ER Catcher: A Static Analysis Framework for Accurate and Scalable Event-Race Detection in Android

Navid Salehnamadi, Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek
School of Information and Computer Sciences
University of California, Irvine, USA
{nsalehna,aalshayb,iftekha,malek}@uci.edu

ABSTRACT
Android platform provisions a number of sophisticated concurrency mechanisms for the development of apps. The concurrency mechanisms, while powerful, are quite difficult to properly master by mobile developers. In fact, prior studies have shown concurrency issues, such as event-race defects, to be prevalent among real-world Android apps. In this paper, we propose a flow-, context-, and thread-sensitive static analysis framework, called ER Catcher, for detection of event-race defects in Android apps. ER Catcher introduces a new type of summary function aimed at modeling the concurrent behavior of methods in both Android apps and libraries. In addition, it leverages a novel, statically constructed Vector Clock for rapid analysis of happens-before relations. Altogether, these design choices enable ER Catcher to not only detect event-race defects with a substantially higher degree of accuracy, but also in a fraction of time compared to the existing state-of-the-art technique.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging

KEYWORDS
Program Analysis, Android, Concurrency, Event-Race Detection

1 INTRODUCTION
Modern mobile frameworks promote the development of highly concurrent software applications, or “apps” for short. In the case of Android, concurrency is ingrained in all facets of app behavior: (1) an app’s components run within their own threads, (2) components can simultaneously interact with components within and outside of the app by exchanging Intent messages, (3) the components at any point in time may receive lifecycle (e.g., onStart(), onPause()) and system (e.g., location change, battery low) callbacks without any guarantees as to the order in which they may occur. To aid the developers with development of concurrent software, Android provisions several new concurrency constructs in the form of libraries, such as AsyncTask and Looper. Nevertheless, concurrency is a major source of confusion for developers [26] and remains among the top 5 reasons for defects in Android apps [51].

The conventional techniques for detection of concurrency issues in Java, such as data-race defects [15, 30, 34, 46], are not readily applicable for Android [10, 38]. They neither explicitly consider the event-driven model of app behavior, nor support the new concurrency constructs in Android. More recently, researchers have investigated both dynamic [10, 20, 29] and static [17, 22, 45] analysis techniques for detection of concurrency issues in Android. Dynamic analysis techniques proposed so far are limited in their capability, as they miss true event races due to their limited coverage of the app behavior [22]. Existing static analysis techniques [17, 22, 45] fail to accurately identify many event races due to (1) imprecise modeling of concurrency behavior in Android, and (2) adoption of analyses that are innately flow-, context-, thread-insensitive. Moreover, the existing static analysis techniques are slow, e.g., nAndroid [17] takes about 50 minutes on average to analyze an app. Besides that, these works support only a small and fixed set of concurrency libraries, e.g., SARD [45] only considers asynchronous invocations by Handler, Activity, and Thread APIs, and cannot be easily extended to support other libraries. Furthermore, the tools realizing these techniques are either unavailable ([22, 45]), or closed-source ([17]).

In this paper, we introduce ER Catcher, a static analysis framework for effective detection of event-race defects in Android apps that aims to overcome the shortcomings of prior works. Three novel concepts set our work apart from the prior techniques and allow us to succeed where others have failed.

First, ER Catcher relies on a new type of summary function, called Concurrency-aware Summary Function (CSF), for modeling the concurrent behavior of methods in both Android apps and libraries. The CSF for each app method is automatically extracted, while the set of CSFs representing concurrent behavior of methods comprising a library are manually constructed as a one-time effort. ER Catcher processes methods in an app’s call graph in a reverse topological order, thereby improving the performance of analysis, i.e., eliminating the need to reanalyze the same method multiple times. Furthermore, by modeling each library (e.g., AsyncTask, Handler) as a set of CSF specifications, the implementation of ER Catcher is completely separated from its support of Android libraries. This, in turn, allows one to add support for new or modified Android libraries by simply providing ER Catcher with the proper CSF specifications, and without requiring changes to the ER Catcher’s implementation.

Second, ER Catcher builds its analysis on an abstraction representation of app, called Context and Concurrency-aware Call Graph (CG). In CG, each method call is represented in terms of its three execution states: invocation, start, and end. The states are connected
through edges that distinguish between synchronous and asynchronous interactions. Unlike a conventional call graph, $G(V,E)$ is aware of the running thread of each method. The fine-grained representation of app behavior in $G(V,E)$ enables ER Catcher to perform flow-, context-, and thread-sensitive analysis. This increased sensitivity of analysis addresses the limitations imposed by imprecise modeling in prior techniques that affect their accuracy.

Third, ER Catcher employs a novel, statically computed data structure, called Static Vector Clock (SVC), to efficiently analyze the happens-before relations in an app. SVC is inspired by the concept of Vector Clock [24] from the domain of distributed systems for determining the order of events in such systems. SVC enables ER Catcher to quickly query happens-before relations for a subset of events suspected to be involved in an event race, thereby making its analysis substantially faster than prior techniques.

Besides the above, ER Catcher is the first of its kind, completely open-source static analysis tool for event-race detection in Android [2]. Based on our extensive empirical evaluations, ER Catcher outperforms the state-of-the-art static event-race detector, nAdroid [17], in terms of accuracy (13% more precise) and performance (12 times faster). Moreover, our experiments show ER Catcher is both practical and scalable—capable of analyzing 90% of 500 randomly selected real-world apps from F-Droid repository [3] in under 5 minutes per app.

The remainder of this paper is organized as follows. Section 2 provides an example of event-race defect in Android that is used to describe the approach. Sections 3 describes the details of our approach, while Section 4 presents our experimental evaluation. Section 5 summarizes the related work and finally Section 6 concludes and discusses the future work. The tool, research artifacts, and proofs of soundness can be found on the companion website [2].

2 ILLUSTRATIVE EXAMPLE

Figure 1a shows the code snippet of a “screen time” tracking app that logs the time a user has spent on an app on an external server and displays it on the screen. TrackTimeActivity is created when the app starts. It immediately sends a request to the server to record the initial activation time. From then on, whenever the user clicks on recordButton, the app requests the server to record the time and displays the elapsed time on screen.

According to the code in Figure 1a, onCreate method asynchronously invokes initTime.run(initTime in short) and RecordTimeTask doInBackground method (lines 8–9). initTime method runs on the UI (main) thread, since it is initiated by a Handler of main Looper. Looper is a FIFO queue wrapper for a thread in Android, enabling tasks to be queued for execution by the thread. Handler is responsible for enqueuing tasks in the associated Looper using the post method. Here, the task is initTime method, which simply initializes startTime, elapsedTime, and textView variables (lines 20–24). The executeOnExecutor method called on line 9 is an AsyncTask library API that eventually results in the invocation of doInBackground method in SERIAL_EXECUTOR, i.e., a construct that enqueues tasks and executes them sequentially in a single thread. doInBackground method sends currentTime to the server through a blocking send method (line 29). Once doInBackground is finished (the data has been sent

![Diagram](a) The code of the app

![Diagram](b) The timeline of the event race

Figure 1: An event race example in Android
to the server), RecordTimeTask invokes onPostExecute (onPost in short) on the UI thread and passes the currentTime to it. This method sets startTime (if it has not been set so far), updates elapsedTime, and displays it on timeView widget (lines 32-37).

The UI event handler for recordButton is onClick, which executes RecordTimeTask. Point to note, dIB invokes in onClick (lines 11-15) runs on THREAD_POOL_EXECUTOR, i.e., a construct that collects tasks and executes them in parallel without any order.

To distinguish between the invocations of RecordTimeTask through onCreate and onClick, we add an index to them: dIB-0 and onPostExecute-0 when invoked by onCreate, and dIB-1 and onPostExecute-1 when invoked by onClick.

The logically correct behavior for this app should follow a specific execution order. Namely, initTime should happen before onPostExecute-0, and onPostExecute-0 must happen before onPostExecute-1 in order to show the correct elapsed time on the screen. This assumption, however, can be violated, since there is no happens-before relation between onPostExecute-0 and onPostExecute-1, as depicted in Figure 1b. Due to network communication latency, the execution time of send method (line 29) is non-deterministic. It is thus possible that dIB-0 finishes after dIB-1, resulting in onPostExecute-1 method to execute prior to onPostExecute-0. Consequently, startTime is set to the time that recordButton is clicked rather than the time onCreate is invoked, which means a negative value for time is shown on screen. This issue can be resolved by uncommenting line 13 to execute all RecordTimeTasks in a serial mode. That would enforce onPostExecute-1 to be executed after onPostExecute-0.

A static analysis approach for effectively detecting event-race defects in Android needs to be flow-, context-, and thread-sensitive:

- **Flow-Sensitive**: A flow-insensitive approach would ignore the statement ordering, thereby reporting false positives. For example, the only reason initTime happens before onPostExecute-0 is that initTime is invoked before dIB-0 in the body of onCreate (lines 8 and 9).
- **Context-Sensitive**: A context-insensitive approach does not distinguish between the invocation of two methods under different execution contexts, thereby misses true event races. In the above example, there is one onPostExecute method, but it can be executed under different contexts (through onCreate or onClick). Without considering the context of execution, the occurrence of such an event race could not be detected.
- **Thread-Sensitive**: A thread-insensitive approach either falsely assumes all tasks are executing in one thread, or conservatively assumes no tasks share threads. A thread-insensitive approach that assumes all tasks are running in the same thread would miss true event races shown in the example of Figure 1a. On the other hand, if both dIB-0 and dIB-1 are actually running in the same thread (in the fixed version of Figure 1a), a thread-insensitive approach that assumes no tasks share threads cannot detect the actual happens-before relation and reports a false positive.

Furthermore, a practical solution needs to be (1) **Scalable**, to make it suitable for execution during the development process, (2) **Extensible**, to accommodate the evolution of concurrency constructs in modern mobile platforms, and (3) **Available**, to allow practitioners to make use of it, and to enable researchers to build on top of it.

The aforementioned criteria constitute the requirements for ER Catcher, described next.

## 3 APPROACH

Figure 2 shows the components of ER Catcher. We explain the details of each component in this section. For brevity, we elide the details of call-graph generation and points-to analysis, since we employ existing off-the-shelf solutions [7, 25].

### 3.1 Concurrency Augmentation

Consider the implementation of the illustrative app (Figure 1a), which overrides doInBackground and onPostExecute methods of AsyncTask library. Here, most of the semantics of concurrent behavior does not exist in the source code of the app itself. First of all, there is no explicit invocation of these methods. Moreover, the code itself does not reveal a particular order of execution. Besides that, these methods may run in different threads. However, from our understanding of the AsyncTask library, we know that by invoking executeOnExecutor method, doInBackground will be implicitly invoked on the thread which can be determined by the first parameter (e.g., SERIAL_EXECUTOR), followed by onPostExecute that runs on the UI thread. Although this information is embedded in the implementation of libraries, including the libraries in the analysis would make the approach slow and inefficient. More importantly, this information may not be extracted precisely due to the limitations of static analysis. To overcome this challenge, we introduce the notion of Concurrency-aware Summary Function (CSF).
The concept of CSF is inspired by summary-based static analysis [18, 40], where for each method a model representing the method’s behavior is constructed and subsequently reused to expedite the analysis. We adapted this technique to model only concurrency-relevant information, e.g., the synchronous versus asynchronous nature of invocation calls, or the type of thread that an outgoing method call will execute on.

As shown in Figure 3, a CSF models a method call in terms of its three execution states: invocation, start, and end. We formally define CSF of a method m to be a Directed Acyclic Graph (DAG) represented as (N, E). Here, N consists of a start node and an end node indicating the initiation and termination of m respectively, and zero or more invocation nodes indicating the methods that are called as a result of execution of m. E is the set of directed edges such as (n₁, n₂), where n₁ dominates n₂, i.e., all paths starting from the entry of a method reaching n₂ have to pass through n₁. The edges of a CSF capture the intra-procedural flow that is missing in a conventional call graph.

An invocation node has five attributes (Name, Type, RT, Task, TT), where Name is the name of the callee’s method, Type indicates the type of invocation (synchronous or asynchronous), RT is the running thread identifier that the callee method will run on, and Task is the callback method that will eventually run on the task thread (TT) at some point in future.

We augment the call graph of an app, obtained using FlowDroid [7], with the concurrency behavior of its methods modeled in CSFs to arrive at a Concurrency-aware Call Graph, called C³G. Figure 3 depicts a subset of the C³G for the illustrative example (recall Figure 1a). Each large white box corresponds to a node in the call graph, while the internals of each box represent the CSF. The top row in Figure 3 shows the entryPoint method, which similar to prior work (FlowDroid [7]) is artificially created to emulate the lifecycle of an Activity.

The second row consists of the defined/overridden methods in the app. For instance, the CSF of onCreate consists of start, two invocation, and end nodes (due to limited space, some invocations, such as findViewById, are not shown). The first invocation node in onCreate represents line 8 in Figure 1a that invokes post (Name) method synchronously (Type) on onCreate’s thread (RT) and eventually initTime (Task) will be executed on UI thread (TT). Note that, certain contextual information, such as specific threads executing some of the methods, are unknown at this point, denoted in bold as UNK, e.g., UNKᵣ in post invocation node is the running thread of onCreate that is not yet resolved.

The bottom row depicts the CSFs of library methods called by the app. The CSF of executeOnExecutor in AsyncTask library shows that this method first synchronously invokes onPreExecute on its own running thread, UNKᵣ, then asynchronously invokes interDoInBG method on another thread, UNKᵣᵗ. Moreover, the last invocation node passes UNKᵣ as the task thread (TT), to model the fact that the result of computations performed in the background should eventually be returned to the main caller’s thread. The other CSF, interDoInBG, first synchronously invokes doInBackground, and next asynchronously invokes onPostExecute on UNKᵣᵗ. Note that, the actual implementation logic of AsyncTask library is much more complex than its corresponding summaries, e.g., it consists of several exception handlers and various classes, such as WorkerRunnable. However, the summaries are sufficiently modeling the concurrent behavior of the library for our analysis with a few CSF nodes.

To construct the C³G, we expand each method in the call graph of an app with its corresponding CSF representation. The CSFs of concurrency library methods are provided as an external artifact, while the CSFs of app methods are automatically extracted. Each concurrency library method is associated with a CSF specification and a helper function that determines the attributes of its invocation nodes.
For each app method, the CSF nodes and edges are extracted using the control-flow graph of its body. Next, for each invocation node of a concurrency library method encountered in the app logic, the corresponding helper function determines the attributes such as RT and TT. Finally, to capture the order of calls more accurately, we change the source of call-graph edges to invocation nodes.

We model the concurrent behavior of Android components (i.e., Activity, Service, and BroadcastReceiver) using CSFs as well. An artificially created EntryPoint method emulates the invocation of a component’s callback methods according to the lifecycle of the corresponding component type. We automatically generate the CSF of EntryPoint methods similar to app methods, except the callbacks are invoked asynchronously in main thread. Currently, ER Catcher supports the following concurrency-related constructs in Android: Thread, Looper, Handler, AsyncTask, IntentService, ServiceConnection, and lifecycle methods of Activity, Service, and BroadcastReceiver. Note that ER Catcher analyzes both app code (written by the app developers) and app libraries, e.g., an advertisement library. However, if a library introduces its own concurrency mechanism, ER Catcher requires its corresponding CSF for precise analysis. For more details on the implementation of Library CSF, please visit the ER Catcher’s website [2].

3.2 Event-Race Candidate Detection

In parallel to the construction of fine-grained models representing the concurrent behavior of an app, ER Catcher analyzes the app to identify a list of all candidate event races (recall Figure 2). The crude event races identified at this stage are then filtered in the subsequent steps. An event race is defined as a triplet (Stmt₁, Stmt₂, F), where Stmt₁ and Stmt₂ are executing in methods M₁ and M₂, and F is a field (representing a memory location statically). These triplets have to satisfy the following properties: (1) Stmt₁ and Stmt₂ have access to memory location F, and at least one of the accesses is a write; and (2) there is no happens-before relation between Stmt₁ and Stmt₂.

For example, the illustrative app of Figure 1a has two potential event races involving the memory location elapsedTime: (initTime:22, onPostExecute:33, elapsedTime) (onPostExecute:33, onPostExecute:35, elapsedTIme)

In this component, we do not have any information about the context of methods; therefore, we use a conservative approach and presume no happens-before relation exists between the statements. We use an off-the-shelf points-to analysis technique (SPARK [25]) to determine the statements in two methods are accessing the same memory location.

3.3 Context Augmentation

For precise event-race detection, we need to account for the execution context (recall Section 2). To that end, we need to augment C²G with contextual information to resolve the unknown entries such as UNKRT and UNKTT in CSFs. One naive approach for determining the contexts of all methods is to traverse the call graph along the edges, starting from the entry point. However, such an approach does not scale, because the number of potential paths is \(O(2^n)\), where \(n\) is the number of methods. To address this challenge, we use dynamic programming together with two filters (Race Involvement and Synchronous Substitution), which substantially prune the analysis and memory space. Once the contexts are determined, the remaining parts (running and task threads) can be determined by simply propagating threading information through the invocation caller sites. The result of this component is Context- and Concurrency-aware Call Graph, called C²G.

Figure 4 depicts the process of determining the contexts for a subset of the illustrative example. For brevity, only the CSF nodes involved in the context augmentation are shown here. The edges annotated with S are synchronous call-graph edges.

We process the methods of C²G, shown in Figure 4a, in a reverse topological order. If during this process, a method can be in the visited, visiting, or non-visited stages. For each visiting method \(m\), we first apply Race Involvement filter: if neither \(m\) nor any of \(m\)'s descendants are involved in an event race, we do not process \(m\), e.g., method findViewById in Figure 4b is pruned. This filter reduces the analysis space drastically. The Event-Race Candidate Detection component, discussed earlier, provides us with a list of methods that may be involved in an event race.

If \(m\) passes this filter, for each incoming edge to its start node, we make a copy of \(m\) and all of its descendants to make its incoming edges unique. We then connect each edge to the start node of its corresponding copy (making the incoming edge of \(m\)'s start node unique). For example, in Figure 4b, the start node of method onPostExecute has only one incoming edge; therefore, there is only one copy of onPostExecute in Figure 4c.

If the invocation of a method is synchronous, e.g., onCreate \(\rightarrow\) executeOnExecutor, it does not impact the concurrent behavior of the app. In this situation, we apply Synchronous Substitution filter by (1) removing the callee method, (2) connecting the outgoing edges of caller method to the callee method’s children, (3) annotating these edges with the caller to maintain the sequence of invocations. For example, Figure 4c shows Synchronous Substitution of executeOnExecutor. The method executeOnExecutor is eliminated and replaced with two annotated edges, representing alternative execution contexts. This filter substantially improves the utilized memory and processing time of subsequent steps. Note that since we are processing the methods in the app call graph in a reverse topological order, all descendants of \(m\) are already visited.

Figure 4d shows the final stage of context augmentation. As mentioned earlier, for each incoming edge of intermDoInBG’s start node (the edges from onCreate and onClick), a copy of intermDoInBG and its descendants (onPostExecute) is created, making the incoming edge of each node unique.

Once the contexts of all methods are determined, the remaining two parts (running and task threads) can be determined by propagating the thread information starting from the entry point. The thread of the entry point is already known, since it runs on the UI thread and there is no task thread (as depicted in Figure 3). Using the C²G edges we propagate these information. For example, in Figure 3, once the caller sites of executeOnExecutor are resolved to be onCreate and onClick, the specific thread
used to execute tasks in the background, as modeled by \( UNK_{RT} \) in \( \text{intermDoInBG} \), can be determined. In the case of reaching \( \text{executeOnExecutor} \) through \( \text{onCreate} \), \( UNK_{RT} \) of \( \text{intermDoInBG} \) is determined to be the SERIAL\_EXECUTOR, while in the alternative case of reaching \( \text{executeOnExecutor} \) through \( \text{onClick} \), \( UNK_{RT} \) of \( \text{intermDoInBG} \) is determined to be THREAD\_POOL\_EXECUTOR.

At this stage, all contextual information is determined. The final \( C^3G \) of the example in Figure 1a can be seen in Figure 5.

### 3.4 Happens-Before Analysis

For precisely determining event races, we need to determine the happens-before relation, denoted as \(<\), among the execution states of methods (i.e., start, invocation, and end). To that end, we developed an efficient approach for statically reasoning about the happens-before relations in an event-driven program, called Static Vector Clock (SVC). Our solution is inspired by the notion of Vector Clock [24]—a well-known algorithm for determining partial ordering of events in a distributed system, whereby timestamps associated with messages exchanged among the distributed nodes are used to establish a logical clock. The conventional Vector Clock relies on the following basic principle to adjust the logical clock of a distributed system: the transmission of a message must precede its receipt in time. Similarly, in our work, we leverage the execution states of methods (nodes) represented in \( C^3G \) to construct a logical clock. We rely on three principles: (principle 1) for a given thread of execution, a method’s invocation state must precede its start state, and a method’s start state must precede its end state; (principle 2) a method’s invocation state precedes another method’s invocation state if the call statement of the former dominates that of the latter in the control-flow graph; and (principle 3) in a FIFO thread, a method’s end state precedes the start state of another method, if the former invocation state precedes the latter invocation state. The proof of correctness for these principles can be found on the companion website [1].

Figure 5 illustrates how we compute the value of SVC for each node using the \( C^3G \) of the illustrative app. Each method call is represented in terms of three nodes with suffixes .i (invocation state), .s (start state), and .e (end state). Moreover, each node has a numerical identifier, e.g., \( \text{onCreate}\.i \) is the invocation state of \( \text{onCreate} \) identified as node 1. The nodes are divided into three zones representing the threads that they are running on. For the sake of clarity, the main entry point method that invokes \( \text{onCreate} \) and \( \text{onClick} \) methods is not shown here.

The value of SVC for each node is shown above it as a vector. Formally, in an event-based system with \( |T| \) threads, the SVC value of a node \( n \) is a vector \( \text{SVC}(n) = \langle S_1, \ldots, S_T \rangle \), where \( S_i \) is the minimized set of nodes occurring in thread \( T_i \) before node \( n \). \( S_i \) is minimized when there does not exist any happens-before relation between any of its members. For example, the SVC value of node 20, meaning the last nodes occurring before node 20 in SerialUIThread, and ThreadPool are 5, 19, and 13, respectively. Here, since SVC is minimized, node 14 is not in SVC(20), even though it happens before node 20, i.e., the invocation state of \( \text{onPost} \) occurs before its end state. Because \( 14 < 19 < 20 \), the inclusion of 19 in the SVC is sufficient. On the other hand, SVC(9)2 contains two nodes \( \langle (3, 6) \rangle \), since there is no happens-before relation between the invocation state of \( \text{onClick} \) and the end state of \( \text{onCreate} \).

Given the SVC of an app, we can quickly determine the existence of happens-before relation among its nodes:

1. **Theorem 1.** Node \( a_i \) happens before node \( a_n \), if there exists a sequence of nodes \( (a_1, \ldots, a_n) \) such that \( 1 \leq i < n \) and \( a_i \in \text{SVC}(a_{i+1}) \).

2. **Theorem 2.** Two nodes \( a \) and \( b \) may happen without any order if there exists two threads, \( t_1 \) and \( t_2 \), such that \( \text{SVC}(a)_{t_1} = \text{SVC}(b)_{t_1} \) and \( \text{SVC}(b)_{t_2} < \text{SVC}(a)_{t_2} \).
Algorithm 1 SVC calculation

Require: \( N \) (\( C^3G \) nodes), \( E \) (\( C^3G \) edges), \( T \) (set of threads)

1. for each \( n \in N \) do
2. \( SVC^0(n) \leftarrow \emptyset_0, \ldots, \emptyset_{|T|} \)
3. for each \( n_1 \rightarrow n_2 \in E \) do
4. \( t \leftarrow n_1.t \)
5. \( SVC^0(n_2.t) \leftarrow SVC^0(n_2.t) \cup n_1 \)
6. \( i \leftarrow 0 \)
7. repeat
8. \( i \leftarrow i + 1 \)
9. \( SVC^i(n) \leftarrow SVC^{i-1}(n) \)
10. \( HBM \leftarrow \) an empty dictionary from nodes to methods
11. for each \( n \in O \) do
12. \( SVC^i(n) \leftarrow SVC^i(n) \)
13. if \( n = m.i \land n.t \) is FIFO then
14. for each \( m' \in HBM[m]|_t \) do
15. \( SVC^i(m'.s, t) \leftarrow SVC^i(m'.s, t) \cup m'.e \)
16. Minimize \( SVC^i(m'.s, t) \)
17. \( HBM[n]|_t \leftarrow m \)
18. for each \( n' \in Ino(SVC^i(n)|_t) \) do
19. \( HBM[n'] \leftarrow HBM[n'] \cup HBM[n] \)
20. until \( SVC^i = SVC^{i-1} \)
21. \( SVC \leftarrow SVC^i \)

Figure 5: The \( C^3G \) of the illustrative app in Figure 1a annotated with Static Vector Clock values.

The proof of correctness for these theorems can be found on the companion website [1]. We can determine if there is a happens-before relation between two nodes using Theorem 1. For example, \( 7 < 21 \) because there exists a sequence of nodes, \( (7, 8, 18, 21) \), where each element belongs to the \( SVC \) of its next element, or \( 7 \in SVC(8), 8 \in SVC(18) \) and \( 18 \in SVC(21) \). Using Theorem 2 we can determine if there is no happens-before relation between two nodes. For example, there is no happens-before relation between nodes 20 and 21, since we have \( SVC(20) = 5 < SVC(21) = 10 \) and \( SVC(21) = 10 < SVC(20) = 13 \).

Algorithm 1 shows how \( SVC \) is computed. We first initialize the \( SVC \) using \( C^3G \) edges as shown on lines 1 – 5 (\( n.t \) indicates the thread that node \( n \) is running on). This initialization satisfies the first two principles: call-graph edges enforce a method’s invocation state happens before its start state (principle 1), while CSF edges enforce the happens-before relations implied by the control-flow graph (principle 2). The rest of the algorithm satisfies the third principle using a fixed-point iteration method, i.e., update \( SVCs \) according to Same Task-Queue Order until the values of \( SVCs \) do not change.

Same Task-Queue Order ensures if two methods, \( m_1 \) and \( m_2 \), run on the same FIFO thread and \( m_1.i \) happens before \( m_2.i \) then \( m_1 \) finishes before \( m_2 \) (or \( m_1.e < m_2.e \)). For example, because the invocation states of methods \( onClick \) and \( initTime \) (nodes 3 and 4) happen before the invocation of \( onPost-1 \) (node 14), we have \( \{8, 12\} \subseteq SVC(19) \) (the end states of \( onClick \) and \( initTime \) happen before the start state of \( onPost-1 \)). Note that even though \( initTime \) and \( onClick \) happen before \( onPost-1 \), there is no happens-before relation between their end states and \( onPost-1 \) invocation state.

A naive approach to apply Same Task-Queue Order is to query happens-before relation between all pairs of method invocation states and update \( SVCs \) accordingly. However, we can avoid the query cost by traversing an ordered DAG \( (N, Ino(SVC^i)) \), where the nodes are \( N \) and the directed edges are the set of \( (n_1, n_2) \) such that \( n_1 \in SVC(n_2) \). One property of this ordered DAG, denoted by \( O \) for short, is that if \( n_1 < n_2 \) then \( n_1 \) precedes \( n_2 \) in \( O \). For example, the node identifiers in Figure 5 show the position of nodes in \( O \) and there is no \( n_1 < n_2 \) where the id of \( n_1 \) is greater than the id of \( n_2 \). We store the set of methods where their invocations occur before node \( n \) in \( HBM[n] \) while iterating over \( O \), since all the nodes that happen before \( n \) are already visited; as a result, there is no need to query happens-before for each pair of the invocation nodes.

Algorithm 1 initializes \( O \) and \( HBM \) on lines 10–11, then iterates over \( O \). For each \( n \) running in a FIFO thread that is the invocation state of method \( m \), the algorithm appends the end node of methods in \( HBM[n]|_t \) to the \( SVC \) of \( m \)’s start node (lines 13 – 15). Since \( SVC^i(m'.s, t) \) may be updated, it should be minimized (line 16). To increase the performance of the algorithm, \( HBM[n]|_t \) is set to \( m \), since \( n \) is the latest invoked node in thread \( t \) for the remaining nodes (line 17). Finally, the algorithm propagates \( HBM[n]|_t \) to the nodes in \( Ino(SVC^i(n)|_t) \) which is the set of nodes like \( n' \) that \( n \in SVC^i(n'|_t) \) (lines 18 – 19). This process repeats until \( SVC^i \) does not change.

3.5 Thread-Sensitive Filter

In this component, we use \( C^3G \) and \( SVC \) to prune the false event races produced by Event-Race Candidate Detection component (recall Figure 2). Given an event race, we check if there is a happens-before relation between the corresponding
methods. Method $m_1$ happens before method $m_2$ if all end states of $m_1$ happen before start states of $m_2$. We remove all event races in which there is a happens-before relation between their method calls. For example, we filter out the event race $(\text{initTime}:22, \text{onPostExecute}:35, \text{elapsedTime})$, because as depicted in Figure 3, there is a happens-before relation between all end states of $\text{initTime}$ and start states of $\text{onPost}$: $\text{initTime}.e < \text{onPost}-0.s$ and $\text{initTime}.e < \text{onPost}-1.s$. However, for the event race $(\text{onPostExecute}:35, \text{onPostExecute}:35, \text{elapsedTime})$, there is neither a happens-before relation between $\text{onPost}-0.s$ and $\text{onPost}-1.e$, nor between $\text{onPost}-0.e$ and $\text{onPost}-1.s$. We thus report it as a possible event race.

To further prune the false event races, two filters (If-Guard and Null-At-End) are designed for Use-after-Free (UF) defects. A UF defect is a harmful event race where a memory location free (writes null) and another access uses (reads) it, resulting in a NullPointerException to be thrown. If-Guard removes the UF defects where the read access is guarded by a null-checking condition, e.g., $\text{if}(f!=\text{null})f.use();$ the field $f$ will not be accessed if it is null. In addition, Null-At-End filters UF defects where the memory location is not reassigned with a value other than null after becoming free e.g., in $f = \text{null}; f = \text{new F}();$ the field $f$ is not null.

In addition to the above-mentioned filters that are sound, we also provide several heuristics that are unsound, but in practice can significantly reduce the false warnings, as described next.

### 3.6 Prioritization

The reported event races by the previous component include all possible event races that need to be reviewed by developers. To facilitate this manual process, we prioritize the detected event races by our confidence in their existence. Our confidence is inversely related to the degree of over-approximation in the static analysis, i.e., less over-approximation in detecting an event race leads to more confidence about its existence. Due to the over-approximation of static analysis and our conservative approach, we do not filter event races for which we have incomplete information.

We prioritize event races according to the number of satisfied over-approximation properties. (Reachability) We prioritize event races that are reachable, i.e., there are paths in the call graph from the entry-point to both methods of a reachable event race. The thread-sensitive filters are not applied on unreachable event races because their thread information is unknown. (Must-Alias) Event races that involve statements accessing must-alias fields have priority over may-alias fields. Two fields are must-alias if they always point to the same memory location. (Known-Thread) Event races where the threads of their methods are known have more priority than others. For example, when we are not certain about the corresponding Looper of a Handler (which can be the main thread Looper or a custom thread Looper), we assume its messages are handled in an unknown thread that dispatches messages in parallel.

### 4 EVALUATION

This section discusses our experimental evaluation to answer the following research questions:

- **RQ1** How accurate is ER Catcher in detecting true event races?
- **RQ2** How fast does ER Catcher analyze real-world apps?
- **RQ3** How effective is ER Catcher in filtering false event races?
- **RQ4** What is the impact of modeling concurrency in improving the overall accuracy of static analysis?

We evaluate ER Catcher using three different datasets. First, we use a set of benchmark apps containing event-race defects, called BenchERoid [39]. For these apps the ground truth is known, allowing us to report the precision, recall, and F1 score of ER Catcher. Second, we use a "Curated" dataset of 31 real-world apps with event-race defects that have been confirmed, through either dynamic analysis, or code commit messages. We collected these apps by reviewing the prior literature [17, 21, 29] and crawling the open-source repositories. Finally, we evaluate the scalability and effectiveness of ER Catcher using 500 randomly selected apps from F-Droid [3]. Table 1 summarizes the datasets used for evaluation.

We compare ER Catcher with nAdroid [17], the state-of-the-art static Use-after-Free (UF) event-race detector that is available publicly, but not entirely open-source. Since nAdroid detects only UF event races, to make the comparison fair between ER Catcher and nAdroid, we configure ER Catcher to report only UF event races. We also tried to empirically compare ER Catcher to several other tools, namely SIERRA [22], SARD [45], ASYNCCLOCK [20], and EventRacer [10]. Since none of these tools are available, we contacted the corresponding authors. Unfortunately, despite multiple attempts, the authors either did not respond to us or confirmed their inability to release their tool. We provide a qualitative discussion of the differences between ER Catcher and these other techniques in Section 5.

### Table 1: Properties of datasets used in our experiments.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Apps</th>
<th>#Methods</th>
<th>Criteria</th>
<th>Average</th>
<th>Median</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>BenchERoid</td>
<td>34</td>
<td>27</td>
<td></td>
<td>1123</td>
<td>1</td>
<td>18</td>
<td>51</td>
</tr>
<tr>
<td>Curated</td>
<td>31</td>
<td>19</td>
<td></td>
<td>3850</td>
<td>9</td>
<td>1</td>
<td>138</td>
</tr>
<tr>
<td>F-Droid</td>
<td>500</td>
<td>1041</td>
<td></td>
<td>4171</td>
<td>5</td>
<td>10</td>
<td>976</td>
</tr>
</tbody>
</table>

### 4.1 RQ1: Accuracy

We ran ER Catcher and nAdroid on the 34 apps provided by BenchERoid to measure their respective accuracy. The results are shown in Table 2. The actual number of event races in the benchmark apps are shown in column 2. nAdroid uses two different filters (sound and unsound) for reducing false event races; we report both of them in columns 3 and 4, respectively. We report the event races identified by ER Catcher in column 5 and the prioritized event races (recall Section 3.6) in column 6. We categorize the apps provided by...
Table 2: Benchmark Result. The general event races (highlighted rows) were not considered for the accuracy metrics.

<table>
<thead>
<tr>
<th>Event Races</th>
<th>nAdroid</th>
<th>ER Catcher</th>
<th>ER Catcher Prioritized</th>
</tr>
</thead>
<tbody>
<tr>
<td>SingleActivity7</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>SingleActivity8</td>
<td>2</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Service5</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>AsyncTask5</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>AsyncTask6</td>
<td>0</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Flow-Sensitive UF

<table>
<thead>
<tr>
<th>Event Races</th>
<th>nAdroid</th>
<th>ER Catcher</th>
<th>ER Catcher Prioritized</th>
</tr>
</thead>
<tbody>
<tr>
<td>SingleActivity2</td>
<td>0</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>SingleActivity5</td>
<td>2</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>SingleActivity6</td>
<td>2</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>AsyncTask2</td>
<td>0</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Looper1</td>
<td>0</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Thread-Sensitive UF

<table>
<thead>
<tr>
<th>Event Races</th>
<th>nAdroid</th>
<th>ER Catcher</th>
<th>ER Catcher Prioritized</th>
</tr>
</thead>
<tbody>
<tr>
<td>SingleActivity3</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>AsyncTask1</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>AsyncTask3</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Thread1</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Thread2</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Looper2</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Service2</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Service4</td>
<td>0</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Other UF

<table>
<thead>
<tr>
<th>Event Races</th>
<th>nAdroid</th>
<th>ER Catcher</th>
<th>ER Catcher Prioritized</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiComp1</td>
<td>2</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Receiver</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Service1</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>LifeCycle1</td>
<td>3</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>LifeCycle2</td>
<td>0</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>LifeCycle3</td>
<td>2</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>LifeCycle4</td>
<td>2</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>SingleActivity1</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>SingleActivity4</td>
<td>0</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>AsyncTask4</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Executor1</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Executor2</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>TimerTask1</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>TimerTask2</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Looper3</td>
<td>1</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Overall, ER Catcher is significantly more accurate than nAdroid. ER Catcher achieves 80% precision and 93% recall, compared to nAdroid’s 67% precision and 87% recall. Due to its flow-sensitivity, ER Catcher is able to filter all of the false positives reported by nAdroid under the Flow-Sensitive UF category. Due to its thread-sensitivity, ER Catcher is able to identify additional true event races compared to nAdroid under the Thread-Sensitive UF category.

Although ER Catcher is more accurate than nAdroid, it has its own limitations. ER Catcher is path-insensitive, therefore it reports false positive UF event races that reside on execution paths that will not execute, e.g., Service1. Due to time-insensitivity, ER Catcher reports false positives in cases where the statements execute at specific times, e.g., SingleActivity4. ER Catcher fails to filter one false-positive event race in SingleActivity1 due to incomplete reachability information; however, it is removed in "ER Catcher Prioritized". The imprecision in the off-the-shelf points-to analysis used by ER Catcher leads to a false negative in Service2.

We also ran ER Catcher and nAdroid on the Curated dataset, consisting of 31 real-world apps with confirmed event races. ER Catcher was able to analyze all of the 31 apps, while nAdroid could only analyze 27 of them. ER Catcher achieved 100% recall, detecting all of the event races in these apps, while nAdroid achieved 88% recall. Since not all of the event races in these apps are known, we are unable to report the recall using this dataset.

4.2 RQ2: Scalability

We analyzed all three datasets using ER Catcher and nAdroid to compare the scalability of these techniques. Table 3 summarizes the results. For the first two datasets (i.e., BenchERoid and Curated datasets), we did not set a timeout. However, due to the large number of apps in the third dataset (F-Droid), we set a timeout of 5 minutes. On average, ER Catcher analyzed each app in the Curated dataset within 231 seconds, while nAdroid required 3,134 seconds (∼52 minutes). Overall, ER Catcher finished the analysis between 12 to 13 times faster than nAdroid for the first two datasets.

For the F-Droid dataset, ER Catcher analyzed 459 out of 500 apps (more than 90%) within the designated time of 5 minutes (two apps could not be analyzed because FlowDroid could not generate the call graph for them). However, nAdroid could only analyze 30 apps (6%) within the allotted time. Furthermore, nAdroid crashed during analysis of 316 apps (more than 60%) indicating nAdroid is unable to complete the analysis irrespective of time.

Table 3: Analysis time summary

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Analyzed Apps</th>
<th>Average</th>
<th>Median</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>BenchERoid</td>
<td>nAdroid</td>
<td>29</td>
<td>217/4s</td>
<td>154/4s</td>
<td>133/4s</td>
</tr>
<tr>
<td></td>
<td>ER Catcher</td>
<td>29</td>
<td>18/4s</td>
<td>7/4s</td>
<td>18/4s</td>
</tr>
<tr>
<td>Curated dataset</td>
<td>nAdroid</td>
<td>27</td>
<td>313/4s</td>
<td>786/4s</td>
<td>83/4s</td>
</tr>
<tr>
<td></td>
<td>ER Catcher</td>
<td>31</td>
<td>231/4s</td>
<td>29/4s</td>
<td>4/4s</td>
</tr>
<tr>
<td></td>
<td>nAdroid</td>
<td>0.87x</td>
<td>13x</td>
<td>27x</td>
<td>21x</td>
</tr>
<tr>
<td></td>
<td>ER Catcher</td>
<td>0.87x</td>
<td>13x</td>
<td>27x</td>
<td>21x</td>
</tr>
<tr>
<td>F-Droid dataset</td>
<td>nAdroid</td>
<td>30</td>
<td>153/4s</td>
<td>152/4s</td>
<td>108/4s</td>
</tr>
<tr>
<td></td>
<td>ER Catcher</td>
<td>459</td>
<td>53/4s</td>
<td>29/4s</td>
<td>4/4s</td>
</tr>
<tr>
<td></td>
<td>nAdroid</td>
<td>0.06x</td>
<td>3x</td>
<td>5x</td>
<td>27x</td>
</tr>
</tbody>
</table>

4.3 RQ3: Effectiveness of Filters

To evaluate the degree to which ER Catcher filters out false event races in real-world apps, the number of UF event races in three stages of analysis are reported in Table 4. The first row shows the number of event-race candidates identified by the “Event-Race Candidate Analysis” component (recall Section 3.2). The second
row shows the number of filtered UF event races reported by the "Thread-Sensitive Filter" component (recall Section 3.5). The fourth row reports the number of prioritized UF event races (recall Section 3.6). The reduction rates achieved for the Curated and F-Droid datasets are %77 and %86, respectively. This results in prioritizing 37 and 23 UF event races that require manual investigation by developers. In practice, many of the detected event races are caused by the same defect in code, i.e., one defect in code causes event race conditions under multiple execution contexts. As a result, developers can often confirm the presence of a defect without having to review the complete list of reported event races.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Average</th>
<th>Median</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>#UF Candidates</td>
<td>317</td>
<td>341</td>
<td>0</td>
<td>1238</td>
</tr>
<tr>
<td>#Filtered UFs</td>
<td>150</td>
<td>114</td>
<td>0</td>
<td>540</td>
</tr>
<tr>
<td>% Candidates</td>
<td>50%</td>
<td>54%</td>
<td>0%</td>
<td>92%</td>
</tr>
<tr>
<td>#Prioritized UFs</td>
<td>37</td>
<td>15</td>
<td>0</td>
<td>147</td>
</tr>
<tr>
<td>% Prioritized Candidates</td>
<td>23%</td>
<td>12%</td>
<td>0%</td>
<td>100%</td>
</tr>
</tbody>
</table>

### 4.4 RQ4: Impact of Modeling Concurrency

A byproduct of modeling the concurrent behavior of Android apps and libraries in the form of CIC is that ER Catcher can discover a number of additional methods that are invoked indirectly, i.e., through the library callbacks. This enables ER Catcher to compute reachability of methods in ways that are more accurate than other state-of-the-art techniques, such as FlowDroid [7]. To evaluate this facet of our work, we ran both ER Catcher and FlowDroid on the F-Droid dataset. We identified additional reachable methods in 169 of these apps. Compared to FlowDroid, ER Catcher was able to detect on average 8 more reachable methods per app, and up to 78 more reachable methods in one app.

This is notable given the extensive number of tools (e.g., [8, 9, 23, 48]) that rely on FlowDroid for analysis of Android apps, particularly for security assessment. Consider, for instance, an information leakage vulnerability caused by flow of data from a private source, e.g., camera, to a sink, e.g., network. If the sink is located in a method that is determined to be unreachable by the analysis, the vulnerability cannot be discovered. According to our results, numerous Android security analysis tools [8, 9, 48] built on top of FlowDroid [7] would fail to discover vulnerabilities or malicious behaviors that reside in such locations in code.

### 5 RELATED WORK

Our work builds upon three major threads of research: concurrency analysis, event-race detection, and summary-based static analysis in Android. We provide a brief overview of the relevant research for each of these threads in this section.

**Analysis of Concurrent Behaviors**: Researchers have investigated concurrent behaviour of traditional programming languages such as C or Java, primarily focusing on data races [27, 33, 36], deadlocks [11, 14, 31], happens-before relation [52], sequential or non-sequential concurrency errors [35, 49, 50] and testing [42]. None of these, however, has investigated the concurrent behavior of Android apps. The most relevant work is that of Zhou et al. [52] that introduces the concept of Static Vector Clock to analyze may-happen-in-parallel relation between instructions in C/C++; however, their realization of this concept is not applicable to event-based systems like Android. ER Catcher addresses this limitation by introducing event-based properties, e.g., Same Task-Queue Order.

Researchers have also analyzed the concurrent behavior of Android apps, mostly focusing on dynamic analyses such as test generation [41], manipulating the code execution [16, 44], and detecting happens-before relations [28]. None of these techniques is able to detect event races in Android.

**Event Race Detection in Android**: Although Android has traditional Java thread constructs, tools developed for Java cannot readily detect event races in Android since existing tools are unaware of relations between events. To meet this gap, researchers have devised dynamic and static event-race detection approaches for Android. Dynamic event race detection approaches gather execution traces of Android apps either manually [21] or automatically [10, 20, 29] using an app crawler such as Monkey [5]. Then the execution traces are analyzed off-line to detect happens-before relations between events either by graph analysis [10, 21, 29], or leveraging vector clock [20]. These techniques fail to identify event races due to their limited coverage of the behaviors of apps [22].

Static event-race detection techniques address the problem of missing true defects by analyzing the whole program. DeVA [38] detects "event anomalies" where two events access the same memory location, and one of the accesses is a write. Since DeVA does not consider happens-before relations, it produces a large number of false positives. SIERRA [22] considers happens-before relation and has limited context-sensitivity (capturing only one asynchronous action as the context), but it is not aware of other threads except for the main thread which may result in failure to identify bugs; we addressed this issue by our thread-sensitive analysis modeled in CIC. nAdroid [17] uses an existing traditional race detection technique to identify Use-after-Free (UF) bugs. Due to the imprecise threadification model of nAdroid it reports a large number of false positives; we extensively explained and empirically evaluated the limitations of nAdroid in comparison to ER Catcher earlier. SARD [45], similar to nAdroid, detects UF bugs, using a flow- and context-sensitive model of Looper. SARD applies an exhaustive context creation strategy making it unscaleable. We address the scalability issue by using summary-based analysis techniques to reduce the space complexity.

In all of the mentioned techniques, the support for various Android concurrency libraries is hardcoded in the implementation of tools, making it difficult to revise the tools to support new or modified Android libraries. ER Catcher addresses this limitation by separating the implementation of tool from the library CSFs representing the concurrent behavior of Android libraries. Provided with
new or modified library CSFs, ER Catcher can be readily extended to support new or modified Android libraries without requiring changes to its implementation. Summary-Based Static Analysis in Android: Another research thrust has investigated summary-based approaches for precise and fast analysis [4, 6, 7, 12, 13, 47]. These techniques mostly use Inter-procedural Distributed Environment (IDE) framework [37] for modelling the inter-procedural data-flow of the code; however, to the best of our knowledge, no prior work has leveraged IDE frameworks to model concurrent behavior of code like happens-before relations. Moreover, none of these techniques capture inter-thread communication precisely. As a future work, it would be interesting to see if IDE frameworks, or non-distributive summary based analyses such as [32], can improve the scalability of creating the $CG$ model. ER Catcher takes a step towards addressing this issue by introducing ways of summarizing concurrency behavior in Android.

6 CONCLUSION
Concurrency-induced defects, such as race, are one of the most frequently encountered types of defect in Android apps [51]. We presented ER Catcher, a fast, novel and accurate static analysis framework for event-race detection in Android. Experiments using benchmark apps show that ER Catcher is accurate, capable of detecting event races with 80% precision and 93% recall. Compared to the only other publicly available tool for event-race detection in Android, ER Catcher is substantially faster (by a factor of 12) and more accurate (11% higher F1-measure). Results further corroborate its effectiveness in detecting all of the event races confirmed to exist in a set of 31 real-world apps.

In our future work, we plan to expand the applications of ER Catcher to other concurrency-related analyses. We believe the efficient happens-before analysis of ER Catcher can be used as a plugin in Android Studio to provide real-time feedback to developers during the development. Furthermore, since concurrency is the main cause of flaky tests [43], we aim to study the application of ER Catcher in the detection of such tests. ER Catcher and research artifacts are publicly available for download from the companion website [2].

ACKNOWLEDGMENT
This work was supported in part by award numbers 1618132 and 1823262 from the National Science Foundation. We would like to thank the anonymous reviewers of this paper for their detailed feedback, which helped us improve the work.

REFERENCES
event-place: Williamsburg, VA, USA.

334


