Effectively Manifesting Concurrency Bugs in Android Apps

Qiwei Li, Yanyan Jiang, Tianxiao Gu, Chang Xu*, Jun Ma, Xiaoxing Ma and Jian Lu
State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China
Dept. of Computer Science and Technology, Nanjing University, Nanjing, China
liqiwei1992@gmail.com, jiangyy@outlook.com, tianxiao.gu@gmail.com, {changxu,majun,xxm,lj}@nju.edu.cn

***Abstract*—Smartphones are indispensable in people's daily lives. As smartphone apps are being increasingly concurrent, developers are increasingly unable to tackle the complexity and to avoid subtle concurrency bugs. To better address this issue, we propose a novel approach to manifesting concurrency bugs in Android apps based on the fact that one can simultaneously generate input events and their schedules for an app. We conduct static-dynamic hybrid analysis to find potentially conflicting resource accesses in an app. The app is then automatically pressure-tested by guided event and schedule generation. We implemented the prototype tool AATT and evaluated it over thirteen popular real-world open-source apps. AATT successfully found 9 concurrency bugs out of which 7 were previously unknown.**

I. Introduction

The mobile device and app market flourishes in recent years. At the end of year 2015, there had been more than 1.9 billion apps in the Google Play Store [1].

Concurrency is a key factor in a mobile app: it should quickly respond to incoming events as well as processing time-consuming tasks in the background [2]. However, concurrent programs are notoriously difficult to write, test and debug. Therefore, Android has a set of constraints to avoid concurrency bugs (e.g., Android UI updates are constrained in the Main Thread and all `AsyncTasks` are scheduled in a single background thread). Unfortunately, as mobile apps are being increasingly complicated, developers are unable to correctly understand app's behavior and leave subtle concurrency bugs in the releases.

Existing work [3], [4], [5] tackles the problem of detecting concurrency bugs in an app by extending the concept of data race to event-driven systems. However, their bug detection capabilities heavily depend on the quality of input events that are used to build the event happens-before graph. Furthermore, non-commutative events may not lead to concurrency bugs and these techniques usually have a high false-positive rate [5].

In this paper, we take a different approach to manifesting hidden concurrency bugs in an app. We observed that in mobile app testing, one can simultaneously generate both events and their schedules to manifest potentially mistake event-schedule combinations. To realize this idea, the two challenges are 1) how to figure out which events are potentially related to concurrency bug; and 2) how to systematically generate such events and schedules to manifest real concurrency bugs.

To address the first challenge, we deploy static-dynamic hybrid analysis to find each concurrent task (`Listener`, `Thread` or `AsyncTask`)'s shared resource access sites (i.e., *access points*, APs). Following the definition of non-commutative race [3], two concurrent tasks are conflicting if they both access a particular AP and at least one is a write operation. Conflicting tasks are potentially related in a concurrency bug and should be scheduled by a guided event generation.

To address the second challenge, we propose a scheduling oriented depth-first search (SO-DFS) algorithm that integrates both event generation and schedule generation. SO-DFS is based on the traditional state space exploration algorithm [6] that traverses each transition between distinct app states (defined by the GUI layout) exactly once and is not concurrency-aware. We extend it by systematically manifesting all k-combination schedules of conflicting tasks that are relevant to the current app state (we set $k = 2$ in the implementation for practical consideration).

We implemented our approach as a prototype tool named AATT. We evaluated the effectiveness and efficiency of the tool using popular open-source apps. We found nine concurrency bugs (cause crashes or functional bugs) where seven were *previously unknown* and three are already confirmed by the developers. Detailed evaluation results show that AATT achieved improved effectiveness over concurrency-unaware techniques (DFS) and random testing (Monkey) with a reasonable overhead.

Our contributions are summarized as follows:

1) We proposed an effective approach to manifesting concurrency bugs in Android apps based on hybrid-analysis guided systematic event and schedule generation.
2) We implemented the prototype tool and evaluated it on real-world open-source apps. We found previously unknown bugs and quantitative analysis shows that our approach is both effective and efficient.

The rest of the paper is organized as follows. Section II gives a motivating example. Sections III, IV and V describe our concurrency bug manifesting approach. Sections VI and VII evaluate our approach and discuss its experimental results.

*Chang Xu is the corresponding author.

```
1  downloadItemMenu.setOnClickListener(new
       View.OnClickListener() {
2    @Override
3    public void onClick(View v) {
4      PopupMenu popup = new PopupMenu(context,
           downloadItemMenu);
5      popup.inflate(R.menu.mission);
6      Menu menu = popup.getMenu();
7      MenuItem del = menu.findItem(R.id.del);
8      del.setVisible(true);
9
10     popup.setOnMenuItemClickListener(new
           PopupMenu.OnMenuItemClickListener() {
11       @Override
12       public boolean onMenuItemClick(MenuItem
             item) {
13         if (item.getItemId() == R.id.del) {
14           manager.deleteItem(downloadItem.pos);
15           return true;
16         }
17         return false;
18       }
19     });
20   }
21 });
```

Fig. 1: Simplified code snippet from GigaGet. A particular event schedule allows a download item to be deleted twice, causing an app crash.

Section VIII discusses related work, and finally Section IX concludes this paper.

II. Background and Motivation

The Android concurrency model provides constraints and mechanisms to help programmers tackle the need of multi-tasking and at the same time to avoid errors due to non-determinism. First, all UI and system events are handled *atomically* in the Main Thread to avoid data race on the shared resources. Second, UI updates are only allowed in the Main Thread to eliminate UI update races. Third, time-consuming tasks (e.g., network accesses) are postponed by asynchronous tasks (`AsyncTask` recommended by the Android documentation [7]) that trigger events at completion.

However, as the apps are becoming increasingly complicated, they started to create complicated cascading events that have non-deterministic outcomes [3], [4] as well as to mix native threads (e.g., `Thread` objects and `ThreadPool`) in the execution. Such complication also brings subtle concurrency bugs that are difficult to detect with a limited testing budget.

Figure 1 shows a motivating example of a concurrency bug in GigaGet[1], a lightweight multi-threaded file downloader [8]. When the menu of the download item is clicked (Lines 1–3), it creates a `PopupMenu` (Line 4) and sets its deletion `MenuItem` (named `del`) visible (Lines 7–8). When `del` is clicked, the listener (Lines 10–19) is executed and the download item is removed by `manager` (Line 14). The code snippet seems to work at a first glance. However, if one clicks the menu quickly enough, there can be two identical `PopupMenus` (Line 4) with two `MenuItems` on the screen. Clicking both `dels` leads to double-invocation of `deleteMission` (Lines 13–16) and crashes the app.

This example demonstrates a subtle atomicity violation bug that requires both a specific event sequence and a specific event schedule to trigger: the click event and its handler are assumed to be atomic by the developer (once the `onClick` event is completed, one is not able to click the menu again). An unexpected click event in between (by quickly performing an extra click) breaks such assumption and causes the failure.

Unfortunately, this bug is difficult to detect by the existing predictive trace analysis techniques [3], [4], [5]. The effectiveness of these techniques heavily relies on the input events that produce the execution trace. Unless the trace contains the demonstrated input sequence and the atomicity-breaking interleaving, they cannot detect this bug. Random testing (e.g., Monkey), on the other hand, may have a chance to detect this bug, however, it has low probability to generate the appropriate event sequence and have no bug manifestation guarantee.

This motivates us to design AATT to proactively detect concurrency bugs in Android apps. We observed that triggering a concurrency bug in Android requires both a designated input event sequence and a particular schedule. While the existing work focuses on either event generation [9] or schedule generation [10], it is insufficient to exposure many hidden concurrency bugs. We leverage the interplay of both event and schedule generation at runtime to automatically manifest combinations of conflicting events and thus expose concurrency bugs early.

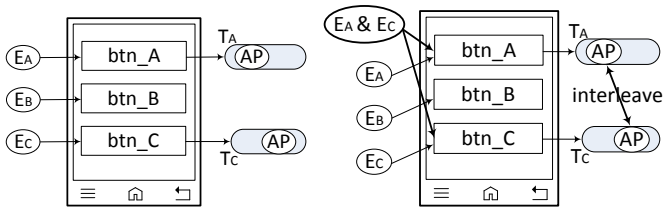
III. Methodology

The two key factors of manifesting a concurrency bug in an app are (1) generating events that are potentially relevant to concurrency bug; (2) enumerating possible schedules for such events. Concurrency bugs are caused by problematic orderings of conflicting shared resource accesses (e.g., data races and atomicity/order violations of the shared memory, database, file system, etc.) [11]. In contrast to testing traditional concurrency programs, we can *simultaneously* generate events and their schedule in a mobile app, yielding a two-phase approach that works as follows:

1) A pre-processing phase that finds each event handler's relevant resource accesses by a hybrid static-dynamic analysis;
2) A manifestation phase that proactively generates potentially concurrency-bug related events as well as their schedules for further checking.

We expand our discussion of the approach by first introducing some definitions. *Task* is the unit of concurrency in an app and can be scheduled by the runtime system. A *task* is created by a GUI *event* and can be a `Listener`, a native `Thread`, or an `AsyncTask`. Any task that is not executed in the Main Thread is called a *background task*. In the Android concurrency model, most tasks are executed

[1]We have reported this bug found by our tool and it has been confirmed by the developers: https://github.com/PaperAirplane-Dev-Team/GigaGet/issues/25.



(a) Conventional approach (b) Our approach

Fig. 2: The comparison between two approaches

atomically, however, their timing/chronological order affects the app's behavior and may lead to concurrency bugs.

A task may access shared resources. We define an *access point* (*AP* for abbreviation) to be a program point that accesses such a resource. Particularly, we focus on the shared memory, database and file system accesses. Note that an AP is either read-only or write. Two APs are *conflicting* if they access the same resource and at least one is a write access. Tasks that contain conflicting APs are considered *conflicting* and *non-commutative*. A particular schedule of such tasks is the root cause of most concurrency bugs [3], [4], [5].

Conflicting tasks are obtained by a hybrid static-dynamic analysis. The static analysis finds each task's relevant APs by call-graph reachability analysis; the dynamic analysis is based on a depth-first exploration of the app's GUI model. These two analyzes complement each other to obtain a set of possible conflicting tasks. The hybrid analysis also obtains each GUI event's relevant tasks.

We use such conflicting information to guide the event generation. Particularly, we generate GUI events at runtime such that all k-combination of conflicting tasks are manifested in testing. Particularly, for any k tasks that each task conflicts with the others, all permutations of them are enumerated in testing ($k = 1$ is equivalent to the existing concurrency-unaware event generation technique [6], and we use $k = 2$ for practical consideration because any number of tasks can lead to exponential complexity).

The basic idea of our approach is illustrated in Figure 2. The activity consists of three buttons btn_A, btn_B and btn_C and pressing btn_A and btn_C yields conflicting tasks t_A and t_C because of conflicting APs. Figure 2a denotes conventional event generation technique that is schedule-unaware: only some specific interleavings are explored. In our approach (Figure 2b), we attempt more schedules of t_A and t_C to manifest potential concurrency bugs. Particularly, if at least one of t_A and t_C is a background task, we attempt to make them run in parallel and schedule APs in random order. If t_A and t_C are atomic tasks, we try to manifest both $t_A \rightarrow t_C$ and $t_C \rightarrow t_A$.

IV. Hybrid Access Point Analysis

Concurrency bugs are related to the schedule of the conflicting tasks. Recall that two APs are conflict if they access the same resource and at least one of them is a write operation. Therefore, our goal is to obtain each event's relevant tasks and each task's relevant resource accesses. We employ both static analysis and dynamic profiling as hybrid analysis to find GUI event's relevant tasks and potentially conflicting APs in these tasks.

Static analysis obtains static data- and control-flow information and need not run target apps, so it does not depend on input events. We firstly build call graphs for all tasks with certain entry points which start tasks. Note that one task's call graph may be involved by another task's, because the latter has started the former. For each method in call graphs, we locate all appeared APs and further parse out conflicting ones.

However, the call graph may be incomplete due to implicit control flows that are difficult to precisely determine. For example, there are methods whose receivers can only be determined at invocation-time because of polymorphism.

Therefore, we complement the static analysis by a dynamic profiling phase to obtain more complete AP information. Particularly, we run the app with existing event generation technique that is not designed for detecting concurrency bugs and collect the execution traces via a modified Android Runtime (ART) virtual machine [12]. We record method and field events that are related to real resource accesses for tasks and produce conflicting APs information from them.

V. Automated Testing With Guided Event Generation

We automatically run the app under test with guided event generation, which is the second step in our approach. Our automated testing is based on state model where the state of a running app is defined as its layout, i.e., different layouts correspond to different states.

Algorithm 1 gives our scheduling oriented DFS (SO-DFS) algorithm. The algorithm consists of two parts: a depth-first search state-space exploration (event generation) and a pressure testing of conflicting tasks (schedule generation).

The event generation part resembles the standard DFS algorithm [13]. When calling `SO-DFS` with state s and event sequence π (Line 2), we firstly add s to S that stores explored states (Line 4) and parse out all enabled events of s in variable E (Line 5). Then we explore all its relevant events to reach more app states (Lines 12–19) and recursively call `SO-DFS` if the resulted state s' is unexplored (Lines 15–17). At last, we restore state s if necessary (Lines 18–19). State restoration is achieved by `restore`. It firstly tries to press the `BACK` key (Line 29). If this operation successfully restores state s, we continue our testing procedure. Otherwise, it restarts the app and replays the event sequence π to restore s (Lines 31–34). We cannot guarantee absolute state restoration, because the variables and database of app may be modified in the execution of the previously exercised sequence of events and it is a big challenge to restore them automatically and effectively.

The schedule generation part simultaneously generates concurrent tasks and their schedules at the same app state. At each app state s, we first obtain all pairs of events of s that can start conflicting tasks (Lines 7–9). For each pair of conflicting

Algorithm 1: The Scheduling Oriented DFS Algorithm

```
1  S ← φ; // explored states
2  function SO-DFS(s, π)
3      // s is current state, π is event sequence to reach s
4      S ← S ∪ {s};
5      E ← getEnabledEvent(s);
6      P ← φ; // conflicting event pairs
7      for each pair p = ⟨e1, e2⟩ in E × E do
8          if e1 potentially conflicts to e2 then
9              P ← P ∪ {⟨e1, e2⟩};
10     for each pair ⟨e1, e2⟩ in P do
11         sendConflictingPair(s,π,⟨e1, e2⟩);
12     for each event e in E do
13         sendEvent(e);
14         s' ← getCurrentState();
15         if s' ∉ S then
16             // "::" denotes list concatenation
17             SO-DFS(s',π :: ⟨e⟩);
18         if s' ≠ s then
19             restore(s,π);
20 function sendConflictingPair(s,π,⟨e1, e2⟩)
21     // send events e1 and e2 orderly
22     sendEventPair(e1,e2);
23     restore(s,π);
24     // send e and block background tasks started by e
25     bts1 ← sendEventAndBlock(e1);
26     bts2 ← sendEventAndBlock(e2);
27     // unblock background tasks bts
28     unblock(bts2);
29     unblock(bts1);
30     restore(s,π);
31 function restore(s,π)
32     pressBack();
33     s' ← getCurrentState();
34     if s ≠ s' then
35         restartApp();
36         for ⟨ei⟩ ∈ π do
37             sendEvent(ei);
```

events, we schedule them via `sendConflictingPair` (Lines 10–11).

Our goal is to schedule conflicting APs in different tasks to manifest bugs between conflicting tasks due to race conditions. Our scheduler yields the current thread when the thread is about to execute a conflicting AP (other tasks are thereby scheduled). We also adopt two heuristic rules to manifest more schedules of such tasks. Note that we do not take the causal dependency between tasks into consideration, which means that there may be such a situation that the second task is not enabled any longer after performing the first one.

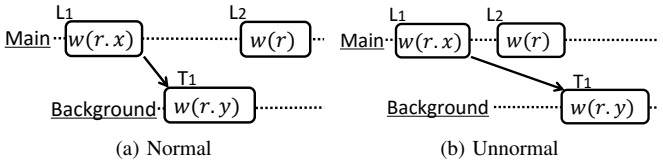


Fig. 3: Two different schedulings for tasks

The first method is aimed at conflicting tasks that run in Main Thread. When performing relevant conflicting events, we send them with different orders to attempt more schedules of those tasks. For example, there are two tuples of event, listener and background thread, i.e. $\langle e_1, (L_1, T_1)\rangle$ and $\langle e_2, (L_2, T_2)\rangle$, which means that the event triggers the listener and the listener starts the background thread. If there are some conflicting APs between (L_1, T_1) and (L_2, T_2), we would like to enumerate as many interleavings as possible between them to manifest. Because both L_1 and L_2 execute in Main Thread, they cannot interleave with each other arbitrarily and there are only two different schedules for them actually. If E_1 is sent earlier than E_2, L_1 certainly execute earlier than L_2 and vice versa. Therefore, we can send event E_1 to trigger L_1 firstly, but send event E_2 for L_2 firstly in the next attempt. This explains why conflicting event pair is sensitive to the order of events.

The second method is proposed for task combinations. We insert executions of corresponding conflicting tasks into the task combinations, which may break the atomicity of the latter to cause some unexpected outcomes. For example, there are a combination of listener and background thread (L_1, T_1) and another listener L_2 and they can be scheduled with different sequences as shown in Figure 3. It requires that variables `x` and `y` in `r` must be consistent. In Figure 3a, execution flow $(L_1 \rightarrow T_1 \rightarrow L_2)$ is a normal one. But in Figure 3b, L_2 writes `r` between the operations of L_1 and T_1, which may cause inconsistency of `x` and `y`. Therefore, if L_1 has started thread T_1, we may block T_1 instantly and send corresponding event to trigger L_2. After L_2 has finished, we then unblock T_1. At last, we may see some unexpected outputs because of atomicity violation. Particularly, we block the background tasks by function `sendEventAndBlock` and unblock them by function `unblock` in SO-DFS (Lines 23–26).

VI. Implementation

We implemented the AATT prototype whose architecture is presented in Figure 4. The `Anomaly analyzer` component integrates both static analysis and dynamic profiling, and the latter includes a subcomponent `Event collector` to collect event traces. This component receives an apk file as input, and generates potential conflicting APs and their corresponding conflicting tasks. The `Instrumentor` component inserts control statements into apk and produces the final apk file. The last component `Executor` automatically runs app under the control strategy described in Section V.

The static analysis is implemented based on Soot [14]. When we build the call graph for the task, the entry method

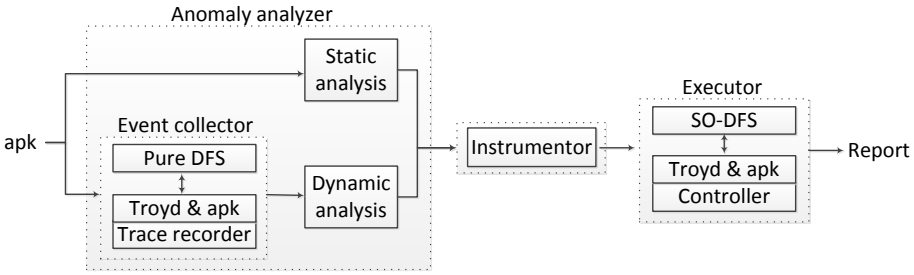


Fig. 4: The architecture of our system

for the `Listener` is the user defined callback method. For a `Thread` or an `AsyncTask`, its entry method is the starting method, i.e. `run` or `doInBackground`. After that, we determine APs based on the field access instructions. For other resources, we recognize APs by API invocations. We localize all APs of methods appeared in the call graph for each task. We analyze those APs to pinpoint the conflicting ones.

The dynamic profiling is realized by DFS that systematically explores the app's state space [13]. `Event collector` a part of dynamic profiler is implemented by modifying the interpreter of ART to collect field access and method invocation events. When we have collected enough event traces or we cannot continue to traverse the target app further, we stop this procedure to parse records. For field read/write events, we produce memory access information from them. For method events, there are three types, one entry and two exit events. We infer access information of other resources from these events. The rest of dynamic profiling is to generate conflicting APs.

The component `Instrumentor` instruments control statements into the apk file at the beginning of conflicting background tasks and the points before potential conflicting APs. We block the background tasks by inserting semaphore `P` operations at the beginnings of the background tasks. And then we can control their executions by our controller. For other potential conflicting APs, we insert system API invocations, such as `Thread.sleep` and `Thread.yield`, before them to reschedule conflicting tasks. The instrumented apk file is installed and tested in the next component.

In `Executor` which is built on our previous testing framework ATT [15], we do resignature for target apk file and implant a service Troyd in it. Troyd runs in the same process with the app and collects its runtime information under given commands. We then install resignatured apk file on the device. We unblock conflicting tasks described in SO-DFS by taking `V` operations on the corresponding semaphores. Besides, we sort all enabled candidate events for app state exploration by their priority, and an event has higher priority if it starts a conflicting task. The controller of automated testing runs on PC to guide the execution of the app with SO-DFS. The detail of the controller is described in Section V. We also record event sequences and schedulings of tasks to reproduce those concurrency bugs once observed. We define the state of a running app as its layout whose hash value is calculated based on all widgets with their coordinates, sizes and types.

When we test an app with guided event generation, it may also need some inputs, such as url, username and password. This is an open problem for automated testing of Android apps [30]. Therefore, we have prepared some inputs for each app under test if necessary. When the app requires specific inputs, AATT feeds it directly.

TABLE I: Evaluated subjects

App	Availability	LoC	Size (KB)
2buntu [16]	Github	963	767
aarddict [17]	Github	5,077	1,856
aNarXiv [18]	Github	3,357	62
andiodine [19]	Github	1,502	136
Down [20]	EOEAndroid	2,046	41
DroidWeight [21]	Google Code	5,078	170
externalIP [22]	Github	2,416	14
falling_blocks [23]	Google Code	1,763	124
GigaGet [8]	Github	3,123	1,161
HostIsDown [24]	Github	631	292
KindMind [25]	Github	5,510	301
LilyDroid [26]	Google Code	10,471	805
MultiPing [27]	Github	547	22

† The first column gives the names of subjects, second column is source code availability, and columns `LoC` and `size` presents the scale of subjects.

VII. Evaluation

In this section, we experimentally evaluate our approach by applying it to real-world open source apps.

A. Experimental Setup

We evaluated AATT using real-world open-source apps regarding the following questions: (1) (effectiveness) whether AATT can find concurrency bugs in Android apps that are both unaware by developers and hard to detect by conventional approaches; and (2) (efficiency) whether AATT consumes a reasonable amount of resources in testing mobile apps.

We downloaded thirty popular open-source apps from Github, Google Code, and EOEAndroid and run them with AATT. We chose nine subjects in which AATT finds concurrency bug plus four randomly chosen subjects for detailed effectiveness and efficiency evaluation. The experimental subjects are shown in Table I.

To evaluate the effectiveness, we compare the crash bug detection capability between AATT and (1) the industrial standard random testing tool Monkey [28]; and (2) a standard model-based depth-first search (DFS) [13]. Both AATT and DFS are run until completion, i.e., all reachable transitions are explored at least once. Besides, we have enhanced DFS strategy by trying to send events twice for each widget, which is a little different from DFS in `Event collector`. To maximize Monkey's probability of detecting concurrency bugs, we provide Monkey twice as much time as AATT during the testing procedure.

Furthermore, we manually inspect the program execution of AATT to figure out whether there is a functional bug to evaluate whether AATT is able to manifest hidden concurrency bugs in an app. Such bugs are reported to the app developers.

To evaluate the efficiency, we also collected profiling data of AATT: the testing time, the number of generated events, and number of APs extracted.

All experiments were conducted on a machine with Intel Core i5-4200U CPU and 4GB RAM running Ubuntu 12.04 LTS. Apps are tested on a Google Nexus 5 with Android 5.0.

B. Evaluation Results

Our effectiveness evaluation results are demonstrated in Table II. AATT discovered six concurrency bugs leading to app crash and three bugs leading to malfunctioning. By searching the repository and the issue tracking system, we believe that 7/9 were previously unknown. We submitted bug reports for active projects and 3 have already received confirmation.

Compared with Monkey, it only discovered one bugs (Down) which is also found by our approach. Compared with DFS, it also captured a bug (GigaGet) which is appeared in AATT's report (note that without our enhancement, DFS cannot detect this bug). These results show that AATT is promising in detecting concurrency bugs that are difficult to reveal by conventional techniques. Though Monkey can detect any bug in theory, the evaluation shows that our targeted approach is much more effective when testing resources are limited.

The efficiency evaluation results are shown in Columns `Time` and `Event` of Table II. For all evaluated buggy subjects, AATT hits the bug within 800 seconds and 100 events except for andiodine. If the subject does not fail, AATT spends slightly more time than the pure DFS. DFS detects the bug of GigaGet with slightly less resource than AATT. For the other apps, the total time and events when testing finish are also less than AATT. Besides, Monkey consumes twice as much time as AATT, and much more events if it does not capture any bug.

Finally, we also display the experimental results of the hybrid analysis in Table III. From the statistical information, we can see that both static analysis and dynamic profiling overlap some APs for most of apps except andiodine and DroidWeight. It also shows that both static and dynamic analyses contribute some APs to final analysis results (except DroidWeight) although their contributions have different performances according to subjects. We guess that this is because of the incomplete call graph in static analysis and low code coverage when running app in dynamic profiling, which have been discussed in Section IV. The results suggest that the hybrid analysis is efficient and static and dynamic analyses complements each other.

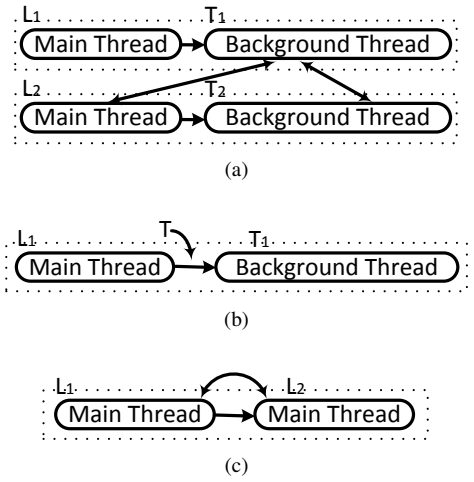


Fig. 5: Concurrency bug categories

C. Bug Categories

We further examined the concurrency bugs found by AATT and summarized out three categories about concurrency bugs from them. Figure 5 presents the schematic diagrams of concurrency bug categories in which two are about data race and one is about atomicity violation.

The first category figures out the situation between two threads when they read/write shared resources without synchronization, which is typical traditional race. As shown in Figure 5a, listener L_1 and background thread T_1 compose an execution flow while L_2 and T_2 make up another one. Here, T_1 may conflict to L_2 and T_2 when they access same resources and at least one of them is a write operation.

The second category presented in Figure 5b is about atomicity violation appeared in the combination of multiple threads. This happens when a task starts another one. They may access the global resource, which should be operated in an atomic way. For example, L_1 writes part of a global array `arr` and T_1 writes the rest of the array `arr` based on some complicated calculations. Obviously, these two operations should not be disturbed by other read/write operations. If another thread T_2 reads `arr` before T_1 writes, it just obtains an incomplete and useless array, and behaves unexpectedly.

The last category presented in Figure 5c is also related to data race. Because these bugs are appeared in a single thread, i.e. Main Thread, we also call them single-thread event-based races [3]. Although listeners of events are all executed in Main Thread, they can still interleave with each other as a result of the dispatching orders of events. However, this interleaving can incur races. For example, listener L_1 reads field `r` and listener

TABLE II: The Results of SO-DFS, DFS and Monkey

App	SO-DFS			DFS			Monkey		
	Time	Event	Type	Time	Event	Type	Time	Event	Type
2buntu	1,416	218	-	875	183	-	2,832	2,224,666	-
aarddict	158	14	F	1,252	230	-	316	234,944	-
aNarXiv	203	17	C	956	119	-	406	214,135	-
andiodine	2,247	510	C	2,258	459	-	4,494	3,226,130	-
Down	82	8	C	281	61	-	26	20,228	C
DroidWeight	6,235	1,100	-	2,320	461	-	12,470	9,223,372	-
externalIP	449	87	-	223	27	-	898	724,193	-
falling_blocks	215	23	F	158	24	-	430	356,255	-
GigaGet	655	97	C	271	51	C	1,310	1,086,235	-
HostIsDown	231	17	F	819	105	-	462	351,865	-
KindMind	2,640	329	-	1,478	322	-	5,280	3,774,124	-
LilyDroid	795	79	C	11,909	1,330	-	1,590	1,282,258	-
MultiPing	375	54	C	440	105	-	750	536,864	-

† Symbol "F" stands for "functional bug" and symbol "C" means "crash bug"

‡ "SO-DFS", "DFS" and "Monkey" present corresponding approaches. The "Time" means consumed time when it captures the first bug, or presents all time that they need if target app does not crashes. "Event" indicates the number of events for the first bug, or the total number of events if app does not crash. "Type" gives the type of bug, including functional bug and crash bug.

TABLE III: The APs' Numbers of Static Analysis and Dynamic Profiling

App	SAP	DAP	Unique	Both
2buntu	4	17	17	4 (23.53%)
aarddict	193	47	237	3 (1.27%)
aNarXiv	28	18	38	8 (20.05%)
andiodine	3	14	17	0 (0.00%)
Down	24	26	40	10 (25.00%)
DroidWeight	45	0	45	0 (0.00%)
externalIP	4	2	4	2 (50.00%)
falling_blocks	21	60	68	13 (19.12%)
GigaGet	4	63	63	4 (6.35%)
HostIsDown	8	2	8	2 (25.00%)
KindMind	4	10	11	3 (27.27%)
LilyDroid	168	41	192	17 (8.85%)
MultiPing	3	16	18	1 (5.56%)

† "SAP" presents APs' number of static analysis, while "DAP" means APs' number of dynamic profiling. "Unique" gives the total number of APs for static and dynamic analyses and "Both" shows the number of APs reported by both of them, as well as the percentage of "Both" in "Unique".

L_2 sets `r` to `null`. Normally, L_1 should not be invocated if L_2 has already been scheduled to execute. However, there are some special cases that may still trigger the invocation of L_1, thus causing app to crash.

For subjects in our evaluation, aarddict, aNarXiv, Down and falling_blocks belong to the first concurrency bug category. They all have conflicting resource accesses in background threads, such as a global connection of database. Both HostIsDown and LilyDroid read/write shared resources in listeners and background threads, which should be operated in an atomic way. So they are classified into the second category. For the rest buggy subjects, i.e. andiodine, GigaGet and MultiPing, some of their listeners can set a global reference to `null` before accesses of other listeners in some specific schedulings, thus leading to Null Pointer Exception (NPE). So we put them into the third category.

We think that most of these bugs are due to that developers are too dependent on concurrency mechanism in Android specification. It is just used to protect partial operations, such as UI operation, from races but cannot ensure concurrency bug free. There may still be some conflicting resource accesses of other types. Besides, because time-consuming tasks are required to put into background threads, many operations for resource accesses are taken in several threads. Therefore, developers should not only focus on traditional race and atomicity violation, but also watch out race in the single-thread and atomicity violation in the whole execution flows.

As Android apps are becoming increasingly complicated, developers should be of more care. Even worse, concurrency bugs are more difficulty to defect than ordinary ones. Therefore, we present these categories to help developers avoid some common concurrency bugs.

VIII. Related Work

With advances in mobile technology, the popularity of smartphone uses with ordinary people grows large. The quality properties of smartphone apps become even more important than before. Therefore, some researchers have proposed many novel approaches to finding out the hidden problems.

Some work focuses on general functional testing. DynoDroid [9] generates required events for target apps based on random exploration which is more efficient than Monkey. A^3E [6] implements a depth search based on the dynamic model of an app which considers each activity as an independent state. EvoDroid [29] combines an Android-specific program analysis technique and an evolutionary algorithm as a novel framework for automated testing. However, EvoDroid cannot systematically reason about input conditions. UGA [30] leverages human insights to improve traditional approaches' performance. However, these tools do not consider conflicting resource accesses and may miss concurrency bugs easily.

There is also some work aimed at data races in Android apps. Both DroidRacer [3] and CAFA [4] generate execution traces by systematically testing apps, and then computing the

happens-before relation on traces to detect races. A more thorough and precise happens-before model is presented in [5]. The authors also proposed a scalable algorithm to build and query in the happens-before graph of apps. But their performances heavily depend on the quality of input events that are used to build the happens-before graph. RacerDroid [31] attempts to manifest data races reported by an existing imprecise race detection tool, but it needs to modify the framework when scheduling events and threads. AsyncDroid [32] explores different thread interleaving by repeating given event sequences to detect thrown exceptions and assertion violations.

Some prior work also presents the techniques that construct concurrency test cases together with scheduling for traditional programs. RaceFuzzer [33] executes programs with a randomized thread scheduler which blocks threads at race points and randomly releases an available thread. Another fully automatic testing technique presented in [34] generates test cases which call methods on an instance of the class under test and execute code sequentially to check whether the instance behaves as expected.

Additionally, some researchers pay close attention to non-functional quality. PerfDroid [35] is a static analysis tool which summaries some performance bug patterns and detects them in Android apps. The automated test framework in [36] systematically generates test inputs that may lead to energy hotspots/bugs in Android apps. GreenDroid [37], [38] traverses apps' states as more as possible to find out states with low data utilization coefficient based on an *application execution model* derived from Android specification.

IX. Conclusion

In this paper, we studied Android apps' failures caused by conflicting resource accesses in tasks. We proposed a hybrid approach that adopts both of static analysis and dynamic profiling to find potentially conflicting APs. Our novel SO-DFS algorithm is based on event sequence generation that guides to run target app and schedules tasks with a heuristic strategy to manifest potential concurrency bugs. Our prototype tool AATT has successfully found out several previously unknown bugs in popular open source Android apps.

There are still two limitations in our approach. For example, we cannot manifest concurrency bugs whose conflicting events are appeared in different states. Another drawback is that our strategy for task scheduling is not deterministic and can only schedule tasks with a certain probability. In future, we plan to improve our AATT aimed at its limitations and validate it with more real-word apps.

Acknowledgment

This work was supported in part by National Basic Research 973 Program (Grant #2015CB352202), National Natural Science Foundation (Grants #61472174, #91318301, #61321491) 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. http://www.appbrain.com/stats/number-of-android-apps.
[2] Processes and threads. http://developer.android.com/guide/components/processes-and-threads.html.
[3] P. Maiya, A. Kanade, and R. Majumdar, "Race Detection for Android Applications," in *PLDI*, 2014.
[4] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, "Race Detection for Event-driven Mobile Applications," in *PLDI*, 2014.
[5] P. Bielik, V. Raychev, and M. Vechev, "Scalable Race Detection for Android Applications," in *OOPSLA*, 2015.
[6] T. Azim and I. Neamtiu, "Targeted and Depth-first Exploration for Systematic Testing of Android Apps," in *OOPSLA*, 2013.
[7] AsyncTask. https://developer.android.com/reference/android/os/AsyncTask.html.
[8] A multi-thread & lightweight downloader designed for Android. https://github.com/PaperAirplane-Dev-Team/GigaGet.
[9] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An Input Generation System for Android Apps," in *FSE*, 2013.
[10] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. Vechev, "Stateless Model Checking of Event-driven Applications," in *OOPSLA*, 2015.
[11] C. Hu and I. Neamtiu, "Automating GUI Testing for Android Applications," in *AST*, 2011.
[12] Android Runtime. https://source.android.com/devices/tech/dalvik/index.html.
[13] R. Tarjan, "DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*," *SICOMP*, 1972.
[14] A framework for analyzing and transforming Java and Android Applications. https://sable.github.io/soot/.
[15] Z. Meng, Y. Jiang, and C. Xu, "Facilitating Reusable and Scalable Automated Testing and Analysis for Android Apps," in *Internetware*, 2015.
[16] X. Li, Y. Jiang, Y. Liu, C. Xu, X. Ma, and J. Lu, "User Guided Automation for Testing Mobile Apps," in *APSEC*, 2014.
[17] 2buntu-android-app. https://github.com/2buntu/2buntu-android-app.
[18] Aard Dictionary for Android. https://github.com/aarddict/android.
[19] aNarXiv. https://github.com/nephoapp/anarxiv.
[20] iodine for Android. https://github.com/yvesf/andiodine.
[21] Down. http://www.eoeandroid.com/thread-311180-1-1.html.
[22] DroidWeight. https://code.google.com/archive/p/droidweight.
[23] Connectivity notification tool for Android. https://github.com/kost/external-ip.
[24] fallingblocks-android. https://code.google.com/archive/p/fallingblocks-android/.
[25] HostIsDown. https://gitlab.com/ilpianista/HostIsDown.
[26] KindMind. https://github.com/SunyataZero/KindMind.
[27] LilyDroid. http://qqgroup.googlecode.com/svn/trunk/LilyDroid/.
[28] MultiPing for Android. https://github.com/softgearko/MultiPing-for-Android.
[29] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.
[30] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: Segmented Evolutionary Testing of Android Apps," in *FSE*, 2014.
[31] H. Tang, G. Wu, J. Wei, and H. Zhong, "Generating Test Cases to Expose Concurrency Bugs in Android Applications," in *ASE*, 2016.
[32] B. K. Ozkan, M. Emmi, and S. Tasiran, "Systematic Asynchrony Bug Exploration for Android Apps," in *CAV*, 2015.
[33] K. Sen, "Race Directed Random Testing of Concurrent Programs," in *PLDI*, 2008.
[34] M. Pradel and T. R. Gross, "Fully Automatic and Precise Detection of Thread Safety Violations," in *PLDI*, 2012.
[35] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and Detecting Performance Bugs for Smartphone Applications," in *ICSE*, 2014.
[36] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting Energy Bugs and Hotspots in Mobile Apps," in *FSE*, 2014.
[37] Y. Liu, C. Xu, and S. Cheung, "Where Has My Battery Gone? Finding Sensor Related Energy Black Holes in Smartphone Applications," in *PerCom*, 2013.
[38] Y. Liu, C. Xu, S. Cheung, and J. Lu, "GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications," *IEEE TSE*, 2014.