Science of Computer Programming 163 (2018) 1–18

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

AATT+: Effectively manifesting concurrency bugs in Android apps

Jue Wang[a,b], Yanyan Jiang[a,b], Chang Xu[a,b,*], Qiwei Li[a,b], Tianxiao Gu[a,b], Jun Ma[a,b], Xiaoxing Ma[a,b], Jian Lu[a,b]

[a] *State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China*
[b] *Department of Computer Science and Technology, Nanjing University, Nanjing, China*

ARTICLE INFO

Article history:
Received 4 August 2017
Received in revised form 8 March 2018
Accepted 11 March 2018
Available online 30 March 2018

Keywords:
Android
Concurrency

ABSTRACT

Smartphones are indispensable in people's daily activities, and smartphone apps tend to be increasingly concurrent due to the wide use of multi-core devices and technologies. Due to this tendency, developers are increasingly unable to tackle the complexity of concurrent apps and to avoid subtle concurrency bugs. To better address this issue, we propose a novel approach to detecting concurrency bugs in Android apps based on the fact that one can generate simultaneous input events and their schedules for an app, which would easily trigger concurrency bugs in an app. We conduct systematic state space exploration to find potentially conflicting resource accesses in an Android app. The app is then automatically pressure-tested by guided event and schedule generation. We implemented our prototype tool named AATT+ and evaluated it with two sets of real-world Android apps. Benchmarking using 15 Android apps with previously known concurrency bugs, AATT+ and existing concurrency-unaware techniques detected 10 and 1 bugs, respectively. Evaluated with another set of 17 popular Android apps, AATT+ detected 11 concurrency bugs and 7 of them were previously unknown, achieving an over 80% higher detection rate than existing concurrency-unaware techniques.

1. Introduction

Smartphone and its various apps (i.e., applications) gain popularity rapidly in recent years. By the end of September, 2017, there had been more than 3.3 million apps in the Google play store [1].

For smooth user experience, smartphone apps should quickly respond to incoming events as well as processing time-consuming tasks at background [2]. Therefore, concurrency plays an important role in smartphone apps, even though it is known to be notoriously difficult to write, test, and debug concurrent programs. Taking Android, one of the most popular smartphone platforms, as an example, although it has a set of constraints and mechanisms to make concurrent programming simpler (e.g., Android UI updates are constrained in the main thread and time-consuming tasks are forced to background), developers are still unable to always correctly understand an app's concurrent behaviors as the app becomes increasingly complicated, and thus leave subtle concurrency bugs in its releases.

* Corresponding author at: Department of Computer Science and Technology, Nanjing University, Nanjing, China.
E-mail addresses: juewang591@gmail.com (J. Wang), jiangyy@outlook.com (Y. Jiang), changxu@nju.edu.cn (C. Xu), liqiwei1992@gmail.com (Q. Li), tianxiao.gu@gmail.com (T. Gu), majun@nju.edu.cn (J. Ma), xxm@nju.edu.cn (X. Ma), lj@nju.edu.cn (J. Lu).

https://doi.org/10.1016/j.scico.2018.03.008
0167-6423/© 2018 Elsevier B.V. All rights reserved.

To catch concurrency bugs early in the development, existing work [3–5] finds non-commutative events as "data races" to be indicators of concurrency bugs. However, finding such races requires high-quality inputs that trigger racing events to be manifested in the execution, at the same time being close enough in time. Furthermore, races may not always lead to concurrency bugs and thus filtering out false-positive reports is still an open research problem [3].

Therefore in this work, we focus on taking a different approach to detecting hidden concurrency bugs in an Android app by manifesting them during the execution of an app. We observed that in Android app testing, one can generate both simultaneous events and their schedules, and manifest potential concurrency bugs with such event-schedule combinations. The main advantage of this approach is that it produces only true positives. To realize this idea, there are two challenges: (1) how to determine which events are potentially related to concurrency bugs, and (2) how to systematically generate events and their schedules to manifest the concerned concurrency bugs.

To address the first challenge, we adapt the app state space exploration mechanism from our previous work GreenDroid [6], which has a systematic state exploration engine, to conduct dynamic analysis for an Android app, in order to find each concurrent task (of classes Listener, Thread, AsyncTask, etc.)'s shared resource access sites (i.e., *access points*, or APs). Following the definition of non-commutative race [5], two concurrent tasks are *conflicting* if they can access a particular AP at the same time and at least one access is a write operation. Conflicting tasks potentially relate to concurrency bugs, and we need to right schedule them to trigger such bugs by a guided event-schedule combination generator.

To address the second challenge, we propose a scheduling oriented depth-first search (SO-DFS) algorithm, which integrates both event generation and schedule generation. SO-DFS is based on existing work on state space exploration [7], which traverses each transition between distinct app states (defined by an app's GUI layout) exactly once. We extend it by systematically examining all *k*-combination schedules of concurrency-bug related events and conflicting tasks available at the current app state as the exploration goes.

We implemented our approach as a prototype tool named AATT+. Our tool targets Android apps since Android is one of the most popular smartphone platforms. We evaluated the effectiveness and efficiency of our tool using two sets of real-world Android apps. For the 15 apps with known concurrency bugs from GitHub and Google Code, AATT+ successfully detected 10 of the known bugs. For another set of 17 randomly selected apps, AATT+ detected 11 concurrency bugs, 7 of which were previously unknown. Detailed evaluation results show that AATT+ achieved an over 80% higher bug detection rate than existing concurrency-unaware techniques (e.g., DFS and Monkey/random testing), with reasonable overhead and without any false positive.

Summarizing all concurrency bugs studied in our evaluation, we observed a few Android concurrency bug patterns. Echoing a previous empirical study [8], we found that all the concurrency bugs were caused by either *atomicity violation* or *order violation*. Among these bugs, 10 were caused by Android life-cycle events, which can easily be out of a developer's consideration. Other common Android-specific concurrency bug causes include incorrectly assumed atomicity (e.g., a developer can consider that events of the same type can be triggered at most once a time, and this is equivalent to incorrectly assuming the atomicity of an event and its asynchronous tasks), and improper use of Android-provided concurrency mechanisms (e.g., a developer can use an Android-provided asynchronous task model to process a shared resource, but does not access the resource with the post-process part of the task model, which guarantees to execute when the resource processing finishes, and this can break the assumptions about execution orders and lead to concurrency bugs). We hope that our findings can make developers aware of such bug patterns and help avoid them in Android app development.

We summarize our contributions in this article as follows:

- We proposed an effective approach to detecting concurrency bugs in Android apps based on the interplay of both event and schedule generation, which produces only true positives.
- We implemented our prototype tool named AATT+ and evaluated it using real-world Android apps. AATT+ detected previously unknown concurrency bugs and our quantitative analysis shows that our approach is both effective and efficient.
- We studied all detected concurrency bugs from our experimental subjects and identified common bug patterns, which can benefit both Android developers and researchers.

The work presented in this article is based on our previous work AATT [9], and has significantly extended it. Previously, we used a static-dynamic hybrid analysis to determine events potentially relating to concurrency bugs. The static analysis constructs partial call graphs of an Android app and identifies APs in the graphs, while the dynamic analysis examines these APs in a depth-first fashion. However, this approach can miss some APs due to imprecise call graphs from the static analysis and the poor state space exploration ability of the dynamic analysis. In this work, we have extended our approach with a GreenDroid-enhanced dynamic analysis [6], which can systematically explore an Android app's state space to identify APs by dynamic analysis. Moreover, in the previous work, we did not concern component life-cycle events of an Android app during the guided event-schedule combination generation, and this weakens AATT's ability of manifesting concurrency bugs. In this work, we have extended our approach by adopting heuristic mechanisms to address this issue. Our evaluation demonstrates the effectiveness of our extensions. Detailed evaluation results show that AATT+ successfully detected 21 concurrency bugs, 47.6% more than what AATT detected.

The rest of this article is organized as follows. Section 2 introduces some background knowledge and presents a motivating example. Sections 3 elaborates on our concurrency bug manifesting approach. Section 4 introduces the implementation of our prototype tool, AATT+. Section 5 experimentally evaluates our AATT+ and analyzes the experimental results. Section 6 presents the lessons we learned and common bug patterns we identified from experimental results. Section 7 reviews the related work in recent years, and finally Section 8 concludes this article.

2. Background and motivation

In this section, we introduce some Android background knowledge, and present a motivating example for concurrency bug detection.

2.1. Components and event handling of Android apps

Android is one of the most popular smartphone platforms. It provides a common application model for all apps running on it, which are typically written in Java and compiled to Dalvik bytecode. The application model mainly contains four types of components [10]: (1) an *Activity* contains a graphical user interface (GUI) and is responsible for interacting with users, (2) a *Broadcast Receiver* is responsible for receiving system-wide messages and responding to them accordingly, (3) a *Service* performs time-consuming tasks, and (4) a *Content Provider* is responsible for managing shared data. These major components can also comprise many other sub-components for more complicated program logics, such as *Fragment* and *AsyncTask*.

During the execution, an Android app takes sequences of various types of events as input. An *event* can either be issued by a user (e.g., clicking or swiping) or by the Android Runtime system (e.g., connecting to wifi networks). Each app component consists of event handlers invoked by the Android system for handling certain types of events. An Android app can thus be regarded as a set of loosely-coupled handlers.

In order to handle input events, an Android app can post tasks on different threads. A *task* is the unit of execution in an Android app. It can be a method of an app component such as a `Listener`, a native `Thread`, an `AsyncTask`, etc.

For a set of input events, sending events to an app in different orders forms different *event sequence*s, i.e., the *schedule*s of events. For each event sequence, the tasks posted to handle them can be posted and executed in different orders. Such orders are *schedule*s of these tasks.

2.2. Android concurrency model

The Android concurrency model provides its own constraints and mechanisms to help developers better utilize multi-tasking and avoid errors due to non-determinism. First, all GUI and system events are handled in the main thread to avoid data races on shared resources. Second, GUI updates are serialized in the main thread in order to eliminate GUI update races. Finally, time-consuming tasks (e.g., network accesses) must not run in the main thread and can only be carried out by asynchronous tasks, which trigger events at completion, such that the main thread can quickly respond to user and system events. Developers usually use built-in asynchronous tasks to manage concurrency, including `AsyncTask` and `Loader` [11].

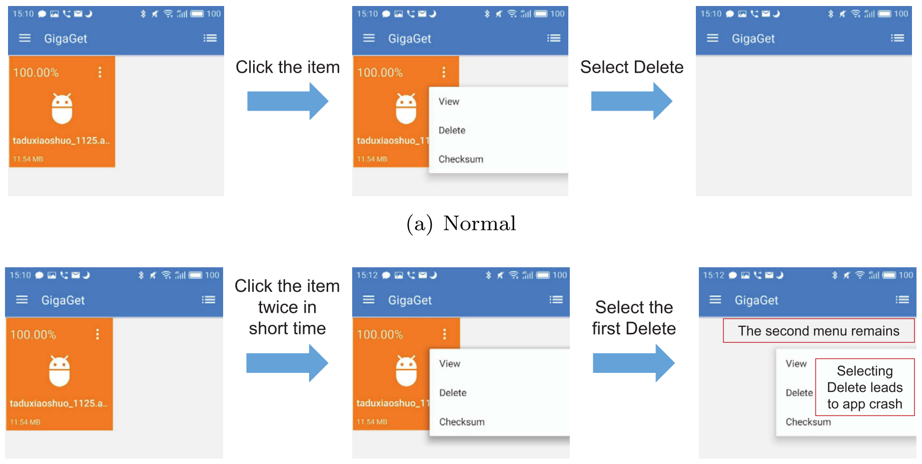
Moreover, tasks can post new tasks to different threads during the execution. Therefore, such situation can occur where a task running at background posts a task to the main thread or the other way around. This concurrent mechanism brings convenience to developers. However, it also complicates the concurrent execution of Android apps and could result in many concurrency bugs.

2.3. Motivating example

The Android concurrency model helps improve apps' performance. However, as apps are becoming increasingly complicated, more and more developers tend to create complicated cascading tasks that have non-deterministic outcomes [4,5], as well as to mix native threads (e.g., `Thread` and `Threadpool` objects) in the execution. Such complication can easily lead to concurrency bugs. Moreover, the Android platform is a complicated system and many developers could misunderstand many of its concurrency mechanisms, and this also brings subtle concurrency bugs, which are difficult to detect with a limited testing budget.

Fig. 1 gives an example of concurrency bug in GigaGet, a lightweight multi-threaded file downloader, while Fig. 2 presents the simplified code snippet. The code seems to work at a first glance, as Fig. 1(a) presents: the `item` represents a downloaded file (the orange square on the screen). When the `item` object is clicked, its associated `OnClickListener` object's `onClick` method (Lines 3–21) is invoked. Then a `PopupMenu` object (`popup`) is created (Lines 4–5) and its `MenuItem` object (`del`, which represents the **Delete** menu item on the screen) is set visible (Lines 6–8). When creating the `popup` menu, the Android system will disable the `item` object's associated `OnClickListener` object. Then an `OnMenuItem-ClickListener` object is associated with the `popup` menu (Lines 10–19). If the `del` menu item is selected, the `item` object and the file it represents will be deleted by invoking the corresponding `deleteItem` method (Lines 13–16).

However, as Fig. 1(b) shows, if the app's user clicks the `item` object twice quickly before the `popup` menu is created, two identical `popup` menus will be created and both of them are functional. Selecting the `del` menu item on both `popup` menus would lead to double invocation of the `deleteItem` method (Lines 13–16) and delete the `item` object and the file it represents twice. This abnormal practice then crashes the app.



(a) Normal

(b) Abnormal

Fig. 1. Work flow in the motivating example to trigger the concurrency bug. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

```java
item.setOnClockListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
      PopupMenu popup = new PopupMenu(context, item);
      popup.inflate(R.menu.mission);
      Menu menu = popup.getMenu();
      MenuItem del = menu.findItem(R.id.del);
      del.setVisible(true);

      popup.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListener() {
         @Override
         public boolean onMenuItemClick(MenuItem item) {
            if (item.getItemId() == R.id.del) {
               manager.deleteItem(downloadItem.pos);
               return true;
            }
            return false;
         }
      });
      popup.show();
   }
});
```

Fig. 2. Motivating example from app GigaGet's code.

This example demonstrates a subtle atomicity violation bug due to an event schedule out of developers' consideration. GigaGet developers incorrectly assumed that if the `item` object is clicked, the follow-up action sequence (i.e., disabling the click event on the `item` object and creating the `popup` menu) will be carried out atomically. Such assumption can be easily made for sequential code. However, a concurrent system allows inserting another click event, and this breaks the assumption.

As the example shows, both a specific *event sequence* and a specific event-task *schedule* are required to manifest this concurrency bug. The required event sequence is two click events on the `item` object and one click event on each `del` menu item, and the specific schedule is that the `onClick` task, which handles the second click event, must be posted and executed before the popup menu's `show` task is posted and executed. Unfortunately, existing predictive trace analysis techniques [3–5] have difficulties in detecting this bug, unless the input event sequences for testing already contain consecutive *file-deletion* events. Moreover, even if these techniques do detect the bug with such events, they can also produce many false positives, which is difficult to filter out. Random testing (e.g., Monkey), on the other hand, may have a chance to detect this bug. However, it has a very low probability to generate such an appropriate event sequence and has no bug manifestation guarantee.

This motivates us to design our approach to proactively detecting concurrency bugs in Android apps. As demonstrated, while existing work focuses on either event generation [12] or schedule generation [13], it is insufficient to expose many hidden concurrency bugs. We aim to leverage the interplay of both event and schedule generation at runtime to automatically generate combinations of conflicting events and suspicious schedules, so as to manifest concurrency bugs in Android apps. Such approach guarantees to detect the concurrency bug in our motivating example and produces only true positives.

3. Effectively manifesting concurrency bugs

In this section we elaborate on our approach to detecting concurrency bugs in Android apps. Notations and definitions are presented in Section 3.1, an approach overview is presented in Section 3.2, technical details are presented in Sections 3.3 and 3.4, and a discussion about the improvement from our previous work AATT is presented in Section 3.5.

3.1. Preliminaries

We use P to denote the Android app under test. We start by formally defining *event* and *task*.

Definition 1 *(Event).* An *event* e is what an Android app takes as input and responds to. An app receives events and executes its corresponding code to handle these events. We use E to denote the set of all possible events for the app under test P.

Sometimes, an Android app will not respond to an event until it receives another certain event first. We define such pre-required events as *enabling events*.

Definition 2 *(Enabling events).* An event e' is another event e's *enabling event* if the app P does not respond to e until it receives e' first.

When receiving input events, Android apps post *tasks* to handle these events.

Definition 3 *(Task).* A *task* t is what an Android app posts on different threads to handle input events. It represents a method of an app component such as a `Listener`, a native `Thread`, etc. An event e *triggers* a task if the task is posted to handle this event. We use $Tr(t)$ to denote events triggering t.

When handling an event sequence, tasks are posted on different threads and can have different execution orders. We define the *schedule of tasks* based on this observation.

Definition 4 *(Schedule of tasks).* For an event sequence $\overline{seq}$, a *schedule* of tasks $\overline{sch} = [(t_1, th_1), (t_2, th_2), ..., (t_n, th_n)]$, where $t_i \in T$ is a task executed on thread th_i, is a possible posting and execution order of tasks handling $\overline{seq}$. For a certain $\overline{seq}$, there can be multiple possible schedules. We denote all possible schedules of tasks of P for $\overline{seq}$ as $SCH(\overline{seq}, P)$.

During the execution of a task, it can access shared resources. We define *access point* (*AP* for abbreviation) based on this observation.

Definition 5 *(Access point).* An *Access Point* AP is a program point of a task t where t accesses a resource shared by other tasks, and t is the AP's *belonged task*.

Note that an AP can be either a read operation (i.e., a read AP) or a write operation (i.e., a write AP). We follow the definition of non-commutative race [5] and define conflicting APs and tasks.

Definition 6 *(Conflicting APs and tasks).* Two APs AP_1 and AP_2 are *conflicting* if they can access the same resource and at least one of them is a write AP. Their belonged tasks t_1 and t_2, where $AP_1 \in t_1$ and $AP_2 \in t_2$, are considered *conflicting* and *non-commutative*. We use $CONF(\overline{seq})$ to denote all conflicting task pairs triggered by an event sequence $\overline{seq}$.

Particular schedules of such conflicting tasks and their triggering events can manifest most concurrency bugs [3–5]. Therefore, we focus on finding such tasks and events, and generating all possible schedules for them.

3.2. Overview

Our approach aims to detect potential concurrency bugs in Android apps by manifesting these bugs during the execution of an app. Note that we only detect concurrency bugs within apps and not across apps. As shown in Section 2.3, concurrency bugs are caused by abnormal schedules of both conflicting tasks and events these tasks handle. Therefore, the two key factors of manifesting a concurrency bug in an Android app are: (1) identifying events potentially relating to concurrency bugs and conflicting tasks triggered by these events, and (2) enumerating all possible schedules of such events and tasks. In order to address these two key factors, we adopt a two-phase approach that works as follows:

1. A pre-processing phase that identifies conflicting tasks and concurrency-bug related events. This is achieved by a systematic dynamic analysis guided by GreenDroid [6].
2. A manifestation phase that proactively generates potentially concurrency-bug related events, and enumerates all possible schedules of both these events and conflicting tasks triggered by them, during a depth-first GUI model exploration.

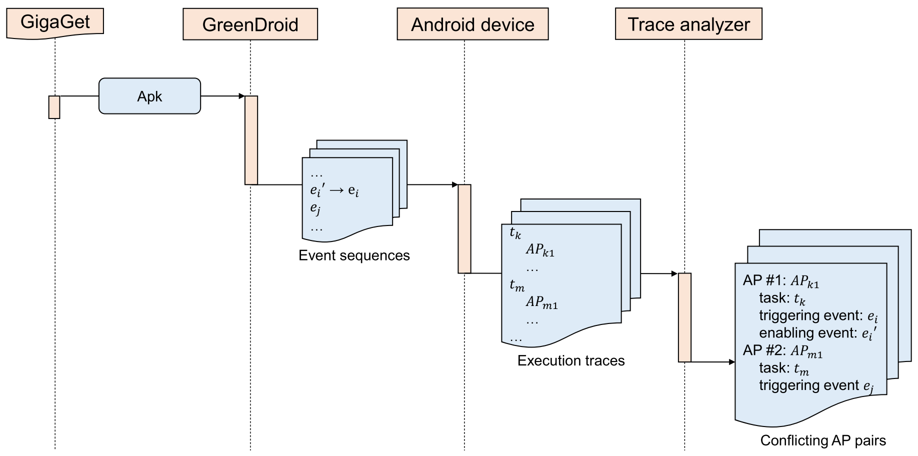


Fig. 3. Work flow of the pre-processing phase.

For the pre-processing phase, we adapt GreenDroid [6], a powerful Android app state space exploration engine originally designed for detecting energy inefficiency bugs of Android apps, to guide the dynamic analysis in order to identify conflicting tasks and concurrency-bug related events. GreenDroid systematically executes an Android app in a virtual environment to explore each event handler's behavior, which fits well the task of finding conflicting tasks. We will address more of this phase in Section 3.3.

We then use the information about events and conflicting tasks obtained in the pre-processing phase to guide the manifestation phase. We conduct a depth-first exploration on an Android app's GUI model. During the exploration, we generate k simultaneous events potentially relating to concurrency bugs at each state, and schedule both these events and the conflicting tasks posted to handle them. We address more of this phase in Section 3.4.

Our approach works as follows for the motivating example from GigaGet. In the pre-processing phase, GreenDroid guides the dynamic analysis to systematically explore the state space of GigaGet. During the exploration, it notices that the `deleteItem` task accesses shared resources (i.e., the `item` object and the file it represents) and conflicts with itself. Its belonged task is the `onMenuItemClick` task, which is enabled by the `onClick` task. Therefore, we record these tasks and the input events these tasks handle. Then in the manifestation phase, a depth-first exploration is conducted. When it arrives at the state where the `item` object's click event is enabled, it enumerates all possible schedules of two click events on the `item` object and one click event on each `del` menu item, as well as the tasks posted to handle these events. When it schedules two click events on the `item` object before the `show` task of the popup menu is posted and executed, and then schedules one click event on each `del` menu item, it successfully manifests the concurrency bug.

3.3. Dynamic access point analysis

To manifest concurrency bugs in our manifestation phase, we need to generate potentially concurrency-bug related events, and schedule these events as well as conflicting tasks posted to handle them. Therefore, our goals in the pre-precessing phase are to: (1) obtain all APs $\{AP_1, AP_2, .., AP_n\}$ and determine if any of them are conflicting with others, (2) obtain conflicting tasks $\{t_1, t_2, .., t_m\}$ where $AP_i \in t_i$ for each AP_i conflicting with others, and (3) obtain concurrency-bug related events $\{Tr(t_1), Tr(t_2), ..., Tr(t_m)\}$, which can trigger these conflicting tasks.

Fig. 3 presents the work flow of the pre-processing phase. As described in Section 3.2, in order to obtain the above information, we conduct a GreenDroid-enhanced dynamic analysis on the app under test. GreenDroid is originally designed for detecting energy inefficiency bugs in Android apps, and is able to systematically execute an Android app. It statically analyzes an Android app's configuration files to collect events acceptable by each app component (e.g., an Activity). Then during the state space exploration, it exhaustively enumerates all possible events acceptable by current active components at each state. In this manner, GreenDroid generates all possible event sequences,[1] and systematically explores the state space of an Android app.

After GreenDroid provides event sequences, we use these sequences to guide our dynamic analysis. We execute the Android app under test on a real Android device with these event sequences as inputs. During the execution, the Android device records execution traces of method invocations and field accesses, which relate to task posting and shared resource accesses, respectively. We then parse the execution traces to collect information about events, tasks, and APs, and determine if any task is conflicting with others, and if any event has enabling events. We record their information for our analysis in the manifestation phase.

[1] Since an event sequence of an Android app can be infinitely long, GreenDroid sets up a length limit l and generates all possible event sequences no longer than l. This is sufficient since all reachable handlers can be reached with a reasonably large l.

Note that we regard all tasks with write APs conflicting with themselves and record their information. This is because that app users can send the same event to the app twice, and an abnormal schedule of two same tasks handling these events can indeed lead to concurrency bugs. Our motivating example from GigaGet is one example.

3.4. Automated testing with guided event generation

With all the information obtained in the pre-processing phase, we attempt to manifest concurrency bugs in our manifestation phase. Our algorithm of the manifestation phase is a scheduling oriented DFS (SO-DFS). As Algorithm 1 and Algorithm 2 show, the whole algorithm consists of two parts: a depth-first state-space exploration (i.e., state space explo-

Algorithm 1: State space exploration.

```
1  $S \leftarrow \emptyset$ // explored states
2  Function SO-DFS($s, \pi$)
       // $s$ is the current state
       // $\pi$ is the event sequence required to reach $s$
3      $S \leftarrow S \cup \{s\}$
4      $E \leftarrow$ getEvents($s$)
       // generate event-schedule combinations at $s$
5      generateSchedules($E, s, \pi$)
6      for each event $e \in E$ do
7          sendEvent($e$)
8          $s' \leftarrow$ getCurrentState()
9          if $s' \notin S$ then
               // "::" denotes list concatenation
10             SO-DFS($s', \pi :: \langle e \rangle$)
11         if $s \neq s'$ then
12             RESTORE($s, \pi$)
```

Algorithm 2: Event-schedule combination generation.

```
1  k: maximum number of events scheduled each time
2  Function generateSchedules($E, s, \pi$)
3      $\mathcal{E} \leftarrow E \cup E^2 \cup \ldots \cup E^k$
4      for each $es \in \mathcal{E}$ do
5          if triggerConflictingTasks($es$) then
6              $Seq \leftarrow \emptyset$
               // check if each event has an enabling event
7              for each $e \in es$ do
8                  if $e$ has an enabling event $e'$ then
9                      $Seq \leftarrow Seq \cup [e', e]$
10                 else
11                     $Seq \leftarrow Seq \cup [e]$
               // obtain all event sequences
12             $ESC \leftarrow$ generateEventSchedules($Seq$)
13             for each $esc \in ESC$ do
14                 generateTaskSchedules($esc, s, \pi$)
15 Function generateTaskSchedules($esc, s, \pi$)
16     for each $\langle t_1, t_2 \rangle \in CONF(esc)$ executed in different threads do
17         $CA \leftarrow$ obtainConflictingAPPairs($t_1, t_2$)
18         for each AP pair $\langle AP_1, AP_2 \rangle \in CA$ do
               // unblock $AP_1$ first
19             executeSchedule($esc, t_1, AP_1, t_2, AP_2$)
20             RESTORE($s, \pi$)
               // unblock $AP_2$ first
21             executeSchedule($esc, t_2, AP_2, t_1, AP_1$)
22             RESTORE($s, \pi$)
23 Function executeSchedule($esc, t_1, AP_1, t_2, AP_2$)
24     $\langle b_1, b_2 \rangle \leftarrow$ sendEventAndBlock($esc, t_1, AP_1, t_2, AP_2$)
25     unblock($b_1$)
26     unblock($b_2$)
```

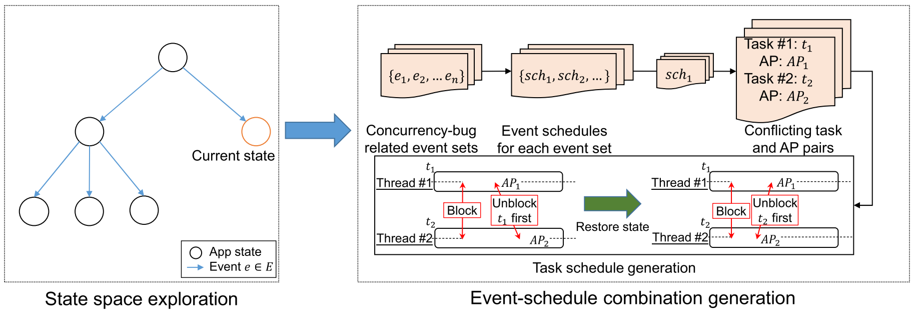


Fig. 4. Work flow of SO-DFS.

ration) and a pressure testing where we generate schedules for concurrency-bug related events and conflicting tasks (i.e., event-schedule combination generation). Fig. 4 presents the work flow of our SO-DFS.

The state space exploration part resembles the standard DFS algorithm [14]. We automatically run an Android app in a depth-first fashion based on a GUI state model where states of an Android app are defined as its GUI layouts. Algorithm 1 presents the recursive exploration procedure. When calling SO-DFS with a state s and an event sequence π (Line 2), which represent the current app state and the event sequence used to reach this state, respectively, we first obtain all events acceptable to the app at s (Line 4), and try to manifest concurrency bugs with them (Line 5), as presented in Algorithm 2. After the manifestation finishes, we systematically send all events acceptable by the app at state s as input, attempting to explore more app states (Lines 6–12). We recursively call SO-DFS if the result state s' has not been explored yet (Lines 9–10). We restore state s after each state exploration attempt (Lines 11–12).

During the state space exploration, we generate schedules for concurrency-bug related events and conflicting tasks at each app state. Algorithm 2 presents the detailed event-schedule combination generation procedure. Each time, we generate schedules of at most k events that trigger conflicting tasks (Lines 4–14). Note that we allow events to be selected repetitively each time since repetitive events can trigger tasks conflicting with themselves, and thus can also lead to concurrency bugs. Moreover, we take *life-cycle events* of app components into consideration. In order to manage resources, the Android system provides various life-cycle events for app components. Such events can change states of these components, and affect tasks associated with these components. We include these events into each $es \in \mathcal{E}$ to further manifest potential concurrency bugs.

For each event e, we first determine whether it has an enabling event e' (Lines 7–8). If so, we add e' in front of e and maintain such order during the schedule generation process (Line 9). We then generate schedules for both these events and conflicting tasks triggered by them. We first combine all events in Seq, and generate all possible event sequences (i.e., event schedules) ESC (Line 12). For each $esc \in ESC$, we then generate schedules for conflicting tasks triggered by esc via `generateTaskSchedules` (Lines 15–22).

In `generateTaskSchedules`, we generate schedules for each conflicting AP pair in each task pair (Lines 16–22). For each pair of conflicting APs $\langle AP_1, AP_2\rangle$ in each conflicting task pair $\langle t_1, t_2\rangle$ executed in different threads, where $AP_1 \in t_1$ and $AP_2 \in t_2$, we send the event sequence esc to the app and block threads executing t_1 and t_2 at the points before they execute AP_1 and AP_2 (Line 24). We unblock AP_1 first, and then unblock AP_2 (Lines 19, 25 and 26). We then restore the app to the state where esc has not been sent (Line 20), send esc to the app again, and block the threads before AP_1 and AP_2 are executed (Line 24). This time we unblock AP_2 first, and then unblock AP_1 (Lines 21, 25 and 26), yielding a different execution order. As such, we generate different schedules of each conflicting AP pair. By scheduling each pair of conflicting APs in each pair of conflicting tasks executed in different threads, we enumerate all possible schedules for conflicting tasks triggered by each esc.

3.5. Discussions

We made several improvement efforts from our previous work AATT for both analysis phases.

For the pre-processing phase, our previous work AATT relies on both static and dynamic analyses. Static analysis obtains static data/control-flow information to build call graphs of all tasks of an Android app in order to determine if any of them are conflicting with others. However, an Android app's data/control-flow often travels out of the app and into the Android framework, and this makes it difficult to build precise and complete call graphs of the app. This lack of precision and completion can lead to both false negatives and false positives in identifying conflicting APs. Dynamic analysis, on the other hand, provides the precise information of execution of an Android app. In our previous work AATT, we run an Android app with existing event generation techniques that are not originally designed for detecting concurrency bugs in Android apps, which cannot reach high coverage of tasks, since many of them only focus on events with certain features [12,15,16]. In order to overcome these disadvantages, we adopt GreenDroid to guide our dynamic analysis. Moreover, we obtain information about enabling events, while AATT does not consider such events.

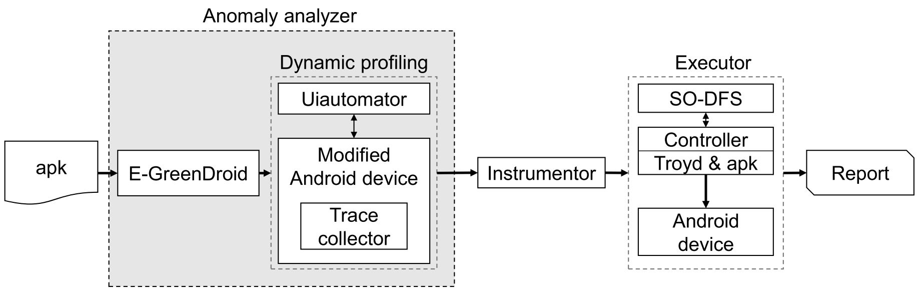


Fig. 5. Architecture of implementation.

For the manifestation phase, we take life-cycle events into consideration when generating event-schedule combinations. Life-cycle events can affect the execution of tasks even though their triggering tasks have no explicit conflicting APs. While our previous work AATT does not consider these events, we add them to each event schedule to better manifest concurrency bugs in AATT+.

4. Implementation

We implemented our prototype tool AATT+[2] whose architecture is presented in Fig. 5. The `Anomaly analyzer` component and the `Instrumentor` component together comprise the pre-processing phase of our approach, while the `Executor` component is implemented as the manifestation phase of our approach. The `Anomaly analyzer` component integrates the `E-GreenDroid` component and the `Dynamic profiling` component. It takes an Android app's apk file as input, and returns a set of execution traces of the app as output, which contains information about events, tasks and APs. The `E-GreenDroid` component generates various event sequences of the app under test. The `Dynamic profiling` component uses these event sequences to systematically explore the state space of the app under test, while its `Trace collector` collects traces of the execution. Then the `Instrumentor` component takes the execution traces as input. It parses the traces to identify conflicting APs and tasks, infers enabling and concurrency-bug related events, and modifies the apk file accordingly in order to allow the `Executor` component to perform `SO-DFS`. Finally, the `Executor` component automatically runs the app under the control strategy described in Section 3.4 using the modified apk file to manifest concurrency bugs. It generates an analysis report about the manifestation process. The report provides the event traces used to reach each state and the event-schedule combinations generated at each state. If the app crashes during the manifestation, the report further provides information about the crash, such as stack information.

We use `E-GreenDroid` [6] to provide event sequences for the app under test. `E-GreenDroid` is an updated version from the original GreenDroid on both methodology and implementation, which can analyze apps of API Level 21 or earlier. `E-GreenDroid` produces desired event sequences to be sent to the `Dynamic profiling` component. One limitation of `E-GreenDroid` is that it does not support concurrency. For asynchronous tasks, `E-GreenDroid` simply ignores them. In order for `E-GreenDroid` to explore asynchronous tasks, for current implementation we modify it to execute these tasks synchronically. When an asynchronous task is posted, instead of ignoring it, our modified `E-GreenDroid` executes this task in the main thread. This modification is temporary for our current implementation, and although it might introduce atomicity violations, we have not encountered such situations in our experiments. We plan to extend `E-GreenDroid` in future to fully support concurrency.

The `Dynamic profiling` component consists of two sub-components, a controller `Uiautomator` and a `Modified Android device`. The `Uiautomator` takes the event sequences from `E-GreenDroid` and uses them to guide the execution of the app under test on the `Modified Android device`. The `Modified Android device` is a real Android Device with a `Trace collector`. It executes the app under the guidance of the `Uiautomator`. During the execution, the `Trace collector` records the execution traces, which will be used by the `Instrumentor` component. The `Trace collector` is implemented by modifying the interpreter of ART [17] to collect execution traces. The traces contain following information: (1) memory access information from field read/write operations, and (2) method entry and exit operations. When the execution finishes, these traces are sent to the `Instrumentor` component.

The `Instrumentor` component is responsible for parsing execution traces to identify conflicting APs, conflicting tasks, and concurrency-bug related or enabling events. It uses the method entry and exit information to determine the caller and the callee of each method invocation, and thus determines who posts each task. Moreover, the `Instrumentor` component parses the memory access information to find APs of each task. Particularly, we focus on APs accessing shared memories, databases, and file systems. By comparing each pair of APs, it can determine which pairs of APs or tasks are conflicting with each other. For each task, if it contains any AP conflicting with that in another task, we record this task, its conflicting APs, the event triggering it, and the enabling event of the triggering event.

[2] The prototype tool can be downloaded at https://github.com/skull591/AATT.

Moreover, in order to allow the `Executor` component to properly schedule conflicting tasks, the `Instrumentor` component inserts extra control statements into the apk file. Recall that in order to properly schedule tasks and APs, we block certain threads, and unblock them in all possible orders. In order to block/unblock threads, the `Instrumentor` component inserts semaphore operations at the beginning of each conflicting task and before each conflicting AP. Moreover, it inserts an extra Java class into the apk file, which is responsible for controlling the inserted semaphores. As such, our `Controller` of the `Executor` component can control schedules of conflicting tasks.

The `Executor` component, which is built on top of our previous testing framework ATT [18], first re-signs the instrumented apk file and implants a service *Troyd* in it. Troyd [19] runs in the same process as the app under test and collects the app's runtime information under given commands. The `Executor` component then installs the re-signed apk file on a real Android device. The `Controller` of the `Executor` component runs on a computer to guide the execution of the app with `SO-DFS`. It guides the execution to traverse the app's state space. As described in Section 3.4, we define the current state of a running app as its current GUI layout, whose hash value is calculated with the coordinates, sizes and types of the layout's widgets. At each state, the `Controller` uses information obtained by the `Instrumentor` component to generate suspicious event-schedule combinations to manifest concurrency bugs. For our current implementation, we set $k = 2$ for event-schedule combination generation. For event schedule generation, the `Controller` forms different event sequences and sends them to the app. For task schedule generation, the `Controller` interacts with the inserted Java class to take semaphore operations. It takes `P` operations on inserted semaphores to block threads and takes `V` operations to unblock them. During the manifestation phase, the `Executor` records event sequences used in state traversing and event-schedule combination generation, and puts them into the output report. If the app under test crashes during the execution, the `Executor` gathers relevant information such as stack backtrack, and puts the information into the report as well.

Some app states require designated inputs (e.g., a user name and password combination) to reach. Automatically generating such inputs is still an open research problem [20]. Therefore, we prepare each app with necessary inputs and when such a situation occurs, the `Controller` will automatically feed it with meaningful inputs.

5. Evaluation

We experimentally evaluated our tool by applying it to real-world Android apps. We focused on the effectiveness and efficiency of AATT+. Moreover, we investigated the effectiveness of our extension from AATT to AATT+. In summary, we aimed to investigate following research questions:

- **RQ1 (Effectiveness):** Can AATT+ manifest concurrency bugs more effectively than existing techniques?
- **RQ2 (Efficiency):** Does AATT+ consume a reasonable amount of time when conducting analysis?
- **RQ3 (Improvement):** How is AATT+ improved from AATT?

For **RQ1** and **RQ2**, we compared our AATT+ with: (1) the industrial standard random testing tool Monkey [21], and (2) an enhanced model-based depth-first search (DFS) [14] that sends each event twice quickly in order to better manifest concurrency bugs.

For **RQ3**, we compared the effectiveness and efficiency of AATT+ with our previous work AATT to evaluate our extension.

Note that for AATT+, AATT, Monkey, and DFS, they detect concurrency bugs by manifesting them during the execution of an app, while techniques utilizing Predictive Trace Analysis (PTA) [3,4] detect concurrency bugs by analyzing execution traces. To distinguish such difference, in this section we use the term *manifest* for AATT+, AATT, Monkey, and DFS, and the term *detect* for techniques utilizing PTA.

5.1. Experimental setup

We evaluated our AATT+ using two sets of real-world Android apps: one with apps that have previously *known* concurrency bugs, and one with randomly selected popular apps utilizing concurrency. Table 1 shows details of these test subjects.

Table 1(a) shows the first set of test subjects, which have known concurrency bugs. This set contains 15 real-world Android apps, which are: (1) popular, large-scale apps with confirmed concurrency-bug related issues or with commits that fix concurrency bugs, or (2) faulty versions of buggy test subjects used by existing work [3,4]. We refer to this set of test subjects as the *Knowns*.

Table 1(b) shows the second set of test subjects. It contains 17 randomly selected popular real-world Android apps with concurrency from GitHub, Google Code, and EOEAndroid. We refer to this set of test subjects as the *Randoms*.

The research questions were investigated using all these 32 apps, which cover 12 app categories and have an average of over 18,000 lines of code. For subjects in the *Knowns* (apps with ground-truth bugs), the effectiveness was reflected by whether a technique could manifest the known concurrency bugs. For subjects in the *Randoms* (randomly selected apps), we investigated the concurrency bugs reported by each technique to determine whether they were real concurrency bugs. Previously unknown bugs were reported to the developers of buggy apps for confirmation.

Table 1
Test subjects of our evaluation.

App	Availability	LoC	Category
	(a) Test subjects of *Knowns*		
vlillechecker	Github	5,330	Travel & Local
AnyMemo	GitHub	29,871	Education
OIFileManager	GitHub	3,484	Productivity
Tomdroid	GitHub	12,233	Productivity
SunShine	GitHub	26,472	Weather
MyTrack	Google Code	43,372	Communication
ChatSecure	GitHub	68,681	Communication
Feedex	GitHub	10,151	News & Magazines
sgtpuzzles	GitHub	5,780	Games
K-9 Mail	GitHub	95,098	Communication
todowidget	GitHub	689	Productivity
AAT	GitHub	43,725	Travel & Local
Douya	GitHub	34,357	Social
weiciyuan	GitHub	78,936	Social
FBReader	GitHub	16,578	Books & References
	(b) Test subjects of *Randoms*		
2buntu	GitHub	963	News & Magazines
aarddict	GitHub	2,077	Books & Reference
aNarXiv	GitHub	3,357	Books & Reference
andiodine	GitHub	1,502	Tools
Down	EOEAndroid	2,045	Tools
DroidWeight	Google Code	5,078	Health & Fitness
exeternalIP	GitHub	2,416	Tools
falling blocks	Google Code	1,763	Games
GigaGet	GitHub	3,123	Tools
HostIsDown	GitHub	631	Tools
KindMind	GitHub	5,510	Lifestyle
LilyDroid	Google Code	10,471	Social
MultiPing	GitHub	547	Tools
ConnectBot	GitHub	26,567	Tools
CoolClock	GitHub	3655	Personalization
Simple Draw	GitHub	12,190	Tools
AnkiDroid	GitHub	23,769	Education

For each test subject, AATT+, AATT and the enhanced DFS ran until completion, i.e. all reachable transitions were explored at least once. To raise the Monkey's probability of manifesting concurrency bugs, we provided Monkey twice as much time as AATT+ took for each test subject.

All experiments were conducted on a machine with Intel Core i5-4200U CPU and 4 GB RAM running Ubuntu 16.04 LTS. Apps were tested on a Google Nexus 5 with Android 6.0.

5.2. Experimental results

Our evaluation results are presented in Table 2. For each test subject, we present the analysis result of each technique, along with the execution time each technique took to conduct analysis. We answer the research questions based on these results.

5.2.1. ***RQ1:*** *effectiveness*

We can answer **RQ1** by comparing the analysis results of AATT+, enhanced DFS and Monkey. For all 32 test subjects, AATT+ reported that 21 subjects had concurrency bugs leading to app crashes or malfunctions. For subjects of the *Knowns*, AATT+ reported that 10 out of 15 apps had potential concurrency bugs. We inspected the reports of AATT+ and confirmed that they all matched the ground truth. Five known concurrency bugs were not manifested by AATT+, two of which were reported by existing work utilizing PTA [3,4]. We will discuss these un-manifested bugs in Section 5.4. For test subjects of the *Randoms*, AATT+ reported that 11 out of 17 apps had concurrency bugs. We carefully examined these apps. All manifested bugs were real bugs concerning concurrency. Since we did not have the ground truth for these bugs, we searched repositories and issue tracking systems of these problematic apps to determine whether the bugs were previously unknown. We found that 7/11 of the bugs were previously unknown. We submitted bug reports to app developers for active projects. The developers confirmed two of the bugs [22,23], and labeled the issue we posted for the bug in andiodine as *bug* [24], although they have not manifested the bug themselves yet.

For Monkey, it manifested only one bug for each set of test subjects (in Down and todowidget), which were also manifested by AATT+. For enhanced DFS, it manifested only one concurrency bug (in GigaGet of the *Randoms*), which our AATT+ manifested as well. Note that it cannot manifest any concurrency bug without our enhancement.

Table 2
Manifestation results.

App	AATT+		DFS		Monkey		AATT	
	Time[a]	Bug	Time	Bug	Time	Bug	Time	Bug
(a) Manifestation results for *Knowns*								
vlilleChecker	1,425	yes	2,835	–	2,850	–	1,524	yes
AnyMemo	872	yes	758	–	1,744	–	1,538	–
OIFileManager	980	yes	2,258	–	1,960	–	870	–
SunShine	2,375	–	2,036	–	4,750	–	2,499	–
Tomdroid	635	yes	1,926	–	1,270	–	561	yes
MyTrack	456	yes	778	–	912	–	1,327	–
ChatSecure	1,358	yes	2,996	–	2,716	–	1,784	–
Feedex	1,028	yes	3,768	–	2,056	–	1,976	–
sgtpuzzles	1,256	–	947	–	2,512	–	1,114	–
k-9 Mail	1,263	yes	1,274	–	2,516	–	1,756	–
todowidgit	685	yes	568	–	1,370	yes	1,563	–
AAT	1,768	–	1,528	–	3,536	–	1,267	–
Douya	1,029	–	1,128	–	2,058	–	923	–
weiciyuan	637	–	537	–	1,274	–	588	–
FBReader	858	yes	1,433	–	1,716	–	1,567	–
(b) Manifestation results for *Randoms*								
2buntu	1,876	–	875	–	3,752	–	1,416	–
aarddict	275	yes	1,252	–	550	–	158	yes
aNarXiv	376	yes	956	–	752	–	203	yes
andiodine	2,450	yes	2,258	–	4,900	–	2,247	yes
Down	98	yes	281	–	26	yes	82	yes
DroidWeight	6,247	–	2,320	–	12,494	–	6,235	–
externalId	450	–	223	–	900	–	449	–
falling blocks	226	yes	158	–	452	–	215	yes
GigaGet	643	yes	271	yes	1,286	–	655	yes
HostIsDown	237	yes	819	–	474	–	231	yes
KindMind	2,679	–	1,478	–	5,358	–	2,640	–
LilyDroid	1,023	yes	11,909	–	2,046	–	795	yes
MultiPing	425	yes	440	–	850	–	375	yes
ConnectBot	875	yes	1,253	–	1,750	–	1,019	–
CoolClock	334	–	235	–	668	–	310	–
Simple Draw	612	–	552	–	1,224	–	483	–
AnkiDroid	486	yes	789	–	972	–	671	–

[a] Column **Time** presents the consumed time when it hits the first bug, or presents all time that they need if the target app does not crash.

These results demonstrate that AATT+ is promising in manifesting concurrency bugs, which are difficult to manifest by conventional techniques. Though Monkey can manifest any bug in theory, the evaluation shows that our approach is much more effective, especially when the resources are limited. Therefore, we can answer **RQ1** that AATT+ can effectively manifest concurrency bugs, comparing with existing techniques.

*5.2.2. **RQ2:** efficiency*

In order to answer **RQ2**, we compared the execution time of AATT+, enhanced DFS and Monkey, which is presented in Column `Time` of Table 2.

For all buggy test subjects except andiodine, AATT+ manifested the bugs within 1,500 seconds. If no bug was manifested, AATT+ spent slightly more time than enhanced DFS did.

Enhanced DFS manifested the bug in GigaGet with slightly less time than AATT+ did. It also took less time to conduct analysis than AATT+ did if no bug was manifested except for Douya. This is because that it merely traverses the states of test subjects without generating event-schedule combinations at each state. Moreover, Monkey consumed twice as much time as AATT+ did with much less productive results.

The comparison demonstrates that AATT+ can manifest concurrency bugs with little overhead comparing with existing techniques, and this suggests its efficiency.

*5.2.3. **RQ3:** improvement*

To answer **RQ3**, we further used AATT to analyze our test subjects. The results are also presented in Table 2. AATT were originally tested by a subset of test subjects of the *Randoms*, therefore we compared the results as regression testing to determine whether our extension preserves the effectiveness of AATT. As Table 2(b) shows, AATT+ manifested all the bugs manifested by AATT. This indicates that AATT+ can still effectively manifest concurrency bugs that AATT can.

To further demonstrate the effectiveness of our extension, we further compared the results for test subjects of the *Knowns* and the *Randoms*. As the Table 2(a) shows, AATT only reported known bugs in two apps, leaving bugs in the rest 13 apps un-manifested. As a comparison, AATT+ successfully manifested concurrency bugs in 10 of 15 apps, as we discussed in Section 5.2.1. Moveover, AATT+ also reported bugs in 11 test subjects of the *Randoms*, as Table 2(b) shows, two of which was not manifested by AATT. According to our further investigation, the difference is due to following reasons:

Capability of identifying conflicting APs AATT+'s extended analysis of the pre-processing phase (i.e., the GreenDroid-enhanced dynamic analysis) is more capable of identifying conflicting APs and tasks. We compared the conflicting APs and tasks found by AATT and AATT+. The static-dynamic hybrid analysis of AATT often missed crucial APs, which were required for manifesting concurrency bugs in some test subjects. This is because that: (1) it ignores enabling events of concurrency-bug related events, which can be useful for manifesting bugs, and (2) the imprecise and incomplete call graphs from static analysis can lead to missing important conflicting APs. We found two cases (AnyMemo and k-9 Mail) where the lack of conflicting APs led to false negatives. On the other hand, with the GreenDroid-enhanced dynamic analysis, AATT+ successfully found such crucial conflicting APs and tasks, and thus successfully manifested the concurrency bugs.

Capability of scheduling life-cycle events AATT+'s extended analysis of the manifestation phase adopts heuristic methods for component life-cycle event scheduling, while AATT lacks the ability of scheduling these events. For instance, our test subject AnyMemo contains a background task that downloads files. It shows a dialogue at foreground when the background downloading finishes. If the Activity object's `onDestroy` task is posted before the dialogue shows, the downloaded content will be discarded and the file will be re-downloaded. Here the precise schedule of the `onDestroy` task is crucial for manifesting the bug. However, AATT is completely unable to achieve such schedule. As described in Section 3.4, our heuristic methods for scheduling component life-cycle events is capable of achieving such scheduling. By rotating the screen which resulted in destroying and recreating the Activity object, AATT+ was capable of manifesting the concurrency bug in AnyMemo.

Due to these findings and the comparison, we can determine that AATT+ is more capable of manifesting concurrency bugs than AATT.

For efficiency, we compared the execution time of AATT and AATT+. In general, AATT+ took more time to analyze the test subjects, due to the heuristic scheduling of component life-cycle events, which is necessary for manifesting concurrency bugs concerning these events. Nevertheless, the overhead is tolerable, comparing with the overall execution time and considering the benefits it brings. Therefore, we can conclude that AATT+ preserves the efficiency.

With all these results, we can safely answer **RQ3** that AATT+ can better manifest concurrency bugs than AATT while preserving the efficiency, i.e. the extension indeed improves AATT.

5.3. Discussions

We manually inspected all concurrency bugs in our test subjects and summarized the common root causes and symptoms of these bugs.

Root cause. The root causes of concurrency bugs in our test subjects can be categorized into three different categories. *Atomicity violation* and *order violation* are common causes of concurrency bugs in traditional concurrent programs [8], while *complicated life-cycle events* are a special cause of bugs in Android apps.

- *Atomicity violation*. This is a common cause of concurrency bugs. A set of operations is *atomic*, if it appears to be instantaneous for the rest of the system [25]. Therefore, shared resources should not be accessed by other tasks when a task is accessing them atomically. The violation of atomicity brings non-determinism and data races, and this can lead to concurrency bugs. In our test subjects, some sets of operations are assumed atomic by developers in order to provide determinism for program logic. However, ensuring atomicity could be tricky, and 17 bugs in our test subjects were caused by atomicity violation.
- *Order violation*. This is another common cause of concurrency bugs. Similar with atomicity violation, developers tend to assume a certain execution order of tasks. However, developers can make poor efforts ensuring assumed execution orders, and this could lead to concurrency bugs. We found 9 concurrency bugs due to order violation.
- *Complicated life-cycle events*. It is a special cause of Android apps' concurrency bugs. In order to ensure performance, Android provides complicated life-cycle events for app components. These events can result in changing states of the app components, and this can affect the atomicity and execution order of tasks associated with the components. There were 10 bugs in our test subjects that were caused by not considering these life-cycle events.

Symptom. In order to study what symptoms these concurrency bugs caused, we inspected reports provided by AATT+ and the executions of analysis tools used in our evaluation. We found that concurrency bugs in our test subjects led to both app crashes and malfunctions.

- *Crash bugs*. There are 11/26 concurrency bugs that led to app crashes. The reported exceptions included *Null Pointer Exception (NPE)*, *Concurrent Modification Exception (CME)*, *Bad Token Exception (BTE)*, etc. These bugs are severe since they caused complete breakdown.

- *Functional bugs*. There are 15/26 concurrency bugs that led to different levels of malfunctions. For instance, the concurrency bug in AnyMemo led to re-downloading of the file, and the one in aarddict, a dictionary app, caused the app failing to load words that were searched. These bugs greatly reduced user experience.

5.4. Limitations

In order to investigate the reasons AATT+ failed to manifest known bugs in 5 of the test subjects of the *Knowns*, we further inspected the code of these test subjects. We found that AATT+ failed to manifest the bugs due to following reasons:

Sophisticated event and task schedules As shown in Section 3.4, we generate schedules for events and tasks to manifest concurrency bugs, but some bugs require more sophisticated schedules to manifest. Some bugs require certain schedules of certain asynchronous tasks to manifest, which can be difficult to determine since these tasks can have no explicit conflicting AP. Since AATT+ schedules tasks based on their conflicting APs, it is unable to generate desired schedules for tasks with no explicit conflicting AP. Our test subject SunShine, a weather app, is an example. When starting the app, SunShine's main Activity object's `onCreate` task sets up a background task, which will be executed after some delay. Sunshine then sets up environment in another background thread, and signals the first background thread to execute the delayed task when the setup finishes. SunShine developers assumed that the execution order is guaranteed. However, if a life-cycle event, which results in recreating the Activity object, is sent before the setup finishes, the setup will be stopped and the Activity object's `onRestoreInstance` method, which will be invoked when recreating an Activity object, will signal the background thread to execute the delayed task before the setup completes. This breaks the developers' assumption on the execution order and crashes the app. To manifest this bug, one must send a certain life-cycle event to the app before the setup procedure processed in a background thread finishes, and this is difficult to determine since these tasks have no explicit conflicting AP.

Sophisticated execution paths. Some subjects' concurrency bugs are on execution paths that are difficult to reach. Some of the bugs are contained in execution paths handling exceptions. For instance, the concurrency bug in weiciyuan is contained in execution paths handling network connection exceptions. Moreover, some of the bugs require more complicated input event sequences to manifest. This can be addressed by adopting more powerful event generation techniques.

Two of these five concurrency bugs were detected by existing techniques [3,4], namely the bugs in weiciyuan and Douya. However, they were reported along with many false positives by these existing techniques, and this is a major disadvantage of these techniques, as compared to AATT+.

Further more, although we did not find such bugs in our test subjects, there can be other types of bugs that our current implementation of AATT+ cannot manifest, such as bugs that require more than two simultaneous events to manifest. Addressing these situations can raise the time complexity exponentially. We will address these challenges in our future work. Nevertheless, AATT+ successfully manifested most of the concurrency bugs in our test subjects, and this suggests its effectiveness.

6. Lesson learned

In order to thoroughly study the common patterns of concurrency bugs in our test subjects, we further inspected the bug-related code in these subjects, and summarized the bug characteristics in Table 3. We categorized the bugs by the following four criteria:

- We categorized the bugs by the number of threads they concern. We found that 12 of the bugs were manifested solely in the main thread, while the other 14 bugs involved multiple threads. We name them *single-threaded event-based bugs* [5] and *multi-threaded event-based bugs*, respectively.
- We categorized the bugs by events triggering them. Seven of the bugs can be triggered by a *single event*, while the other 19 bugs required *multiple events* to trigger.
- We also categorized the bugs by their root causes, namely *atomicity violation*, *order violation*, and *complicated life-cycle events*, as described in Section 5.3.
- The symptoms described in Section 5.3 were also used to categorize the bugs, namely *crash bugs* and *functional bugs*.

Summarizing the bugs we categorized, we found that concurrency bugs of Android apps have both similar and different patterns, as compared with those of multi-threaded desktop/server programs [8]. Our findings are summarized in Table 4, and are elaborated on as follows.

Finding (1): All concurrency bugs of our experimental subjects belong to either *atomicity violation* (17/26) or *order violation* (9/26) despite the efforts on ensuring atomicity and execution orders by both developers and the Android system.
Implication: Concurrency bug detection for Android apps should focus at least on these two patterns.

Table 3
Categorization of experimental subjects.

App	Categories[a]			
	Thread	Event	Root cause	Symptom
aarddict	M	S	AV	F
aNarXiv	M	M	AV	C
andiodine	S	M	LC (AV)	C
Down	M	M	AV	C
falling blocks	M	M	AV	F
GigaGet	S	M	LC (AV)	C
HostIsDown	M	M	AV	F
LilyDroid	M	M	AV	C
MultiPing	S	M	LC (OV)	C
ConnectBot	M	S	LC (OV)	C
AnkiDroid	S	M	LC (AV)	F
vlilleChecker	M	S	AV	C
AnyMemo	S	M	LC (AV)	F
OIFileManager	S	M	LC (AV)	F
Tomdroid	S	M	OV	C
MyTrack	S	S	LC (AV)	F
ChatSecure	M	S	OV	F
Feedex	M	M	AV	F
k-9 Mail	M	M	AV	F
todowidgit	S	S	LC (OV)	F
FBReader	S	M	OV	C
SunShine	M	S	LC (OV)	C
sgtpuzzle	M	M	AV	F
AAT	S	M	OV	F
Douya	M	M	AV	F
weiciyuan	S	M	OV	F

[a] Column **Thread**: **S** for *single-threaded bug*, and **M** for *multi-threaded bug*.
Column **Event**: **S** for *single event*, and **M** for *multiple events*.
Column **Root Cause**: **AV** for *atomicity violation*, **OV** for *order violation*, and **LC (XX)** for *complicated life-cycle events* (**XX** presents the ultimate root cause).
Column **Symptom**, **C** for *crash bug*, and **F** for *functional bug*.

Table 4
Summary of Android concurrency bug patterns.[a]

Findings	Implications
(1) All concurrency bugs of our experimental subjects belong to either *atomicity violation* (17/26) or *order violation* (9/26).	Concurrency bug detection for Android apps should at least focus on these two patterns.
(2) *Life-cycle events* for Android app components complicate the concurrent execution and can introduce Android-specific concurrency bugs.	Developers need to understand component life-cycle events for avoiding Android-specific concurrency bugs.
(3) Developers may incorrectly assume the atomicity of a task sequence, which can be broken by another sequence. Particularly, a self-conflicting task can result in *atomicity violation* with itself.	Developers should pay attention to this special type of bugs as it is Android-specific and can often be out of developers' consideration.
(4) Developers may use concurrency mechanisms provided by Android improperly, and this can lead to *order violation*.	Developers of Android apps should understand Android-provided mechanisms and use them properly.

[a] **Atomicity Violation [8]:** *The desired serializability among multiple memory accesses is violated, i.e. a code region is intended to be atomic, but the atomicity is not enforced during the execution.*
Order Violation [8]: *The desired order between two (groups of) memory accesses is flipped, i.e. A should always be executed before B, but the order is not enforced during the execution.*

We found that although the Android system provides various mechanisms to help developers ensure the proper atomicity and execution orders of an Android app's tasks, these two key factors in keeping a multi-threaded program correct [8] can still be violated.

For *atomicity violation*, the Android system does not allow a thread to execute another task until the current one finishes. This ensures that tasks executed in the same thread can be considered atomic. However, when tasks are executed in different threads, or when developers expect a series of tasks to be executed atomically, such mechanism fails to protect the desired atomicity.

For *order violation*, the Android system provides a set of components, such as `AsyncTask` and `Loader`, to help ensure the proper execution order of multi-threaded tasks. However, when developers mix the Android concurrency with native threads instead of using these Android-provided mechanisms, or when they use such mechanisms improperly, concurrency bugs due to order violation can occur.

We further inspected the code of our buggy experimental subjects. We found that a significant number (17/26) of the concurrency bugs in our test subjects are Android-specific. We focused on studying these Android-specific bugs. Our findings are summarized below.

Finding (2): *Life-cycle events* for Android app components complicate the concurrent execution and are a significant cause of Android-specific atomicity/order violation bugs (10/17).
Implication: Developers need to understand component life-cycle events for avoiding Android-specific concurrency bugs.

As described in Section 5.3, component life-cycle events are an important feature of Android, and many researches focus on or concern it [26,27]. From our categorization and investigation, we found that it is common in Android apps that Android-specific concurrency bugs occur due to these complicated life-cycle events. In fact, we found that 10 of our concurrency bugs fall into this category.

We found that a life-cycle event can trigger a concurrency bug even though it does not have any explicit AP. There are two situations. First, a life-cycle event can result in destroying or recreating an app component, and this affects all the background tasks associated with the component. This is the situation where the atomicity of background tasks can be violated. It normally involves background tasks since the main thread cannot execute another task until the current one finishes. Therefore concurrency bugs of this type are normally multi-threaded. The concurrency bug in our test subject AnyMemo is one example, as described in Section 5.2.3. An unexpected recreation of the Activity object breaks the atomicity of a background task, and this leads to a functional bug.

Second, an unexpected schedule of life-cycle events can lead to order violation. Handling a life-cycle event can change the state of an app component, and this can affect the execution order of tasks associated with the component. For example, our test subject SunShine has a concurrency bug due to order violation, as described in Section 5.4. A life-cycle event, which results in recreating the main Activity object, will lead to invoking the main Activity object's `onRestoreInstance` method, which will signal the background thread to execute the delayed task before the setup completes. This breaks the developers' assumption on the execution order and crashes the app.

In summary, *complicated life-cycle events* can lead to both atomicity and order violation. Detecting and understanding this kind of concurrency bugs require deep understanding to such mechanisms of the Android system.

Finding (3): Developers may incorrectly assume the atomicity of a task sequence, which can lead to a concurrency bug if another task sequence, which is triggered by a simultaneous event input and conflicts with the first sequence, breaks the atomicity. Particularly, a self-conflicting task sequence can result in atomicity violation with itself.
Implication: Developers should pay attention to this special type of bugs as it is Android-specific and can often be out of developers' consideration.

We found that when a single input event triggers more than one task, Android developers can assume that the task sequence handling the event will be executed atomically. However, in order to respond to input event sequences quickly, other tasks can be exercised between the executions of a sequence of tasks, and this can break the assumed atomicity. For instance, LilyDroid, an app for a BBS *Little Lily*, accesses a shared resource in a listener task t_l and a task t_b. The t_l task is triggered by an user event, and the t_b task is posted by t_l. Task t_l writes the resource, and later t_b reads it. The developers of LilyDroid assumed that t_l and t_b together are executed atomically and thus does not lock the shared resource. However, in order to respond to another user event, another task t' with a write AP on the same shared resource, which executes in the main thread, can be posted and executed between t_l and t_b. Therefore, the actual execution order can be $(t_l \rightarrow t' \rightarrow t_b)$. This breaks the atomicity assumed by the developers of LilyDroid, and leads to app crash. This is a common pattern of Android apps' concurrency bugs due to *atomicity violation*. We found four concurrency bugs that were of such case.

LilyDroid crashes when two different events trigger two different task sequences, which lead to atomicity violation. Particularly, we found that there are also situations where two identical events trigger a concurrency bug. It is interesting because developers normally do not consider the situation where a task sequence conflicts with itself, and this can lead to concurrency bugs. We discovered three apps, namely GigaGet, MultiPing, and andiodine, that have this kind of bugs. Moreover, we observed that all bugs in GigaGet, MultiPing, and andiodine led to double executions of a one-time only task. GigaGet crashed when two deletions of the same `item` object were exercised. MultiPing, which is a tool for testing network latency, crashes when the same IP address is deleted twice by clicking it twice quickly. As for andiodine, an Android VPN tool, double replacing transactions of the same Fragment object resulted in app crashes. Both of the bugs occurred because an additional task sequence triggered by the second event broke the atomicity assumed by the developers.

Finding (4): Developers may use Android-provided mechanisms improperly, and this can lead to *order violation*.
Implication: Developers of Android apps should understand Android-provided mechanisms and use them properly.

Although the Android system provides mechanisms to ensure the proper execution order of tasks, we found that there are still situations where concurrency bugs appear due to order violation. We found that some developers failed to properly use

Android-provided asynchronous models. For instance, ChatSecure, a popular instant messaging app, uses an `AsyncTask` object to set up environment when an app user starts a group chat. According to the mechanism of `AsyncTask`, the shared resources processed by the `doInBackground` task should be accessed in the `onPostExecute` task to ensure that the resources are properly processed when accessed. However, ChatSecure directly accesses the resources in the main thread after it deploys the `AsyncTask` object, and this can lead to app crashes due to order violation. We found that 3/9 concurrency bugs due to order violation have this pattern. In order to avoid such concurrency bugs, developers should properly use the asynchronous models provided by the Android system.

7. Related work

With continuous advances in mobile computing technologies, the popularity of smartphone uses with ordinary people grows larger and larger. The qualities of smartphone apps become even more important than before. Therefore, software engineering researchers have proposed various approaches to detecting program bugs and ensuring software quality.

Some pieces of work focus on general Android functional testing. DynoDroid [12] proposed generating required events for target apps based on random exploration and it works more efficiently than Monkey. A^{3}E [7] implemented a depth-first search based on the dynamic model derived from an app, which considers each activity as an independent state in the search. Liu et al. [28] proposed VeriDroid, a tool that extends JPF to automatically verify Android apps. EvoDroid [29] proposed combining Android-specific program analysis techniques and evolutionary algorithms for a novel framework for automated testing. However, EvoDroid cannot systematically reason about input conditions. GAT [30] proposed extracting a UI component's relevant gestures through a static analysis, so as to reduce the amount of gesture events to be delivered in the automated testing. UGA [20] leveraged human insights to improve traditional testing approaches' performance. However, these pieces of work have not considered conflicting resource accesses in an app and can miss concurrency bugs easily.

There are also some other pieces of work aiming at detecting date races in Android apps. Both DroidRacer [5] and CAFA [4] proposed generating execution traces by systematically exercising apps and then computing happens-before relations on the traces to detect races. A more thorough and precise happens-before model was recently proposed [3]. The authors also proposed a scalable algorithm to build and query happens-before relation graphs. RacerDroid [31] attempted to manifest data races reported by an existing imprecise race detection tool, but it needs to modify the Android system code for scheduling events and threads. AsyncDroid [32] explored different thread interleavings by repeating given event sequences to detect thrown exceptions and assertion violations. EventTrack [33] proposed using a novel data structure, called event graph, to maintain a subset of happens-before relation and efficiently infer order between each pair of events. Bouajjani et al. [34] defined a correctness criterion, called robustness against concurrency, for a class of event-driven asynchronous programs including Android apps, and provided algorithms for checking such criterion. Hu et al. proposed ERVA [35], a race verification and reproduction approach that identifies true positives and harmful races in race reports produced by existing race detectors.

Some pieces of work also presented techniques that generate input-schedule combinations for traditional concurrent programs. RaceFuzzer [36] proposed executing programs with a randomized thread scheduler, which blocks threads at race points and randomly releases an available thread for execution. Another fully automatic testing technique [37] proposed generating test cases that invoke methods on an instance of the class under test and executing code sequentially to check whether the instance behaves as expected.

Additionally, some researchers paid close attention to ensuring non-functional quality for Android apps. CHECK-DROID [38] is a dynamic analysis tool that automatically detects both functional and non-functional bugs in Android apps such as null pointer exception and resource leak. RepDroid [39] proposed using layout group graph (LGG) to automatically detect app repackaging. PerfDroid [40] is a static analysis tool, which summarizes some performance bug patterns and detects them in Android apps. An automated test framework [41] systematically generates test inputs that may lead to energy hotspots/bugs in Android apps. GreenDroid, E-GreenDroid, and NavyDroid [6,42–44] traverses an app's states as more as possible to find out states with low sensory data utilization coefficient values based on an *application execution model* derived from the Android specification. The authors also published an empirical study about wake lock misuses in Android apps, and proposed ELITE, a wake lock misuse detection tool for Android apps [45]. CyanDroid [46] extends GreenDroid's ability of diagnosing energy inefficiency for Android apps by generating multi-dimensional sensory data and considering app state changes at a finer granularity. However, these pieces of work have not considered concurrent execution of Android apps and cannot be directly applied for detecting concurrency bugs in Android apps.

8. Conclusion and future work

In this article, we proposed a novel approach to detecting concurrency bugs in Android apps by manifesting them during the execution of an app. Our approach adapts GreenDroid to guide the dynamic analysis to identify potentially conflicting APs in an Android app. Our SO-DFS algorithm explores the app's state space and generates event-schedule combinations to exercise the APs to manifest potential concurrency bugs. Our prototype tool implementation AATT+ successfully detected previously unknown concurrency bugs in popular real-world Android apps.

There are still limitations in our approach. First, for now our tool implementation AATT+ supports only two-combination schedules of concurrency-bug related events due to the consideration of efficiency, and temporarily does not support situations where a schedule of more than two conflicting events are required to manifest a concurrency bug. Second, currently the event-schedule combination generator of our tool implementation of AATT+ supports only part of the Android system events due to complicated mechanisms of the Android system and tedious development efforts. We plan to continue to improve our approach and its tool implementation to make it more practical.

Acknowledgements

This work was supported in part by National Key R&D Program (Grant #2017YFB1001801), National Natural Science Foundation (Grants #61690204, #61472174) of China. The authors would also like to thank the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] Number of available Android applications in the Google play store, https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.
[2] Processes and threads, http://developer.android.com/guide/components/processes-and-threads.html.
[3] P. Bielik, V. Raychev, M. Vechev, Scalable race detection for Android applications, in: OOPSLA, 2015.
[4] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, Race detection for event-driven mobile applications, in: PLDI, 2014.
[5] P. Maiya, A. Kanade, R. Majumdar, Race detection for Android applications, in: PLDI, 2014.
[6] J. Wang, Y. Liu, C. Xu, X. Ma, J. Lu, E-GreenDroid: effective energy inefficiency analysis for Android applications, in: Internetware, 2016.
[7] T. Azim, I. Neamtiu, Targeted and depth-first exploration for systematic testing of Android apps, in: OOPSLA, 2013.
[8] S. Lu, S. Park, E. Seo, Y. Zhou, Learning from mistakes – a comprehensive study on real world concurrency bug characteristics, in: ASPLOS, 2008.
[9] Q. Li, Y. Jiang, T. Gu, C. Xu, J. Ma, X. Ma, J. Lu, Effectively manifesting concurrency bugs in Android apps, in: APSEC, 2016.
[10] Android API guides, https://developer.android.com/guide/.
[11] Asynctask, https://developer.android.com/reference/android/os/AsyncTask.html.
[12] A. Machiry, R. Tahiliani, M. Naik, Dynodroid: an input generation system for Android apps, in: FSE, 2013.
[13] C.S. Jensen, A. Møller, V. Raychev, D. Dimitrov, M. Vechev, Stateless model checking of event-driven applications, in: OOPSLA, 2015.
[14] R. Tarjan, Depth-first search and linear graph algorithms, in: SICOMP, 1972.
[15] D. Amalfitano, A.R. Fasolino, P. Tramontana, S.D. Carmine, A.M. Memon, Using GUI ripping for automated testing of Android applications, IEEE Softw. 32 (5) (2015) 53–59.
[16] Y.-M. Baek, D.-H. Bae, Automated model-based Android GUI testing using multi-level GUI comparison criteria, in: ASE, 2016.
[17] Android Runtime, https://source.android.com/devices/tech/dalvik/index.html.
[18] Z. Meng, Y. Jiang, C. Xu, Facilitating reusable and scalable automated testing and analysis for Android apps, in: Internetware, 2015.
[19] J. Jeon, J.S. Foster, Troyd: Integration Testing for Android, Technical Report CS-TR-5013, 2012.
[20] X. Li, Y. Jiang, Y. Liu, C. Xu, X. Ma, J. Lu, User guided automation for testing mobile apps, in: APSEC, 2014.
[21] UI/application exerciser Monkey, https://developer.android.com/studio/test/monkey.html.
[22] Issue #25 of GigaGet, https://github.com/PaperAirplane-Dev-Team/GigaGet/issues/25.
[23] Issue #1 of Down, http://www.eoeandroid.com/thread-311180-1-1.html.
[24] Issue #24 of andiodine, https://github.com/yvesf/andiodine/issues/24.
[25] M.P. Herlihy, J.M. Wing, Axioms for concurrent objects, in: POPL, 1987.
[26] S. Nakajima, Analyzing lifecycle behavior of Android application components, in: COMPSAC, 2015.
[27] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y.L. Traon, D. Octeau, P. McDaniel, Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps, in: PLDI, 2014.
[28] Y. Liu, C. Xu, VeriDroid: automating Android application verification, in: MIDDLEWARE Doctoral Symposium, 2013.
[29] R. Mahmood, N. Mirzaei, S. Malek, Evodroid: segmented evolutionary testing of Android apps, in: FSE, 2014.
[30] X. Wu, Y. Jiang, C. Xu, C. Cao, X. Ma, J. Lu, Testing Android apps via guided gesture event generation, in: APSEC, 2016.
[31] H. Tang, G. Wu, J. Wei, H. Zhong, Generating test cases to expose concurrency bugs in Android applications, in: ASE, 2016.
[32] B.K. Ozkan, M. Emmi, S. Tasiran, Systematic asynchrony bug exploration for Android apps, in: CAV, 2015.
[33] P. Maiya, A. Kanade, Efficient computation of happens-before relation for event-driven programs, in: ISSTA, 2017.
[34] A. Bouajjani, M. Emmi, C. Enea, B.K. Ozkan, S. Tasiran, Verifying robustness of event-driven asynchronous programs against concurrency, in: ESOP, 2017.
[35] Y. Hu, I. Neamtiu, A. Alavi, Automatically verifying and reproducing event-based races in Android apps, in: ISSTA, 2016.
[36] K. Sen, Race directed random testing of concurrent programs, in: PLDI, 2008.
[37] M. Pradel, T.R. Gross, Fully automatic and precise detection of thread safety violations, in: PLDI, 2012.
[38] Y. Liu, C. Xu, S.C. Cheung, W. Yang, CHECKERDROID: automated quality assurance for smartphone applications, Int. J. Softw. Inform. 8 (1) (2014) 21–41.
[39] S. Yue, W. Feng, J. Ma, Y. Jiang, X. Tao, C. Xu, J. Lu, RepDroid: an automated tool for Android application repackaging detection, in: ICPC, 2017.
[40] Y. Liu, C. Xu, S.C. Cheung, Characterizing and detecting performance bugs for smartphone applications, in: ICSE, 2014.
[41] A. Banerjee, L.K. Chong, S. Chattopadhyay, A. Roychoudhury, Detecting energy bugs and hotspots in mobile apps, in: FSE, 2014.
[42] Y. Liu, C. Xu, S.C. Cheung, J. Lu, GreenDroid: automated diagnosis of energy inefficiency for smartphone applications, IEEE Trans. Softw. Eng. 40 (9) (2014) 911–940.
[43] Y. Liu, C. Xu, S.C. Cheung, Where has my battery gone? Finding sensor related energy black holes in smartphone applications, in: PerCom, 2013.
[44] Y. Liu, J. Wang, C. Xu, X. Ma, NavyDroid: detecting energy inefficiency problems for smartphone applications, in: INTERNETWARE, 2017.
[45] Y. Liu, C. Xu, S.C. Cheung, V. Terragni, Understanding and detecting wake lock misuses for Android applications, in: FSE, 2016.
[46] Q. Li, C. Xu, Y. Liu, C. Cao, X. Ma, J. Lu, CyanDroid: stable and effective energy inefficiency diagnosis for Android apps, Sci. China Inf. Sci. 60 (1) (2016) 230–242.