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 detecting event anomalies. 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 multithreaded 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 at least one of those is a write access. The resulting implicit invocation makes it difficult to identify the control-flow of the code when an event is received and processed.

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 determine the control-flow of the code.

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
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 D\textsc{eva} (for Detecting Event Anomalies), 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 \textsc{D\textsc{eva}} on 20 open-source event-based applications and libraries from different domains. We compared \textsc{D\textsc{eva}}’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 \textsc{D\textsc{eva}} has high precision and was able to find EAs that the other techniques were unable to detect. \textsc{D\textsc{eva}} 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 \textsc{D\textsc{eva}} and its evaluation. Section 6 summarizes the related work. Section 7 concludes the paper. Finally, Section 8 includes information about \textsc{D\textsc{eva}}’s implementation and the replication package we have made available.

2. MOTIVATING EXAMPLE

As a simple example, consider My\textsc{Tracks}, an Android application developed by Google. My\textsc{Tracks} 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: on\texttt{LocationChangedAsync} (line 5) processes LocationChanged-Async events and on\texttt{Destroy} (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.\footnote{Note that invoking on\texttt{Destroy} is not equivalent to performing garbage collection in Android. on\texttt{Destroy} 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.}

In the case of My\textsc{Tracks}, the EA occurs when both Destroy and LocationChangedAsync events occur but Destroy is processed first. The on\texttt{LocationChangedAsync} handler will free memory by setting the private class variable provider\texttt{Utils} to null. At some point shortly thereafter, on\texttt{LocationChangedAsync} will attempt to access provider\texttt{Utils}, 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, \textsc{D\textsc{eva}}, 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].

```java
public class TrackRecordingService
extends Service
 implements LocationListener {
...
private void onLocationChangedAsync(
Location location) {
... Location lastRecordedLocation =
providerUtils.getLastLocation();
...
}
public void onDestroy() {
log.d(TAG,
"TrackRecordingService.onDestroy");
... providerUtils = null;
... }
}
```

Figure 1: Excerpt from the My\textsc{Tracks} Android application

3. FOUNDATIONS

In this section we define the underlying concepts and terminology that we will use to describe \textsc{D\textsc{eva}} 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 Load\textsc{Balancer} component shown in Figure 2. Load\textsc{Balancer} 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 Load\textsc{Balancer} through a New\textsc{Request} event. At lines 6 and 7 of Figure 2, Load\textsc{Balancer} 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 Load\textsc{Balancer} receives a Set\textsc{Limit} event. At lines 9-13, Load\textsc{Balancer} 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 Load\textsc{Balancer} 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\texttt{dependent} on node Y if execution of Y determines whether X will be executed. Consider the CFG of the handle method in the Load\textsc{Balancer} 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\texttt{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 Load\textsc{Balancer} ex-
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 2: Event-based load-balancing component

Figure 3: ICFG of LoadBalancer

2. retrieveSpecificEventInfo() represents methods whose names indicate the method the methods directly obtain 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. A 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...
may occur: the consumption order of the two events may be depend-ent 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 ∈ Components, we define Fc 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 ∈ Fc due to event types e1, e2 ∈ CETc iff the following conditions hold:

1. e1 ̸= e2
2. ∃{X1, ..., Xp} ⊆ Nodes(ICFG) | 1 ≤ i < p, Xi+1 is control or data dependent on Xi ∧ Xi ∈ cerExtract(e1)
3. ∃{Y1, ..., Yq} ⊆ Nodes(ICFG) | 1 ≤ j < q, Yj+1 is control or data dependent on Yj ∧ Yj ∈ cerExtract(e2)
4. Xp is a use or definition of f
5. Yq 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 (f, n) in which f ∈ Fc is a field of component c (recall that Fc is the set of all fields of component c) and n ∈ Nm is a node in a method m’s control-flow graph (Nm 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, FUsem that contains field uses and FDefsem that contains field definitions.

\[
\text{FDefsem} = \{(f,n)|f \in Fc \land n \in Nm \land n \text{ defines } f\}\]
\[
\text{FUsem} = \{(f,n)|f \in Fc \land n \in Nm \land n \text{ uses } f\}\]

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 (preLmt, 10) and (lmt, 11) are added to FDefsem. Also since there are two uses of field lmt at nodes 10 and 12, and a use of field preLmt at node 12 tuples (preLmt, 12), (lmt, 10) and (lmt, 12) are added to FUsem.

Our recently developed technique, Eos [26], is able to identify CETs in EBSs that use ambiguous interfaces. Running Eos on the
**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 \( \{ \langle \text{NewRequest} \rangle, \langle \text{SetLimit} \rangle \} \).

### Algorithm 1: Callback sink extraction

<table>
<thead>
<tr>
<th>Methods</th>
<th>FDef</th>
<th>FUse</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>( { \langle \text{preLmt}, 10 \rangle, \langle \text{lim}, 11 \rangle } )</td>
<td>( { \langle \text{preLmt}, 12 \rangle, \langle \text{lim} \rangle } )</td>
</tr>
<tr>
<td>manageCurrentLoad</td>
<td>( { \langle \text{curLoad}, 18 \rangle } )</td>
<td>( { \langle \text{curLoad}, 17 \rangle, \langle \text{curLoad}, 18 \rangle } )</td>
</tr>
</tbody>
</table>

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

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 stores the `(c, i)` 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 `(c, t, i)` 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`, such that `t`’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 `(t, t, i)` 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` 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 (CETs), the call graph of the component (CGs), and the FUse and FDef sets for each method in `c`. Note that CETs, 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 `(f, n, e)` 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`, 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...
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 \( \text{CET} \); (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 \( \text{CET} \) node for the current \( \text{CET} \) (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 \( \text{StoM} \) (line 5). If \( k \) is dependent on a \( \text{CET} \) node, then this relationship is added to the output set (lines 14 and 15). The intuition here is that the field access \((f, n)\) can be reached via a statement \( k \) that is itself dependent on a \( \text{CET} \) 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 \( F\text{Def}_{\text{handle}} \), listed in Table 1, that are control or data dependent on a node that contains a \( \text{CET} \) statement. Consider \( \langle \text{Int}, 11 \rangle \) in the PDG. Node 11 is connected to node 9, which contains a node with a \( \text{CET} \) statement for the \( \text{SetLimit} \) event; so \( \langle \text{Int}, 11, \text{"SetLimit"} \rangle \) will be added to the return set. Next, \( \langle \text{preLmt}, 10 \rangle \) is also dependent on node 9, so \( \langle \text{preLmt}, 10, \text{"SetLimit"} \rangle \) 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. \( \text{StoM} \) (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 \( \text{CET} \) statement of the NewRequest event. Therefore, any definition or use inside the manageCurrentLoad method is dependent on this event. Since \( F\text{Def}_{\text{manageCurrentLoad}} \) contains \( \langle \text{curLoad}, 18 \rangle \), our algorithm will add \( \langle \text{curLoad}, 18, \text{"NewRequest"} \rangle \) 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:

\[
\begin{align*}
\text{ConsumedToDef}_{\text{loadBalancer}} & = \{ \\
\langle \text{Int}, 11, \text{"SetLimit"} \rangle, \langle \text{curLoad}, 18, \text{"NewRequest"} \rangle, \\
\langle \text{preLmt}, 10, \text{"SetLimit"} \rangle \} \\
\text{ConsumedToUse}_{\text{loadBalancer}} & = \{ \\
\langle \text{curLoad}, 17, \text{"NewRequest"} \rangle, \langle \text{curLoad}, 18, \text{"NewRequest"} \rangle, \\
\langle \text{Int}, 17, \text{"NewRequest"} \rangle, \langle \text{Int}, 10, \text{"SetLimit"} \rangle, \langle \text{Int}, 12, \text{"SetLimit"} \rangle \}
\end{align*}
\]

4.3 Joining

\text{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 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 \( \text{ConsumedToDef} \) and \( \text{ConsumedToUse} \), 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 \( \text{Int} \) at line 11 of Figure 2. This definition dominates the use of \( \text{Int} \) at line 12. Therefore, the use of \( \text{Int} \) at line 12 will be removed from consumedToUse_{\text{LoadBalancer}}, after which we will have:

\[
\begin{align*}
\text{ConsumedToUse}_{\text{loadBalancer}} & = \{ \\
\langle \text{curLoad}, 17, \text{"NewRequest"} \rangle, \langle \text{curLoad}, 18, \text{"NewRequest"} \rangle, \\
\langle \text{Int}, 17, \text{"NewRequest"} \rangle, \langle \text{Int}, 10, \text{"SetLimit"} \rangle, \\
\langle \text{preLmt}, 12, \text{"SetLimit"} \rangle \}
\end{align*}
\]

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
event types is added to the output set. To illustrate the joining algorithm, consider the $\text{ConsumedToDef}_{\text{LoadBalancer}}$ set, reported at the end of Section 4.2, and $\text{ConsumedToUse}_{\text{LoadBalancer}}$ after line 4 of Algorithm 4, reported earlier in this section. There is one CET-field dependency that defines $\text{lmt}$. This definition happens at line 11 and is caused by the $\text{SetLimit}$ event. There is also one CET-field dependency that uses $\text{lmt}$. This happens at line 17 and is caused by the $\text{NewRequest}$ 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 $\text{LoadBalancer}$ is, therefore, \{ ($\text{SetLimit}$, “Limit”), (“NewRequest”) \}.

5. Evaluation

We have empirically evaluated $\text{DEvA}$ (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 $\text{Project.net}$, 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 $\text{Project.net}$’s event-based portion, totaling around 10 KsLOC, as an input to $\text{DEvA}$. However, to pinpoint this portion, we had to analyze the entire system. $\text{DEvA}$ 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 $\text{DEvA}$, as described in Section 3.2.

We ran $\text{DEvA}$ on a quad-core Intel i7 2.80GHz system with 8GBs of memory, running Windows 7 Professional.

5.2 Accuracy of EA Detection

To assess $\text{DEvA}$’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 $\text{DEvA}$ 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 $\text{DEvA}$ was counted as a false negative. Our results are summarized in Table 3. For all but three of the systems, $\text{DEvA}$ was able to find each anomaly identified in the ground truth. The three exceptions were $\text{KLAX}$, for which $\text{DEvA}$ yielded 2 false negatives, $\text{ToDoWidget}$, with 12 false negatives, and $\text{MyTracks}$, with 1 false negative. $\text{DEvA}$ 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.

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

| App Name       | App Type     | SLOC  | Event Mechanism |
|----------------|--------------|-------|-----------------
| Planner        | AI Planner   | 6K    | c2.fw [38]      |
| KLAX           | Arcade Game  | 5K    | c2.fw [38]      |
| DRADELE        | Software IDE | 11K   | c2.fw [38]      |
| ERS            | Crisis Response | 7K  | Prism-MW [37]   |
| Troops         | Simulator    | 9K    | c2.fw [38]      |
| Stoxx          | Stock Ticker | 6K    | DEBECA [41]     |
| JMSCHAT        | Chat System  | 12K   | ActiveMQ, Java events |
| Project.net    | Project Mgmt | 247K  | Sprng [10]      |
| ToDoWidget     | ToDo List Recorder | 2K  | Android Events  |
| FBReader       | Book Feed Reader | 3K  | Android Events  |
| MyTracks       | Location Tracker | 13K | Android Events  |
| ZXing          | Barcode Scanner | 16K | Android Events  |
| Firefox        | Browser App  | 58K   | Android Events  |
| ConnectBot     | SHH Client   | 23K   | Android Events  |
| VLC            | Media Player | 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  |
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.

### 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)

<table>
<thead>
<tr>
<th>App Name</th>
<th>CAFA All</th>
<th>DEvA All</th>
<th>CAFA Harmful</th>
<th>DEvA Harmful</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToDoWidget</td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>FBReader</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MyTracks</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ZXing</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Firefox</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ConnectBot</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>VLC</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Browser</td>
<td>1</td>
<td>23</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Camera</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Music</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>

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 42s per system. The analysis by DEvA ranged between 24s and 122s, with an average of 55s. 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.

33
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. Paffini [34] is a language that uses asynchronous typed events to make designing for modularity and concurrency easier. Paffini 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 Paffini can support such systems. An added disadvantage is that Paffini 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 callback methods. This technique provides an ICFG of the analyzed Android application that shows interactions between user-defined callback methods, 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 maximum 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 fileNameForUFResults 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.

```plaintext
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 fileNameForUFResults -
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.
ConnectBot, Version 1.7.

ZXing, Version 4.5.1.

FBRReader, Version 1.9.6.1.

MyTracks, Version 1.1.7.

ToDoWidget, Version 1.1.7.


Firefox, Version 25.


Spring Framework.

UF bug 1070795 in firefox for android.

UF bug 923407 in firefox for android.

Discussion about the bug in android.support.v4.


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


