5   private void onLocationChangedAsync(
6         Location location){
7      ...
8      Location lastRecordedLocation =
9            providerUtils.getLastLocation();
10     ...
11   }
12   public void onDestroy() {
13     Log.d(TAG,
14           "TrackRecordingService.onDestroy");
15     ...
16     providerUtils = null;
17     ...
18   }
19 ...
20 }
```

Figure 1: Excerpt from the *MyTracks* Android application

3. FOUNDATIONS

In this section we define the underlying concepts and terminology that we will use to describe *DEvA* in Section 4. We describe how events are defined in an EBS, and how we employ our recent static analysis technique [26] to automatically identify different event types in systems with ambiguous interfaces. Finally, we provide a formal definition of event-based anomalies.

To illustrate our definitions, we will use the implementation of a *LoadBalancer* component shown in Figure 2. *LoadBalancer* monitors the state of a server. Whenever the load on the server rises above a specified limit, it will stop responding to new requests. A new request comes to *LoadBalancer* through a `NewRequest` event. At lines 6 and 7 of Figure 2, *LoadBalancer* consumes this event and decides whether to process it on the server or send a notification event to inform other parts of the system that the server is overloaded. The limit for the load on the server is set whenever *LoadBalancer* receives a `SetLimit` event. At lines 9-13, *LoadBalancer* consumes this event, sets the limit, and checks if the new limit is less than the previous one. If so, it informs other parts of the system about the limit reduction.

3.1 Terminology

Our approach relies on control-flow information in the form of a *control-flow graph* (CFG). A CFG is a directed graph in which each node represents a basic block of code and each edge represents the relationship between two blocks that at runtime may be executed after one another. We also use the *inter-procedural control-flow graph* (ICFG), which is a directed graph that contains the CFGs of a component's different methods. If one method calls another, then there is an edge in the ICFG between the method invocation point and the entry node of the target method's CFG.

The ICFG of *LoadBalancer* is shown in Figure 3. The nodes in the ICFG correspond to the statements in Figure 2. We use this graph to illustrate two more concepts. First, a CFG node X is *control dependent* on node Y if execution of Y determines whether X will be executed. Consider the CFG of the *handle* method in the *LoadBalancer* component, which is shown in Figure 3 starting at node *Entry* in the top middle of the graph. Nodes 7 and 8 are control dependent on 6 being true but node 9 is not, since 9 will execute for either condition of node 6. Second, a node X is *data dependent* on node Y if there is a path in the CFG from Y to X, X uses the value of a variable that has been defined at Y, and no other node on that path redefines that variable [27]. In the *LoadBalancer* ex-

```
1  public class LoadBalancer extends Component{
2    Long curLoad;
3    Long lmt;
4    Long preLmt;
5    public void handle(Event e){
6      if(e.getName().equals("NewRequest")){
7        manageCurrentLoad(e.getRequest());
8      }
9      if(e.getName().equals("SetLimit")){
10       preLmt = lmt;
11       lmt = Long.parseLong(e.getAttr("Limit"));
12       if(lmt < preLmt)
13           send(new Event("LimitReduced"));
14     }
15   }
16   public void manageCurrentLoad(Request r){
17     if(curLoad < lmt){
18       curLoad = curLoad + 1;
19       //Start processing new request
20     }
21     else
22       send(new Event("LoadTooHigh"));
23   }
24 }
```

Figure 2: Event-based load-balancing component

ample, node 12 is data dependent on node 11, since at line 12 in Figure 2 the *if* statement uses the *lmt* variable and *lmt* is defined at node 11.

To show all control- and data-dependency relationships between different nodes of an ICFG, we can use the Program Dependence Graph (PDG). In a PDG, the nodes represent basic blocks in the program and edges represent control or data dependencies between the nodes. Figure 4 shows the PDG for the example from Figure 2.

3.2 Consumed Events

Event-based systems (EBSs), also referred to as message-based systems, are widely used in a range of application domains, from user interface software to distributed systems. In an EBS, the process of a component receiving an event and accessing the information defined in the event's attributes is called event *consumption*. At lines 9-11 of Figure 2, the event `SetLimit` is consumed, since *LoadBalancer* receives the event and processes the event's contents by checking its "*Name*" and "*Limit*" attributes. An EBS component's entry point for event consumption is called a *sink*, or alternatively an *event handler*. In Figure 2, the *handle* method defined at line 5 is a sink. We say that *LoadBalancer* has an *ambiguous interface* [26] since it receives all events via the same sink.

Identifying the types of events consumed by an EBS in the presence of ambiguous interfaces is challenging because all events handled by a system arrive as a single, generic `Event` type. The actual type of each event is determined at runtime by checking the attributes contained within the event. Our recent work [26] resulted in a static analysis technique, called Eos, that analyzes the implementation of an EBS to infer the set of event types it can consume at runtime.

Eos leverages *consumed event revealing* (*CER*) statements to infer information about the *consumed event types* (*CETs)* of a component. A CER statement is a method call that retrieves information stored in an event without modifying the event's attributes. A CET is defined by a set of attributes and their values, which can be inferred from CER statements. Methods called by a CER statement are of the following forms:

1. *getAttr*(a) represents methods retrieving the value of the attribute named a of an event. As an example, consider *getAttr* in Figure 2 at line 11.

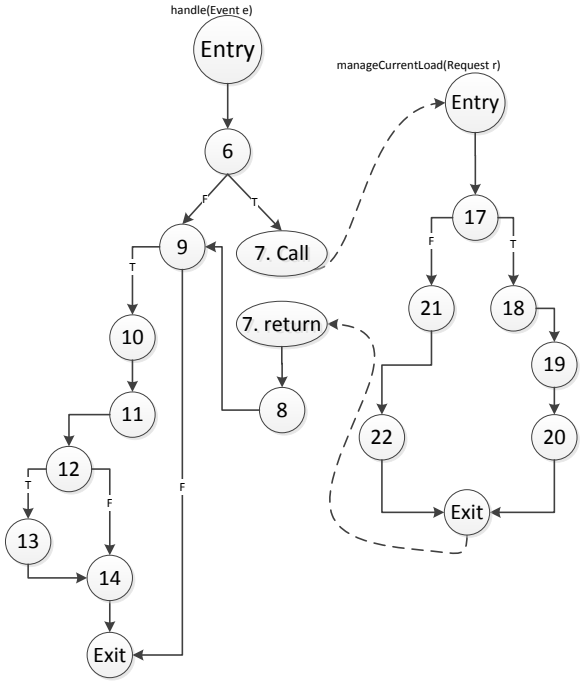

Figure 3: *ICFG* of LoadBalancer

2. *retrieveSpecificEventInfo*() represents methods whose names indicate the information the methods directly obtain about an event; this is in contrast to *getAttr*(a), which relies on parameter a to specify the information of interest. For example, in Figure 2 at line 6 *e.getName*() is used to retrieve the name parameter of event e, while at line 7 *e.getRequest*() is used to retrieve the request parameter contained in e.
3. *hasAttribute*(a) represents methods that check whether an event has a specific attribute with the name a.
4. $m(e)$ indicates methods that accept an event e as an argument. The explicitly declared type of the argument, required by the programming language, reveals information about the type of the event e, which may have been cast elsewhere in the system before $m()$ was invoked.

Eos identifies CER statements in a component and aggregates their information along the paths in the CFGs of the component's methods. The type information revealed by the CER statements along each path is used to define a CET of the component. We refer to the set of all event types that are consumed in a component as its *CET* set. Eos also returns all of the nodes in the ICFG that contain a CER statement that contributed to the definition of each CET.

To illustrate, consider the *LoadBalancer* component from Figure 2. Its CET set consists of two event types. The first event type has two attributes, one with the name "*Name*" (whose value is `NewRequest`) and one with the name "Request". The second event type also has two attributes, "*Name*" (with the value `SetLimit`) and *"Limit"*.

3.3 Definition of Event Anomalies

An event consumption may cause an access to the state of a component, which is represented by the component's fields as well as any shared, non-local variables. For example, in the *LoadBalancer* component, variables *curLoad*, *lmt* and *preLmt* are its fields. An access to a field is either a read (i.e., use) or a write (i.e., definition). When the consumption of two different CETs causes accesses to the same field, with at least one access being a write, a problem

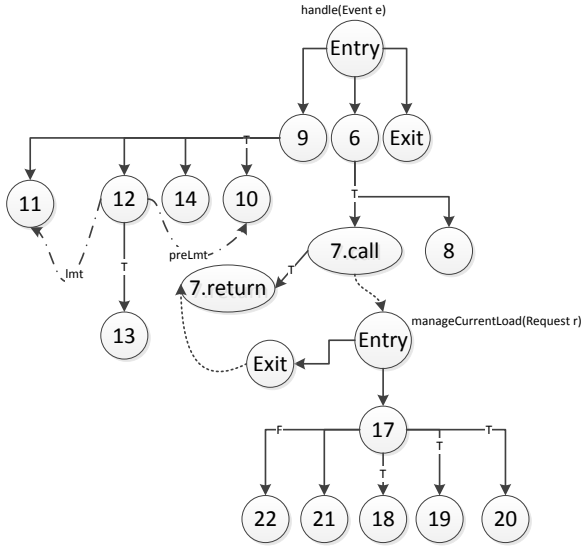

Figure 4: *PDG* of LoadBalancer

may occur: the consumption order of the two events may be dependent on the system's environment and, therefore, the system's behavior may be nondeterministic. We refer to this potential problem as an *event anomaly (EA)*. The goal of our technique, *DEvA*, is to identify and report such anomalies to developers.

Let *Components* be the set of all components in a system. For a given $c \in Components$, we define F_c as the set of all of c's fields. Furthermore, $cerExtract(e)$ is Eos's function that extracts all *CER* statements for an event e. There exists an EA in component c over the field $f \in F_c$ due to event types $e_1, e_2 \in CET_c$ iff the following conditions hold:

1. $e_1 \neq e_2$
2. $\exists\{X_1,...,X_p\} \subset Nodes(ICFG) \mid 1 \leq i < p,\ X_{i+1}$ *is control or data dependent on* $X_i \wedge X_1 \in cerExtract(e_1)$
3. $\exists\{Y_1,...,Y_q\} \subset Nodes(ICFG) \mid 1 \leq j < q,\ Y_{j+1}$ *is control or data dependent on* $Y_j \wedge Y_1 \in cerExtract(e_2)$
4. X_p *is a use or definition of* f
5. Y_q *is a definition of* f

The intuition behind this definition is that we can say an access to a field has been caused by an event consumption whenever the occurrence of consumption determines that the access must happen (control dependency) or the consumption affects the value that is stored in that field (data dependency). The above definition has three principal parts: (i) the CET set of a component (condition 1); (ii) the control or data dependency paths from CETs to fields that imply causality (conditions 2 and 3); and (iii) determining those paths that access the same field with at least one write access (conditions 4 and 5). These three elements form the foundation of our approach for extracting event anomalies, described next.

4. APPROACH

The goal of *DEvA* is to identify EAs in EBSs. To this end, we have developed an automated static analysis technique that examines the implementation of a system and identifies points in the code where EAs may occur. Our approach identifies all possible EAs, regardless of their impact on the EBS. Determining the impact of an EA is challenging, in part because it can vary significantly. For example, prior work has found that some races are harmless and that removing them by introducing additional checks in the code can compromise the performance of the system [51, 43, 29]. On the other hand, EAs can lead to significant reliability problems and could impact system scalability [18]. The determination of the category into which an EA falls is generally a task for a system's engineers who must evaluate the EA's impact on the EBS's functional and non-functional requirements.

The main inputs to our analysis are the implementation of an EBS and a description of the framework used for processing events in the system. The description must specify: (1) the list of all methods used as event sinks (in the case of an EBS with ambiguous interfaces) or callback methods that serve as event sinks (in the case of frameworks, such as Android, that rely on explicit event interfaces); (2) the base class used to implement events in the system; and (3) the set of methods used as consumed event revealing (CER) statements. All of the information in the description can be derived from the API specification of the underlying event-based framework and only needs to be identified once per framework.

DEvA's analysis can be divided into three distinct phases. In the first phase—*extraction*—the analysis identifies all of the consumed event types (CETs) and fields accessed by each component of the system. The second phase—*causation*—performs a path-based analysis to determine if there is a connection between the CETs and accessed fields. Finally, the third phase—*joining*—identifies CETs that will lead to an access of the same field, that is, a possible EA. The CETs and fields identified in the last phase are returned to the developer for more investigation. In the remainder of this section, we discuss each of the three phases in more detail.

4.1 Extraction

During this phase, *DEvA* identifies two types of information about the EBS that will be used in the later phases to identify anomalies. The first is the set of locations within each component where the component's fields are accessed either by a use or definition. The second is the set of CETs accessed by each of the components.

DEvA identifies a component's field accesses via a static intra-procedural analysis of the component's implementation. Formally, we define a field access as a tuple $\langle f, n\rangle$ in which $f \in F_c$ is a field of component c (recall that F_c is the set of all fields of component c) and $n \in N_m$ is a node in a method m's control-flow graph (N_m represents the set of all nodes in method m's CFG) that represents the location in the code where the field is used or defined. *DEvA* generates two different sets, $FUse_m$ that contains field uses and $FDef_m$ that contains field definitions.

$$\begin{array}{lllllll} FDef_m & = \{\langle f,n\rangle | f \in F_c & \wedge & n \in N_m & \wedge & n & \text{defines} \quad f\} \\ FUse_m & = \{\langle f,n\rangle | f \in F_c & \wedge & n \in N_m & \wedge & n & \text{uses} \quad f\} \end{array}$$

To compute these sets, *DEvA* first builds the CFG of each component method. It then traverses the CFG and checks each node to determine if it accesses a field. To analyze a field we performed alias analysis that first used class hierarchy analysis (CHA) to identify potential targets of a field access and then used points-to analysis based on SPARK [33] to refine the results.

To illustrate this step, consider the *LoadBalancer* component shown in Figure 2 and its ICFG shown in Figure 3. The *FDef* and *FUse* sets are shown in Table 1. In the *handle* method, since there is a definition of field *preLmt* at node 10 of the ICFG (corresponding to line 10 in *LoadBalancer*'s implementation) and a definition of field *lmt* at node 11, tuples $\langle preLmt, 10\rangle$ and $\langle lmt, 11\rangle$ are added to $FDef_{handle}$. Also since there are two uses of field *lmt* at nodes 10 and 12, and a use of field *preLmt* at node 12 tuples $\langle preLmt, 12\rangle$, $\langle lmt, 10\rangle$ and $\langle lmt, 12\rangle$ are added to $FUse_{handle}$.

Our recently developed technique, Eos [26], is able to identify CETs in EBSs that use ambiguous interfaces. Running Eos on the

Table 1: *LoadBalancer* **component information**

Methods	*FDef*	*FUse*
handle	$\{\langle preLmt, 10\rangle, \langle lmt, 11\rangle\}$	$\{\langle preLmt, 12\rangle, \langle lmt, 12\rangle, \langle lmt, 10\rangle\}$
manageCurrentLoad	$\{\langle curLoad, 18\rangle\}$	$\{\langle curLoad, 17\rangle, \langle curLoad, 18\rangle, \langle lmt, 17\rangle\}$

LoadBalancer component would identify the two event types discussed in Section 3.2: the first event with the attributes, *Name* and *Request*, that are accessed using CER statements at lines 6 and 7 of Figure 2; and the second event with the attributes, *Name* and *Limit*, that are retrieved using the CER statements at lines 9 and 11. In addition to the CET set, we extended Eos to also output the code locations at which different attributes of each CET are retrieved, rather than just the name and value of each attribute. We thus extract those locations and record them along with the name of corresponding event in the CET set. Therefore, the CET set for *LoadBalancer* will be $\{\langle$"*NewRequest*"$, 6, 7\rangle, \langle$"*SetLimit*"$, 9, 11\rangle\}$.

Algorithm 1: Callback sink extraction

```
   Input: Components, Interfaces, CET and S (sink) sets for each
          component
   Output: Updated CET and S (sink) sets for each component
1  foreach i ∈ Interfaces do
2      foreach c ∈ Components do
3          if c implements i then
4              add (c,i) to ImplementedInterf
5          if ∃f ∈ F_c such that f and i have the same type t then
6              add (c,t,i) to Candidates
7  foreach (c,i) ∈ ImplementedInterf do
8      if ∃f ∈ F_c of type t_f and ∃(t_f,t,i) ∈ Candidates then
9          foreach m ∈ Methods_i do
10             add (m.name,m.entry_node) to CET_c
11             add m to S_c
12             if ∃n ∈ Methods_c ∧ n is called by a thread defined inside m
               then
13                 add (n.name,n.entry_node) to CET_c
14                 add n to S_c
```

Eos targets EBSs with ambiguous interfaces and is not able to identify callback methods, which serve as sinks for explicitly typed events in frameworks such as Android. Example callback sinks in Android are *onLocationChangedAsync* and *onDestroy* from Figure 1. Standard callbacks, such as *onDestroy*, can be easily identified in the code. However, custom, application-specific event sinks, such as *onLocationChangedAsync*, must also be identified. Algorithm 1 identifies two common patterns of custom event sinks, and uses these to identify sinks and CETs for Android applications. The first pattern comprises an interface i, a component c_1 that implements interface i, and a component c_2 that defines a field variable f_1 with the same type as interface i; in turn, c_1 should define a field variable f_2 with the same type as c_2. Two components have the same type when they are either instances of the same class or one extends or implements the other. In this situation, whenever component c_1 is instantiated, it will provide its "*this*" reference via f_2 to component c_2, and c_2 will save the reference in f_1. By doing this, c_2 will be able to call methods that are declared in interface i on component c_1. The second pattern comprises a definition of a thread inside a callback method and a call to another method inside that thread to perform the required operation asynchronously.

Algorithm 1 finds the first pattern by first searching for all components c that implement a given interface i (lines 1-4) and stores the $\langle c, i\rangle$ tuples in the *ImplementedInterf* set. After that, Algorithm 1 checks if a given component c has a field variable of the same type t as the interface i (lines 5-6) and stores the $\langle c, t, i\rangle$ tuple in the *Candidates* set. Finally, Algorithm 1 searches through *ImplementedInterf* to check for all components c that implement an interface i and have a field variable of type t_f, such that t_f's class, in turn, has a field variable of type i (lines 7-8). In other words, Algorithm 1 checks for tuples of the form $\langle t_f, t, i\rangle$ in the *Candidates* set. If such a tuple is found, then the methods in interface i will be added to c's sinks and the CET set (lines 9-11). Algorithm 1 finds the second pattern by identifying methods called asynchronously inside a thread, and adding them to the S (i.e., sink) and *CET* sets of component c (lines 12-14). *onLocationChangedAsync* from the *MyTracks* application of Figure 1 is an example of an asynchronous callback in Android.

Algorithm 1 over-approximates callback sinks to detect all possible user-defined callback methods and to avoid false negatives. This over-approximation may cause some false positives in the manner discussed in Section 5.6.

4.2 Causation

The second phase of *DEvA*'s analysis identifies whether field accesses are dependent on the consumption of specific event types. To this end, *DEvA* analyzes each field access location to determine if it is control or data dependent on CER statements that are used to define a CET. The intuition is that if such dependencies exist, then the field access occurs, at least in part, due to the consumption of an event type, and may be part of an EA.

Intuitively, one can think of this phase as analyzing the program dependence graph (PDG) of a component. For each field access, *DEvA* performs a backwards traversal of the edges in the PDG. If a CER statement is encountered during this traversal, then *DEvA* identifies the corresponding CET and the originating field as being connected. The phase outputs these connected CETs and fields.

For larger systems, generating and traversing a PDG is not scalable. Case in point, to naively analyze the subject systems used in our evaluation (see Section 5), it would be necessary to generate PDGs for over 35 methods on average. This could consume hours for a typical application. To address this issue, *DEvA* only generates PDGs for a component's sink methods, and then uses the call graph (CG) of the system to identify methods that are reachable from each sink. A field's definition or use in method m that is reachable from sink s may be caused by an event's consumption if the invocation in s that initiates the call to m is control or data dependent on that consumption. We now detail the algorithms that implement this approach.

The algorithm for this phase is shown as Algorithm 2. The inputs to this algorithm are the component c to be analyzed, the CETs of the component (CET_c), the call graph of the component (CG_c), and the *FUse* and *FDef* sets for each method in c. Note that CET_c, *FUse*, and *FDef* are the outputs of the first phase discussed in Section 4.1. The outputs of the algorithm are two sets, *ConsumedToDef* and *ConsumedToUse*, which contain tuples representing the fields, and CETs that are linked by a dependency relationship. Each tuple is of the form $\langle f, n, e\rangle$ where f is the field, n is the location of the field's access in the code, and e is the CET. The set *ConsumedToDef* contains tuples where n represents a definition of f, while in *ConsumedToUse* n represents a use of f.

Algorithm 2 first accesses a set S_c that contains all sink methods in c. This set is defined using method signatures for applications with ambiguous interfaces, but is augmented with the results of Algorithm 1 for Android applications. Then the algorithm iterates over each method m in c (lines 2–5). If m is a sink or it can be reached from a sink in the call graph (line 3), then *DEvA* analyzes

Algorithm 2: Causation

Input: $c \in Components$, CG_c, CET_c, $\forall m \in Methods_c : FDef_m, FUse_m$
Output: $ConsumedToDef_c$, $ConsumedToUse_c$

1 Let S_c = set of all c's event sinks
2 **foreach** $m \in Methods_c$ **do**
3 **if** $(m \in S_c)$ *or* $(\exists s \in S_c$ *so that m is reachable from s in* $CG_c)$ **then**
4 $ConsumedToDef_c += fieldAccessBackToConsumption(c, m, FDef_m, CET_c)$
5 $ConsumedToUse_c += fieldAccessBackToConsumption(c, m, FUse_m, CET_c)$

Algorithm 3: *fieldAccessBackToConsumption*

Input: $c \in Components$, $m \in Methods_c$, $FieldAccesses$, CET_c
Output: $ConsumedtoAccess_c$

1 Let N_m = set of all nodes in m's CFG
2 Let $cerNodes(e,m)$ where $e \in CET_c$ and $m \in Methods_c$ = $\{n | n \in N_m \wedge n \in cerExtract(e)\}$
3 Let S_c = set of all of c's event sinks
4 Let $S_m \subset S_c$ = set of sinks that can reach m in CG_c
5 Let $StoM(s,m)$ = set of nodes in the CFG of $s \in S_c$ that can reach m through direct or indirect calls
6 **foreach** $(f,n) \in FieldAccesses$ **do**
7 **foreach** $e \in \text{CET}_c$ **do**
8 **if** $m \in S_c$ **then**
9 **if** $\exists l \in cerNodes(e,m)$ *such that n is directly or transitively control or data dependent on l* **then**
10 *add* (f,n,e) *to* $ConsumedtoAccess_c$
11 **else**
12 **foreach** $t \in S_m$ **do**
13 **foreach** $k \in StoM(t,m)$ **do**
14 **if** $\exists l \in cerNodes(e,t)$ *so that k is directly or transitively control or data dependent on l* **then**
15 *add* (f,k,e) *to* $ConsumedtoAccess_c$
16 return $ConsumedtoAccess_c$

the field accesses in m by calling *fieldAccessBackToConsumption* (lines 4 and 5). The function *fieldAccessBackToConsumption* is shown in Algorithm 3. At a high-level, *fieldAccessBackToConsumption* iterates over each field access in m (line 6) and each event in CET_c (line 7) to determine if a dependency relationship exists between them. Within this iteration, there are two cases to consider. The first case (at line 8) is when m is a sink. In this case, *DEvA* simply checks the PDG of m to see if the field access is dependent on any CER node for the current CET (line 9). If so, the tuple representing the field, location, and event type is added to the output set (line 10). The second case (at line 11) is for any non-sink method. *DEvA* begins by iterating over the set of all sink methods S_m that can reach m (line 12). Within each $t \in S_m$, *DEvA* also iterates over each node k that is in t and can reach m (line 13). We compute this reachability relationship by determining if k can reach an invocation in t's CFG that, in turn, reaches m via the call graph. This relationship is encapsulated in the function *StoM* (line 5). If k is dependent on a CER node, then this relationship is added to the output set (lines 14 and 15). The intuition here is that the field access $\langle f,n \rangle$ can be reached via a statement k that is itself dependent on a CER node.

Let us now consider an example for each of the two cases. To illustrate the first case, consider the *handle* method of the *LoadBalancer* component from Figure 2. Since *handle* is a sink, *fieldAccessBackToConsumption* will use the PDG (shown in Figure 4) to extract those members of $FDef_{handle}$, listed in Table 1, that are control or data dependent on a node that contains a *CER* statement. Consider $\langle lmt, 11 \rangle$ in the *PDG*. Node 11 is connected to node 9, which contains a node with a *CER* statement for the `SetLimit` event; so $\langle lmt, 11, \text{“}SetLimit\text{”} \rangle$ will be added to the return set. Next, $\langle preLmt, 10 \rangle$ is also dependent on node 9, so $\langle preLmt, 10, \text{“}SetLimit\text{”} \rangle$ will be added to the return set.

To illustrate the second case, consider the method *manageCurrentLoad*. Based on the ICFG in Figure 3, only node 7 in the sink *handle* can reach *manageCurrentLoad*. *StoM (handle,manageCurrentLoad)* thus returns a singleton containing node 7. Based on the *PDG* of *handle* (Figure 4), node 7 is control dependent on node 6, which contains a *CER* statement of the `NewRequest` event. Therefore, any definition or use inside the *manageCurrentLoad* method is dependent on this event. Since $FDef_{manageCurrentLoad}$ contains $\langle curLoad, 18 \rangle$, our algorithm will add $\langle curLoad, 18, \text{“}NewRequest\text{”} \rangle$ to the return set for definition accesses of *manageCurrentLoad*.

After Algorithm 2 completes its analysis of the *LoadBalancer* component from Figure 2, its output would be:

$$ConsumedToDef_{LoadBalancer} = \{ \langle lmt, 11, \text{“}SetLimit\text{”} \rangle, \langle curLoad, 18, \text{“}NewRequest\text{”} \rangle, \langle preLmt, 10, \text{“}SetLimit\text{”} \rangle\}$$

$$ConsumedToUse_{LoadBalancer} = \{ \langle curLoad, 17, \text{“}NewRequest\text{”} \rangle, \langle curLoad, 18, \text{“}NewRequest\text{”} \rangle, \langle lmt, 17, \text{“}NewRequest\text{”} \rangle, \langle lmt, 10, \text{“}SetLimit\text{”} \rangle, \langle lmt, 12, \text{“}SetLimit\text{”} \rangle, \langle preLmt, 12, \text{“}SetLimit\text{”} \rangle\}$$

4.3 Joining

DEvA's third phase analyzes the dependencies between CETs and fields to determine which ones may lead to an EA. The intuition is that if one CET–field dependency writes to a field and another one either writes to or reads from that field, then this is an EA.

The algorithm for this phase is shown in Algorithm 4. The inputs to the algorithm are $ConsumedToDef_c$ and $ConsumedToUse_c$, which were generated in the second phase, and whose values for *LoadBalancer* are shown at the end of Section 4.2. The goal of the first step (lines 1–4) of the algorithm is to remove uses that may result in false positives. This step is analogous to identifying reaching definitions [15]: a definition of a given field f reaches a node n in a CFG if there is a path in the CFG from the node at which f is defined to n without any other definition of f on that path. If there is a definition of a field that dominates a use of the same field, then that use cannot be involved in an EA condition with any other definition of the field. Domination occurs when all paths from the entry node of a CFG to the location of field f's use include the node that defines f. In that case, f's use is removed from the *ConsumedToUse* set at line 4 of Algorithm 4. To illustrate this first step, consider the definition of *lmt* at line 11 of Figure 2. This definition dominates the use of *lmt* at line 12. Therefore, the use of *lmt* at line 12 will be removed from $ConsumedToUse_{LoadBalancer}$, after which we will have:

$$ConsumedToUse_{LoadBalancer} = \{ \langle curLoad, 17, \text{“}NewRequest\text{”} \rangle, \langle curLoad, 18, \text{“}NewRequest\text{”} \rangle, \langle lmt, 17, \text{“}NewRequest\text{”} \rangle, \langle lmt, 10, \text{“}SetLimit\text{”} \rangle, \langle preLmt, 12, \text{“}SetLimit\text{”} \rangle\}$$

The second step of the algorithm (lines 5–9) iterates over each CET–field dependency where the field access is a definition, and checks whether there are any definitions or uses of the same field that can be triggered by the consumption of different CETs. Essentially, this performs a join over two inputs when the field in each tuple is the same. If such CET–field dependencies exist, then an EA is detected, and a tuple containing the affected field and the two

Algorithm 4: Joining

Input: $c \in Components$, $ConsumedToDef_c$, $ConsumedToUse_c$, $ICFG$
Output: $EventAnomalies_c$

```
1  foreach ⟨f,n,e⟩ ∈ ConsumedToDef_c do
2      foreach ⟨f2,n2,e2⟩ ∈ ConsumedToUse_c ∧ f = f2 ∧ e = e2 do
3          if f's definition at node n kills all other definitions that reach
           node n2 then
4              remove ⟨f2,n2,e2⟩ from ConsumedToUse_c
5  foreach ⟨f,n,e⟩ ∈ ConsumedToDef_c do
6      foreach ⟨f3,n3,e3⟩ ∈ ConsumedToDef_c ∧ f = f3 ∧ e ≠ e3 do
7          add ⟨f,e,e3⟩ to EventAnomalies_c
8      foreach ⟨f4,n4,e4⟩ ∈ ConsumedToUse_c ∧ f = f4 ∧ e ≠ e4 do
9          add ⟨f,e,e4⟩ to EventAnomalies_c
```

event types is added to the output set. To illustrate the joining algorithm, consider the $ConsumedToDef_{LoadBalancer}$ set, reported at the end of Section 4.2, and $ConsumedToUse_{LoadBalancer}$ after line 4 of Algorithm 4, reported earlier in this section. There is one CET–field dependency that defines *lmt*. This definition happens at line 11 and is caused by the `SetLimit` event. There is also one CET–field dependency that uses *lmt*. This use happens at line 17 and is caused by the `NewRequest` event. These paths access the same field with one of them being a write access and are dependent on different event types, introducing an EA. The output of Algorithm 4 for *LoadBalancer* is, therefore, $\{\langle lmt, \text{“}SetLimit\text{”}, \text{“}NewRequest\text{”}\rangle\}$.

5. EVALUATION

We have empirically evaluated *DEvA* (1) to measure its accuracy in extracting EAs; (2) to compare it with a state-of-the-art race detection technique for multi-threaded systems; (3) to compare its performance with a recently published race detection technique for Android applications; and (4) to examine its execution time.

5.1 Subject Systems and Implementation

Table 2 contains information about the 20 subject systems we have used in our evaluation; 18 of them are applications and the remaining two are widely used Android libraries (e.g., these libraries are used in several of the Android applications from Table 2). All subjects are implemented in Java, but are from different application domains (*App Type*), of different sizes (*SLOC*), and use different underlying mechanisms for consuming events (*Event Mechanism*).

In selecting these subjects, we first located a corpus of suitable systems that make use of events in their implementations. Two PhD students examined a number of open-source applications and identified likely candidates by looking for possible instances of EAs. Each system for which our preliminary examination indicated a potential presence of EAs was then carefully analyzed by the two students with the help of the Eclipse IDE to obtain the ground truth. The generation of the ground truths took slightly more than 15 person-hours per system on average. As discussed in Section 4, even seemingly harmless races may actually harm a system in subtle ways [18], hence our ground truths contained all possible races regardless of their impact on the EBSs.

A notable outlier among our subject systems is *Project.net*, which is significantly larger than the other systems. It partly uses event-based interactions on its server side, but mostly relies on web-based interactions. We only provided *Project.net*'s event-based portion, totaling around 10 KSLOC, as an input to *DEvA*. However, to pinpoint this portion, we had to analyze the entire system.

DEvA is implemented in Java and Scala, and it uses the Soot [55] program analysis library to generate call graphs, control flow graphs, and program dependency graphs. To analyze EBSs that rely on ambiguous interfaces—in our case, this includes all non-Android applications from Table 2—we used an extension of Eos [26] for generating the required inputs for *DEvA*, as described in Section 3.2. We ran *DEvA* on a quad-core Intel i7 2.80GHz system with 8GBs of memory, running Windows 7 Professional.

Table 2: Systems used in the evaluation

App Name	App Type	SLOC	Event Mechanism
Planner	AI Planner [45]	6K	c2.fw [38], Java events
KLAX	Arcade Game [54]	5K	c2.fw [38], Java events
DRADEL	Software IDE [39]	11K	c2.fw [38]
ERS	Crisis Response [36]	7K	Prism-MW [37]
Troops	Simulator [37]	9K	c2.fw [38]
Stoxx	Stock Ticker [40]	6K	REBECA [41]
JMSCHAT	Chat System [8]	12K	ActiveMQ, Java events
Project.net	Project Mgmt [9]	247K	Spring [10]
ToDoWidget	ToDo List Recorder [5]	2K	Android Events
FBReader	Book Feed Reader [3]	34K	Android Events
MyTracks	Location Tracker [4]	13K	Android Events
ZXing	Barcode Scanner [2]	16K	Android Events
Firefox	Browser App [7]	58K	Android Events
ConnectBot	SHH Client [1]	23K	Android Events
VLC	Media Player [6]	101K	Android Events
Browser	Android Browser App	22K	Android Events
Camera	Android Camera App	13K	Android Events
Music	Android Audio Player	8K	Android Events
android.support.v4	Support Library	7K	Android Events
android.support.v7	Support Library	11K	Android Events

5.2 Accuracy of EA Detection

To assess *DEvA*'s accuracy in detecting EAs, we applied it on the subject systems and compared its results to the ground truths. If an anomaly reported by *DEvA* was not in the ground truth, we counted it as a false positive; conversely, an anomaly in the ground truth that was not reported by *DEvA* was counted as a false negative. Our results are summarized in Table 3. For all but three of the systems, *DEvA* was able to find each anomaly identified in the ground truth. The three exceptions were *KLAX*, for which *DEvA* yielded 2 false negatives, *ToDoWidget*, with 12 false negatives, and *MyTracks*, with 1 false negative. *DEvA* did not report any results that we had not found and confirmed as EAs in our ground truths, i.e., it had no false positives.

Table 3: Results of applying *DEvA* on the subject systems

System	Anomalies	Time (*s*)
Planner	11	62
KLAX	12	62
DRADEL	447	122
ERS	37	54
Troops	30	75
Stoxx	1	66
JMSCHAT	3	29
Project.net	1	27
ToDoWidget	7	24
FBReader	74	58
MyTracks	73	65
ZXing	4	52
Firefox	39	81
ConnectBot	32	78
VLC	52	42
Browser	111	36
Camera	212	45
Music	65	33
android.support.v7	50	35
android.support.v4	109	59

In the cases of *KLAX*, *ToDoWidget*, and *MyTracks*, the false negatives occurred because these systems rely on non-standard mechanisms to access component state and communicate state changes. For example, *MyTracks* dispatches an event by directly accessing the event's *"what"* attribute, while *KLAX* passes the entire current state of the game from one component to another via a single event with a single, very complex parameter. *DEvA* could be relatively

easily modified to cover each of these exceptional cases. However, we have chosen not to do so for our evaluation because it will always be possible to use other unforeseen, non-standard "hacks" when generating and processing events so that *DEvA* or a similar technique would not catch them without accounting for additional special cases. *DEvA* relies on EBS engineers to exercise relatively minimal discipline when developing their systems.

DEvA reported several hundred EAs across the 20 subject systems. As discussed above, *DEvA* was able to identify a great majority of the anomalies present in our ground truths, and it did not yield any false positives: each reported anomaly did, in fact, reflect implicitly-concurrent accesses to a component field with at least one access being a write. An EA may have one of three possible outcomes:

1. *It is clearly a bug.* An example is the *UF* anomaly in the *MyTracks* Android application from Figure 1.
2. *It is clearly an undesirable nondeterministic behavior.* An example involves the `GameOver` event in KLAX, which can be preempted by the `GamePaused` event, resulting in an additional life for the player.
3. *It is potentially an undesirable nondeterministic behavior.* An example is a *Stoxx* event that changes a threshold on a stock's price while another event requests a computation using the threshold and the stock's change history.

While not all of the EAs reported by *DEvA* will have the same effect, they all have the potential to cause undesired behavior in a system and should be carefully examined by the engineers [18].

5.3 Comparison to Multi-Threaded Analysis

In a multi-threaded environment, a data race occurs when two or more threads access the same memory location without proper synchronization. Because this is similar to our problem, we investigated whether standard data race detection techniques could be applied to find EAs.

To determine whether this is the case, we studied the literature on race detection and selected *Chord* [42] as a state-of-the-art static analysis technique to apply on our subject systems. *Chord* was chosen because it is the only technique of its kind that has a reliable, actively maintained implementation. The purpose of this study was to establish the extent of overlap between the outputs produced by *Chord* and *DEvA*. The numbers of data races found by *Chord* are shown in Table 4. Since *Chord* starts its analysis from a *main* method and *Project.net* has no such method, *Chord* was unable to analyze *Project.net*. Furthermore, we were unable to apply *Chord* on Android applications because *Chord* is not designed to consider the methods used in Android as event entry points.

Despite this, the comparative analysis was revealing. We found that the result sets produced by *Chord* and *DEvA* do not overlap: none of the EAs reported by *DEvA* were among the data races reported by *Chord*, or vice-versa. *DEvA* was unable to detect any of the data races in the subject systems for the simple reason that it does not target traditional data races. Similarly, *Chord* does not consider the conditions that result in EAs. This suggests that *DEvA* and a static race detection technique such as *Chord* are complementary, and can be used effectively alongside each other.

A deeper analysis sheds further light on why the two techniques yield such different results. EAs are caused by *implicit* concurrency. They are independent of the number of threads in a system and may occur even in single-threaded EBSs. On the other hand, data races only happen in the presence of at least two threads. EAs detected by *DEvA* are potentially more dangerous in that it is harder to track accesses to shared data when those accesses are impacted by a number of factors and actors in a distributed system: the local state of the component processing an event, the network, event routers and dispatchers, and all the other distributed components in the EBS along with their respective states.

Table 4: Results of applying *Chord* on the subject systems

System	Data Races
Planner	14
KLAX	115
DRADEL	0
ERS	1418
Troops	78
Stoxx	31
JMSCHAT	0

5.4 Comparison to Existing Android Analysis

We also evaluated *DEvA* against *CAFA* [29], a recent dynamic analysis technique for detecting *Use-After-Free* (*UF*) EAs in Android applications. In order to compare our approach with *CAFA*, we configured *DEvA* to report only *UF* anomalies among the EAs it detects. We then applied *DEvA* on the same versions of applications used in *CAFA*'s evaluation [29]. Table 5 shows the relevant results. Note that the *CAFA* results did not include the two Android support libraries shown in Table 2, so for this reason, they are omitted from Table 5.

Table 5: Results of *UF* analysis by *CAFA* and *DEvA* (note that *CAFA* is unable to identify the other EAs shown in Table 3)

App Name	*CAFA* All	*DEvA* All	*CAFA* Harmful	*DEvA* Harmful
ToDoWidget	8	1	8	1
FBReader	3	2	1	2
MyTracks	1	5	1	2
ZXing	1	0	0	0
Firefox	4	0	0	0
ConnectBot	1	2	0	1
VLC	0	4	0	1
Browser	1	23	0	1
Camera	1	0	1	0
Music	4	17	2	9

The *CAFA All* and *DEvA All* columns include all *UF* anomalies reported by the two techniques. As discussed by the authors of *CAFA*, certain *UF* anomalies will not actually result in null-pointer exceptions because of checks placed in the code [29]. Using the classification established for evaluating *CAFA*, the *CAFA Harmful* and *DEvA Harmful* columns in Table 5 show the respective numbers of discovered *UF* anomalies that result in actual runtime exceptions.

For all but two of the applications, *DEvA* was able to identify more harmful *UF* anomalies than *CAFA*. For two of the ten applications, *Camera* and *ToDoWidget*, *CAFA* performed better. In the case of *Camera*, *CAFA* reported a single harmful *UF* anomaly that *DEvA* could not detect. After inspecting the source code of this application, we were not able to locate *CAFA*'s reported anomaly. It is possible that the anomaly occurred because of problems in the libraries that *Camera* uses, as *CAFA* did not analyze libraries separately from the application, while *DEvA* did not analyze these particular libraries. *DEvA* also had seven false negatives in the case of *ToDoWidget*. The reason was already discussed above: *ToDoWidget* relies on non-standard mechanisms to access component state.

With regards to false positives, the performance of *DEvA* was also stronger. *CAFA* reported non-harmful *UF* EAs in six out of ten Android applications. As discussed in [29], these are false positives: they are not actual EAs. As configured for this comparative evaluation, *DEvA* reported non-harmful *UF* anomalies in five of the ten applications. However, these false positives were actually

EAs (just not *UF* anomalies): while they will not produce runtime null-pointer exceptions, they do result in nondeterministic application behavior and it is therefore still important for the developers to be aware of them. In summary, out of the combined 25 harmful *UF* anomalies that were discovered by *CAFA* and *DEvA* in the analyzed Android applications, *CAFA* was able to discover 13, while *DEvA* was able to discover 17.

In addition to the comparison study described above, *DEvA* was able to uncover a number of EAs in the two Android libraries we analyzed (recall Table 3). Some of those were, in fact, harmful *UF* anomalies. For example, *DEvA* reported a harmful anomaly in the *android.support.v4* library. This is a known bug [13], but to the best of our knowledge, *DEvA* is the first to identify the bug's root cause. This bug happens in the *android.support.v4.app.DialogFragment* class due to the interaction between events `ActivityCreated` (processed by the method *onActivityCreated*) and `DestroyView` (processed by *onDestroyView*) over the field variable *mDialog*. Because the order of the two events is nondeterministic, *onActivityCreated* is able to access *mDialog* after *onDestroyView* sets it to null.

Overall, the results of this study were very positive. When we configured *DEvA* to identify only *UF* anomalies, *DEvA* was able to uncover more such anomalies than a leading technique in eight of the ten applications. The number of false positives yielded by *DEvA* was lower *and*, while these were not necessarily *UF* anomalies, they were actual EAs. Finally, *DEvA* was able to find the root cause of a previously unsolved bug in a widely used Android library.

5.5 Execution Time

Given our objective of constructing an efficient technique, we designed *DEvA* to generate and use only the subset of system information that is required for event-anomaly analysis. Thus, for example, *DEvA* generates PDGs for component sink methods only, and considers only those CETs that are consumed at a given sink (recall Section 4).

Table 3 shows the execution *Time* required by *DEvA* to analyze each subject system. These measurements include the time used by Soot to generate call graphs, CFGs, and PDGs; the Soot analysis averaged around 42*s* per system. The analysis by *DEvA* ranged between 24*s* and 122*s*, with an average of 55*s*. This execution time is reasonable, especially when we take into account that analyzing each system manually took over 15 hours. As a further comparison, *CAFA*'s execution time for the ten Android applications shown in Table 5 varied between 30 minutes and 16 hours; *DEvA*'s maximum execution time for the same applications was 81 seconds.

5.6 Limitations and Threats to Validity

We have identified three limitations in our evaluation. The first limitation is the largely *manual construction of the ground truth*, aided only by the code-search and visualization features of Eclipse. Human error in this process could affect the reported results. We tried to mitigate this issue by carefully validating the ground truth through inspection, as reflected in the very long time required to construct it. This is further mitigated by the comparison to *CAFA*, which indicates that, with the exception of the single application that relies on a non-standard event-processing mechanism, *DEvA* did not miss any harmful *UF* EAs. The second limitation is that, in analyzing EBSs with ambiguous interfaces, *Eos occasionally yields a small number of false positives* in reporting event types [26]. In turn, this may affect *DEvA*'s results. Note that Eos's inaccuracy did not have any impact on the Android systems we analyzed since *DEvA*'s Algorithm 1, rather than Eos, was used there. The third limitation is that *DEvA assumes that callback methods are defined in the source code* of an Android application to handle events that can occur nondeterministically. If this is not the case, *DEvA* will report false positives.

Beyond the three limitations that are specific to *DEvA*, there are at least two additional limitations that impact the accuracy of static analysis techniques in general: aliasing and detecting feasible paths. *Aliasing* can take the form of data aliasing, in the case of references to variables, or method aliasing, in the case of virtual invocation. Data aliasing can affect data-flow analysis, and virtual invocation can affect call graph generation by introducing false edges between methods. Inaccuracies in data flow analysis and call graph generations can affect the generation of ICFGs and PDGs. As we discussed earlier, to generate ICFGs and PDGs, we relied on Soot, but Soot is occasionally unable to analyze certain systems. In our work, this happened with two components of the KLAX system, for which Soot was unable to provide the PDG. In our subject systems, virtual invocation is used for creating and building events, but not for their consumption. This was the primary reason why aliasing inaccuracies did not cause *DEvA* to report false positives. For the ground truth we tried to detect *infeasible paths* by manual inspection of the subject systems' code. We did not find any infeasible paths that would have resulted in falsely identified EAs by *DEvA*. Checking the feasibility of paths in our subject systems was time consuming, but doable, since the extent of branching after event consumption was low in these systems.

6. RELATED WORK

Our research is related to several approaches that have been proposed to aid the analysis of concurrent, distributed, and loosely-coupled systems. We highlight the most closely related work.

Multi-Threaded Systems. Event anomalies bear resemblance to a well-known problem in multi-threaded systems—*data race*. Data races occur when two or more execution threads access the same variable. If at least one of those threads writes to the variable, then different runtime scheduling of the threads may result in errors. Even though they are well known, data races are hard to find and debug because of the nondeterministic nature of the systems that contain them. Race detection techniques for multi-threaded systems deal with explicit concurrency between threads. On the other hand, *DEvA* is dealing with implicit concurrency that happens as a result of nondeterminism in the order of event consumptions.

There are a number of dynamic race detection techniques for multi-threaded systems [44, 25, 20]. Most of these techniques are based on checking the *happens-before* relationship. This relationship introduces a partial ordering among events in a system [31]. There are also dynamic race detection techniques that are based on checking shared memory references and verifying proper locking (lockset checking) on them [51]. Smaragdakis et al. [52] introduced a sound dynamic analysis technique based on a generalization of *happens-before*, called *causally-precedes*. Dynamic analysis techniques are dependent on the size of the execution trace and cannot be applied to non-executable programs, such as libraries [42].

Static analysis techniques are either flow-sensitive versions of lockset analysis [23], flow-insensitive [48], or path-sensitive model-based [49] techniques. Most static race detection techniques rely on computationally complex analyses such as *may-alias*. Because of this, they may suffer from precision, soundness, or scalability problems. *Chord* [42] is a flow-insensitive analysis that makes use of *conditional-must-not-alias* analysis instead of the *may-alias* analysis to reason about shared memory locations. As described in Section 5, we compared *DEvA* with *Chord* in our evaluation.

Web Systems and Applications. There are techniques that try to solve similar problems to ours in other domains. Zheng and Zhang [57] proposed an approach to extract race problems that may happen in PHP-based web applications. Their approach is a static analysis that reports races that might happen as a result of atomicity violations in accessing external resources such as database tables or files. Their approach also takes explicit concurrency into account.

Paleari et al. [46] proposed an analysis that also looks into atomicity violations occurring due to the inherent concurrency in web applications. Their approach focuses on interactions between an application and a DBMS. It logs SQL queries and looks for specific interleaving patterns in them. Since this is a dynamic analysis, it is unable to detect races unless they occur in an execution.

Petrov et al. introduced WEBRACER [47], a dynamic analysis technique for race detection in client-side web applications. WEBRACER defines the *happens-before* relationship to check for nondeterministic accesses to the same object. Raychev et al.'s EVENTRACER [50] improved WEBRACER to decrease the number of reported harmless races.

Event-Based Systems. P$\overline{an}$ini [34] is a language that uses asynchronous typed events to make designing for modularity and concurrency easier. P$\overline{an}$ini only considers the overlapping that may happen between event-handler methods. Each event-handler consumes a specific type of event, similarly to Android. A common practice in event-based systems is that a component has a single event handler, and inside that handler different events are processed using dispatching and implicit invocation. Neither EVENTRACER [50] nor P$\overline{an}$ini can support such systems. An added disadvantage is that P$\overline{an}$ini is not applicable to legacy systems written in a general-purpose language such as Java.

P [22] is a domain-specific language for event-driven asynchronous systems. A program written in P consists of a set of state machines that interact via events. There is no specific analysis built into P for data races or event anomalies. Instead, a developer has to design policies inside state machines to avoid such situations using P's concepts such as deferred and ignored events.

Android Systems. *CAFA* [29] is a dynamic analysis technique targeting Android applications. Android applications react to events that originate from sources such as the user or Android kernel. *CAFA* considers the causal order between events to check if a race can happen because of the resulting memory accesses. *CAFA* focuses on uncovering a specific class of problems, UF, in which a memory location is accessed by an event without proper reference checking after it is freed by another event. As discussed in Section 5, we compared *DEvA* to *CAFA* over the same set of applications.

DroidRacer [35] is also a dynamic analysis technique for race detection in Android applications. This technique is based on defining *happens-before* relationships between Android events and provides a formal semantic model for Android concurrency by studying the Android framework. However, *DroidRacer* neither supports user-defined asynchronous callback events nor categorizes races based on their harmfulness. Based on the evaluation results, about 50% of callback-related races reported by *DroidRacer* were false positives.

GATOR [56] is a static analysis technique for Android applications that extracts user-defined GUI-based callback methods. This technique provides an ICFG of the analyzed Android application that shows interactions between user-defined GUI-based callbacks, as well as Android's *onCreate* and *onDestroy* life-cycle callbacks. Unlike *DEvA*, *GATOR* does not consider other kinds of user-defined callbacks, such as asynchronous callbacks, nor does it take into account the remaining life-cycle callbacks defined by Android.

7. CONCLUSION

Event-based interaction is an attractive paradigm because of its promise of highly flexible systems in which components are decoupled and can "come and go" as needed. This flexibility comes at a price, however. Debugging EBSs can be particularly onerous. One of the reasons is that the order in which events are sent, received, and processed can be highly unpredictable. This is caused by implicit invocation and implicit concurrency, which, in turn, may result in event anomalies (EAs).

Several existing approaches have targeted different facets of this problem, but they have notable shortcomings that have motivated our research. A long-standing class of data-race detection techniques for concurrent systems is unable to deal with the implicit concurrency and implicit invocation present in EBSs. Approaches that target EBSs directly have tended (1) to rely on dynamic analysis with its inherent limitations, (2) to use new language constructs that cannot be applied to existing systems, or (3) to exploit domain-specific characteristics that lend themselves to simplifying assumptions holding only in relatively narrow settings.

To address these problems, we have developed *DEvA*, a new static analysis technique that targets EBSs. Our empirical evaluation shows that *DEvA* is efficient and scalable, while exhibiting high precision with respect to the manually identified EAs in 20 subject systems. We have demonstrated that *DEvA* can be used in tandem with existing analysis techniques, such as *Chord* [42]. A comparison with *CAFA* [29], a recent technique that targets a specific class of EAs, UF, shows that, in most cases, *DEvA* is able to detect a greater number of such anomalies in a fraction of the time.

There are a number of remaining research challenges that will guide our future work. Some event anomalies may be harmful only in certain scenarios or may even result in acceptable behaviors (e.g., getting location data that is slightly stale when slowly walking). We intend to study the actual harmfulness of event anomalies to ease the decision-making process for EBS developers. To this end, we will explore coupling *DEvA* with our recent technique for extracting accurate models of component behavior from runtime traces [30]. Monitoring, storing, and enabling the replay of event sequences in order to reproduce anomalies is another research direction. The long-term goal of this work is to provide a suite of ready-made remedies to different types of event anomalies, as well as a set of automated wizards to guide developers in selecting and applying these remedies.

8. REPLICATION PACKAGE

We have made available a package allowing independent replication of our results [14]. This package contains *DEvA*'s implementation, the ground truths used for *DEvA*'s evaluation, and all evaluation results presented in this paper. A manual describing how to use *DEvA* is also included. The *DEvA* replication package has been successfully evaluated by ESEC/FSE 2015's Replication Packages Evaluation Committee and found to meet the Committee's expectations.

DEvA uses configuration files to set the environment to perform its analysis. In Figure 5, part of the configuration file for analyzing the *MyTracks* Android application is shown. In this configuration file, at line 1, the maximun heap size for the JVM is set. Line 2 indicates the location of the *jar* file that contains the source code of the system that *DEvA* wants to analyze. *mainComponent* at line 4 indicates the component that is the entry point of *DEvA*'s analysis. *fileNameForRaceResults* at line 7 indicates the location of the output file for the list of event anomalies, and *fileNameForUFRaceResults* at line 9 represents the location of the file for the list of UF anomalies in the case of Android systems. *middlewareID* at line 12 shows the mechanism the system under analysis uses for event-based communications.

```
1  jvm.xmx = 2g
2  process.dir =
3     ${user.dir}/applications/MyTracks.jar
4  mainComponent =
5     com.google.android.apps.mytracks.MyTracks
6
7  fileNameForRaceResults =
8     ${user.dir}/results/MyTracks-detected-EAs.txt
9  fileNameForUFRaceResults =
10    ${user.dir}/results/MyTracks-detected-UF.txt
11
12 middlewareID = Android
```

Figure 5: Portion of the *MyTracks* Configuration used by *DEvA*

Figure 6 shows a view of *DEvA*'s architecture. After providing the configuration file to *DEvA*, the *Initializer* component will load the related part of the *DEvA* to perform the analysis. As we discussed in the paper, *DEvA* acts differently when dealing with Android and non-Android systems. If the system that is under analysis is an Android-based system, the *Initializer* will ask the *Callback Detector* component to extract user-defined callback methods. After the *Callback Detector* finishes its job, it will provide the *EA Detector* with the list of user-defined callbacks as well as the callbacks from the Android life-cycle. After this, the *EA detector* will perform the analysis described in this paper to extract EAs, and will print out the detected EAs in a file that is indicated in the configuration file. It also provides these EAs to the *UF Detector* component to extract use-after-free bugs for Android systems. The *UF* anomalies will be stored in a file that is indicated in the configuration file.

For non-Android systems, the *Initializer* will initiate *Eos* to extract information about event types. *Eos* will provide this information to the *EA Detector*. In turn, the *EA Detector* will extract EAs and will report them in a file that is mentioned in the configuration file.

We have tested *DEvA* on Windows 7, 64 bit, JDK 1.8, IntelliJ IDEA version 13.0.4, with Scala plugin version 0.26.327. We used the Scala compiler version 2.11.6 with language level 2.10. In order to use *DEvA*, it is necessary to first download IntelliJ IDEA (we used version 13.0.4) and install the Scala plugin 0.26.327 on it. After this, one should download the zip file that contains *DEvA*'s source code from [14], unzip the downloaded file in the desired location, and open the *DEvA* project in the IntelliJ IDEA environment by browsing to that location.

For each of the systems that we studied in this paper, we have provided a configuration file in the replication package. Those configuration files can be found in the *configFiles* folder in the project structure view of the IntelliJ IDEA environment. To run *DEvA* on a system with the name *MySystem*, one must open the *MySystem.build.properties* file, copy all of its contents, and paste them in the *build.properties* file. After setting up the *build.properties* file, one just needs to click on the *Run* button in the IntelliJ IDEA environment.

To customize *DEvA* to run on a new system that uses an event-based mechanism not supported by our replication package, one needs to provide the necessary information for *DEvA*. To do this, one must first create a Scala object for the new mechanism in the *edu.usc.softarch.helios.middleware* package that is part of the *Initializer* component. The Scala objects corresponding to the supported mechanisms can be found in that package, and can be used as examples for the new mechanisms. After creating the needed Scala object, one also needs to add the appropriate case line to the *edu.usc.softarch.helios.middleware.Middleware* object.

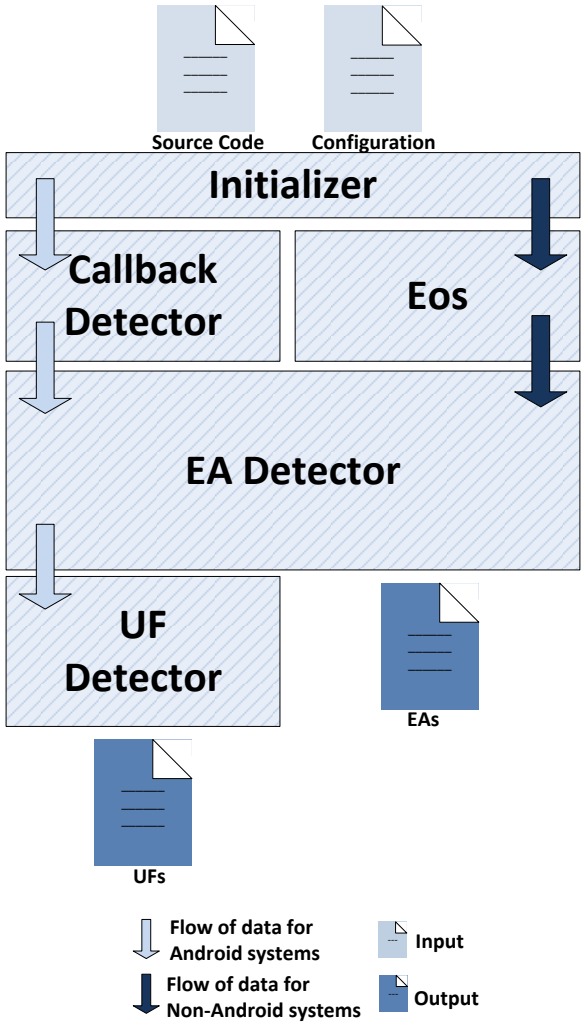

Figure 6: A view of *DEvA*'s architecture

9. REFERENCES

[1] ConnectBot, Version 1.7. http://code.google.com/p/connectbot/, 2011.

[2] ZXing, Version 4.5.1. http://code.google.com/p/zxing/, 2011.

[3] FBReader, Version 1.9.6.1. http://fbreader.org/FBReaderJ, 2012.

[4] MyTracks, Version 1.1.7. http://code.google.com/p/mytracks/, 2012.

[5] ToDoWidget, Version 1.1.7. http://github.com/chrispbailey/ToDo-List-Widget, 2012.

[6] VLC, Version 0.2.0. http://www.videolan.org/vlc/, 2012.

[7] Firefox, Version 25. http://www.mozilla.org/en-US/firefox/fx/, 2013.

[8] JMSChat. http://github.com/julentv/ChatJMS.git, 2013.

[9] Project.net. http://www.project.net/, 2013.

[10] Spring Framework. http://projects.spring.io/spring-framework/, 2013.

[11] UF bug 1070795 in firefox for android. https://bugzilla.mozilla.org/show_bug.cgi?id=1070795, 2013.

[12] UF bug 923407 in firefox for android. https://bugzilla.mozilla.org/show_bug.cgi?id=923407, 2013.

[13] Discussion about the bug in android.support.v4. http://stackoverflow.com/questions/12265611/dialogfragment-nullpointerexception-support-library, 2014.

[14] DEvA's Replication Package. http://www-scf.usc.edu/~gsafi/FSE2015Replication/, 2015.

[15] A. Aho et al. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[16] F. Biscotti et al. Market Share: AIM and Portal Software, Worldwide, 2009. *Gartner Market Research Report*, April 2010.

[17] F. Biscotti and A. Raina. Market Share Analysis: Application Infrastructure and Middleware Software, Worldwide, 2011. *Gartner Market Research Report*, April 2012.

[18] H.-J. Boehm. Nondeterminism is unavoidable, but data races are pure evil. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, RACES '12, pages 9–14, New York, NY, USA, 2012. ACM.

[19] J. Brederekea and R. Gotzheinb. Increasing the concurrency in estelle. 1993.

[20] Y. Cai and W. K. Chan. Loft: Redundant synchronization event removal for data race detection. In *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering*, ISSRE '11, pages 160–169, Washington, DC, USA, 2011. IEEE Computer Society.

[21] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *Software Engineering, IEEE Transactions on*, 27(9):827–850, Sep 2001.

[22] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 321–332, New York, NY, USA, 2013. ACM.

[23] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

[24] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[25] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[26] J. Garcia, D. Popescu, G. Safi, W. G. J. Halfond, and N. Medvidovic. Identifying message flow in distributed event-based systems. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 367–377, New York, NY, USA, 2013. ACM.

[27] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 11–20, New York, NY, USA, 1998. ACM.

[28] M. Hauswirth and M. Jazayeri. A Component and Communication Model for Push Systems. In *ESEC/FSE*. Springer, 1999.

[29] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, New York, NY, USA, 2014. ACM.

[30] I. Krka, Y. Brun, and N. Medvidovic. Automatic Mining of Specifications from Invocation Traces and Method Invariants. In *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE14)*, Hong Kong, China, November 2014.

[31] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[32] T. D. LaToza and B. A. Myers. Developers Ask Reachability Questions. In *ICSE*, 2010.

[33] O. Lhoták and L. Hendren. Scaling java points-to analysis using spark. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 153–169, Berlin, Heidelberg, 2003. Springer-Verlag.

[34] Y. Long, S. L. Mooney, T. Sondag, and H. Rajan. Implicit invocation meets safe, implicit concurrency. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 63–72, New York, NY, USA, 2010. ACM.

[35] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 316–325, New York, NY, USA, 2014. ACM.

[36] S. Malek, N. Medvidovic, and M. Mikic-Rakic. An extensible framework for improving a distributed software

system's deployment architecture. *Software Engineering, IEEE Transactions on*, 38(1):73–100, Jan 2012.

[37] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Trans. Softw. Eng.*, 31(3):256–272, Mar. 2005.

[38] N. Medvidovic et al. The Role of Middleware in Architecture-Based Software Development. *Int. J. of Softw. Eng. and Knowl. Eng.*, 2003.

[39] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 44–53, New York, NY, USA, 1999. ACM.

[40] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, 2002.

[41] G. Mühl et al. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., 2006.

[42] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[43] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 22–31, New York, NY, USA, 2007. ACM.

[44] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM.

[45] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[46] R. Paleari, D. Marrone, D. Bruschi, and M. Monga. On race vulnerabilities in web applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 126–142. Springer, 2008.

[47] 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*, PLDI '12, pages 251–262, New York, NY, USA, 2012. ACM.

[48] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 320–331, New York, NY, USA, 2006. ACM.

[49] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *ACM SIGPLAN Notices*, volume 39, pages 14–24. ACM, 2004.

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

[51] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[52] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 387–400, New York, NY, USA, 2012. ACM.

[53] R. Taylor et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE TSE*, 1996.

[54] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins. A component- and message-based architectural style for gui software. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 295–304, New York, NY, USA, 1995. ACM.

[55] R. Vallée-Rai et al. Soot - a Java Bytecode Optimization Framework. In *Conference of the Centre for Advanced Studies on Collaborative research*, 1999.

[56] 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.

[57] Y. Zheng and X. Zhang. Static detection of resource contention problems in server-side scripts. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 584–594, Piscataway, NJ, USA, 2012. IEEE Press.