Efficiently Manifesting Asynchronous Programming Errors in Android Apps

Lingling Fan
East China Normal University, China

Ting Su*
East China Normal University, China
Nanyang Technological University, Singapore

Sen Chen
East China Normal University, China

Guozhu Meng
Chinese Academy of Sciences, China
Nanyang Technological University, Singapore

Yang Liu
Nanyang Technological University, Singapore

Lihua Xu
New York University Shanghai, China

Geguang Pu
East China Normal University, China

ABSTRACT

Android, the #1 mobile app framework, enforces the *single-GUI-thread* model, in which a single UI thread manages GUI rendering and event dispatching. Due to this model, it is vital to avoid blocking the UI thread for responsiveness. One common practice is to offload long-running tasks into async threads. To achieve this, Android provides various async programming constructs, and leaves developers themselves to obey the rules implied by the model. However, as our study reveals, more than 25% apps violate these rules and introduce hard-to-detect, fail-stop errors, which we term as aysnc programming errors (APEs). To this end, this paper introduces APEChecker, a technique to automatically and efficiently manifest APEs. The key idea is to characterize APEs as specific fault patterns, and synergistically combine static analysis and dynamic UI exploration to detect and verify such errors. Among the 40 real-world Android apps, APEChecker unveils and processes 61 APEs, of which 51 are confirmed (83.6% hit rate). Specifically, APEChecker detects 3X more APEs than the state-of-art testing tools (Monkey, Sapienz and Stoat), and reduces testing time from half an hour to a few minutes. On a specific type of APEs, APEChecker confirms 5X more errors than the data race detection tool, EventRacer, with very few false alarms.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Asynchronous programming error, testing, static analysis, Android

ACM Reference Format:
Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu. 2018. Efficiently Manifesting Asynchronous Programming Errors in Android Apps. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238170

1 INTRODUCTION

Most modern GUI frameworks such as Swing [48], SWT [79], Android [25], Qt [68], WxErlang [84], and MacOS Cocoa [7] enforce the *single-GUI-thread model*, in which one single UI thread instantiates GUI components and dispatches events. Specifically, UI thread fetches system or user events off an event queue, and dispatches them either to a responsible app component's handler or to a UI widget's event handler. These handlers run on the UI thread and exclusively update GUIs if necessary.

Due to this single-GUI-thread model, it is vital to avoid blocking the UI thread. Therefore, most GUI frameworks recommend to offload intensive tasks (*e.g.*, network access, database queries) to async threads (*i.e.*, background threads). Take Android development framework (ADF) as an example, it provides many async programming constructs (*e.g.*, AysncTask, Thread, AsyncTaskLoader, IntentService) to achieve this goal.

Like other frameworks, ADF leaves developers themselves to properly handle the interactions between these async threads and the UI thread — obey the rules implied by the single-UI-thread model. However, our investigation on 930 apps that use async constructs shows, more than 25% apps violate these rules, and introduce *fail-stop* bugs. For example, if a worker thread directly updates the text displayed on the UI thread, the app will crash. Another example is, when an async thread finishes its background task, and tries to send a UI update event to a GUI component. Before the update takes effect, if the user rotates the screen, the UI thread will destroy and recreate that GUI component. By default, the update event is routed to the destroyed GUI rather than the newly created one, which may crash the app. In this paper, we term

*Ting Su is the corresponding author of this paper.
Emails: ecnujanefan@gmail.com, tsuletgo@gmail.com, ecnuchensen@gmail.com, gzmeng@ntu.edu.sg, yangliu@ntu.edu.sg, lihua.xu@nyu.edu, ggpu@sei.ecnu.edu.cn

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238170

such fatal programming errors that violate the rules implied by the single-UI-thread model as *async programming errors* (APEs).

Such bugs in Android are *not easy to detect manually*, due to (1) they usually reside in the code of handling interactions between UI thread and async threads, which can be rather complicated for manual analysis; (2) they can only be triggered at the right states of GUI components (*e.g.*, activity, fragment) with complicated lifecycle [21, 30]; (3) they have to be triggered at right thread scheduling, while the execution time of async threads is affected by the task and its running environment (*e.g.*, network stability, system load).

Even worse, *existing bug detection techniques are ineffective for such bugs*. First, most GUI testing techniques, *e.g.*, random testing [39, 57], search-based testing [58, 60], and model-based testing [2, 3, 8, 78, 85], are designed for functional testing in general. They aim at enumerating all possible event sequences (GUI-level events in particular) to manifest bugs, which is unscalable and time-consuming. Additionally, they mainly aim at improving code coverage, which may not be sufficient for exhibiting APEs — require specific event sequences with appropriate lifecycle states and thread scheduling. Second, static analysis tools, *e.g.*, Lint [35], FindBugs [19] and PMD [67], although scalable, only enforce simple rules (syntax or trivial control/data-flow analysis) to locate suspicious bugs. For example, Lint declares it can find "WrongThread" errors (one type of APEs) [24]. However, as our evaluation in Section 5 demonstrates, Lint incurs a number of false negatives — failing to detect those sophisticated "WrongThread" errors as well as other types of APEs. Third, other fault detection techniques [10, 45, 59, 88] (*e.g.*, data race detection) have only tackled parts of APEs. To systematically tackle APEs, we conducted a formative study on 2097 Android apps to understand them. First, we find the async constructs are indeed widely used in 48.6% apps, and AysncTasks and Threads account for the majority. Second, we identified 3 async programming rules (see Section 2.2) implied by the single-GUI-thread model by analyzing Android docs, technical posts and previous fault studies on async programming. Third, from 1019 apps that use async constructs, we found that developers do violate these rules and introduce APEs. We collected 375 real APEs, involving 9 exception types (thrown from apps), *e.g.*, CalledFromWrongThread, IllegalStateException, BadTokenException, *etc.*

Informed by the above results, we develop an approach APEChecker. It first characterizes 3 fault patterns from 375 issues based on the 3 rules, and synergistically combines static analysis and dynamic UI exploration to efficiently manifest APEs. Specifically, it encodes the fault patterns into a static analyzer, locates suspicious APEs in the app code, generates a set of program paths that can reach the faulty code, maps program traces to real event sequences with appropriate environment, and finally verify these errors on the app.

We evaluate APEChecker on a set of 40 real-world Android apps, and compare it with three state-of-the-art GUI testing tools (Monkey [39], Sapienz [60], and Stoat [78]), and two fault detection tools (Lint [35] and EventRacer [10]). The results show (1) APEChecker unveils and successfully processes 61 APEs, of which 51 can be reproduced (83.6% hit rate) with real tests; (2) APEChecker detects 3X more APEs than the testing tools, and reduces detection time from half an hour to a few minutes; and (3) within comparable analysis time, APEChecker detects 5X more APEs than EventRacer with very few false positives.

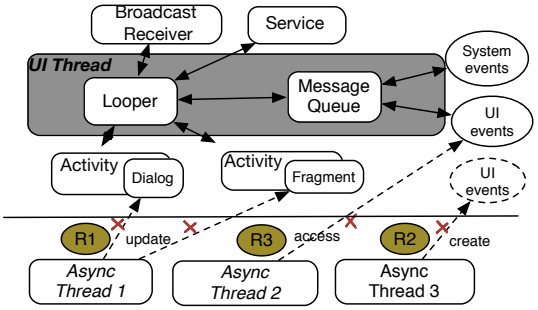

Figure 1: Single-GUI-thread model of Android and its three basic rules.

To summarize, this paper makes the following contributions:

- We conduct a formative study on 2097 Android apps to investigate APEs, and identify three async programming rules implied by the single-GUI-thread model.
- We develop APEChecker, a technique that synergistically combines static analysis and dynamic UI exploration, to efficiently detect and verify APEs.
- We evaluate APEChecker on 40 real-world apps, and clearly demonstrate its effectiveness over the state-of-the-art testing and other fault detection techniques on APEs.

2 ASYNC PROGRAMMING ERRORS

2.1 Async Programming in Android

Fig. 1 depicts the single-GUI-thread model of Android. The UI thread maintains a MessageQueue, and its Handler enqueues system or UI events into this queue. These events come from app components (*e.g.*, Broadcast Receivers, Services) or GUIs (*e.g.*, Activity and its associated visible components like Dialogs or Fragments). The UI thread's Looper dequeues events in a sequential order and dispatches them to the Handler for processing. The UI thread invokes the corresponding event handler *w.r.t.* an event and updates GUIs if necessary.

ADF provides various async programming constructs [23]. There are four typical constructs, *i.e.*, AysncTask, Thread, AsyncTaskLoader and IntentService. Among them, AsyncTask allows one to perform short background operations and publish results on the UI thread; Thread is inherited from Java, and executes tasks in the background; AsyncTaskLoader utilizes AysncTask to perform async data loading, and has similar callbacks as AsyncTask, but it is lifecyle aware: ADF binds/unbinds the worker thread according to GUI's lifecyle. IntentService handles async requests in an async thread, and sends the results to the UI thread vis a Broadcast Receiver. Fig. 2 illustrates the use of AysncTasks in *ADSdroid* [15], it starts two async threads, *i.e.*, SearchByPartName (Lines 9-22) and DownloadDatasheet (Lines 34-49) to search electronic components' datasheet, and download from a remote server if requested. The activity SearchPanel (Lines 2-23) searches for the result with user input in the doInBackground, showing a progress dialog in onPreExecute before searching, and dismisses it via onPostExecute. The results are shown in a ListView of the activity PartList (Lines 25-50), in which users can click any matched item (Lines 28-32) for downloading.

However, there are two APEs (Lines 17 and 46), neither of which has been covered by developers. The root causes for these two APEs are similar. When users rotate the screen right after the start of the

```
1  // the acitivty that shows the search panel
2  public class SearchPanel extends Activity{
3   ProgressDialog mSearchDialog;
4   public void searchByPartName(View view) {
5    mSearchDialog = new ProgressDialog(this);
6    new SearchByPartName(searchMode, partName).execute();
7   }
8   // AsyncTask to search parts
9   private class SearchByPartName extends AsyncTask<...> {
10   protected ArrayList<Part> doInBackground(Void...) {
11    ... // other stuffs
12    return Search.searchByPartName(partName, mode);
13   }
14   protected void onPostExecute(ArrayList<Part> result) {
15    super.onPostExecute(result);
16    //mSearchDialog is shown in onPreExecute()
17    mSearchDialog.dismiss();
18    if (result != null) {
19     Intent intent = new Intent(SearchPanel.this,
           PartList.class);
20     intent.putExtra(PartList.PARTS, result);
21     startActivity(intent);
22    }} ...
23 }
24 // the activity that shows the list of matched Parts
25 public class PartList extends ListActivity{
26  ProgressDialog mDownloadDialog;
27  // the event handler for clicking list item
28  protected void onListItemClick(...) {
29   Part selectedPart = new Part(...);
30   mDownloadDialog = new ProgressDialog(this);
31   new DownloadDatasheet().execute(selectedPart);
32  }
33  // AsyncTask to download the data sheet
34  private class DownloadDatasheet extends AsyncTask<...>{
35   protected String doInBackground(Part... parts) {
36    Part selectedPart = parts[0];
37    String fileName = fileNameForPart(selectedPart);
38    URLConnection pdfConnection = selectedPart.
          getPdfConnection();
39    pdfConnection.connect();
40    ... // fetch data from network and update progress
41    return fileName;
42   }
43   protected void onPostExecute(String result) {
44    super.onPostExecute(result);
45    // mDownloadDialog is shown in onPreExecute()
46    mDownloadDialog.dismiss();
47    if (result != null) {
48     openPDF(result);
49    }
50 }}}
```

Figure 2: Motivating Example

aysnc tasks SearchByPartName and DownloadDatasheet but before they finish (*i.e.*, before the execution of onPostExecutes), the app will crash when mSearchDialog (Line 17) and mDownloadDialog (Line 46) are dismissed. Because the rotation will destroy the current activity and create a new one, but the dialogs were attached to the original one, which does not exist anymore. This leads to a fatal BadTokenException. We can see the right timing and lifecycle states are crucial to manifest these APEs. This paper aims at tackling such hard-to-detect errors.

2.2 Formative Study

To understand APEs, we conducted a formative study to investigate the following questions. This enables our problem definition in Section 2.3 and fault pattern analysis in Section 3.1.

Q1: Are async constructs widely-used by Android developers to follow the single-GUI-thread model? To answer this, we focus on the four typical constructs introduced in Section 2.1, and investigate 2097 open-source Android apps from F-droid to observe the use of these constructs since F-droid is one of largest Android app repositories, which covers diverse application categories; and all apps are open-source and maintained on Github, Google Code, *etc.* We use Soot [81], a static analysis tool, to identify aysnc constructs.

Answer: Async constructs are widely-used by Android developers. We found 1019 out of 2097 apps use async constructs (account for 48.6%). Among 1019 apps, 2968 AsyncTasks, 1248 Threads, 286 IntentServices, and 35 AsyncTaskLoaders are used. AsyncTask and Thread account for the majority, which also conforms to prior work [53, 54].

Q2: Are there any basic rules implied by the single-GUI-thread model that Android developers should obey in async programming? Are they common to other GUI frameworks? To answer this, we conducted a thorough and careful inspection on (1) Android docs and APIs [23], including the principle of single-GUI-thread model [40], various async programming constructs [27, 32–34, 37, 38, 41, 42], GUI components [21, 28, 30, 36, 43], *etc*; and (2) technical posts filtered from Stack Overflow (the largest developer Q&A community) by the keywords "Android" plus the names of async constructs, tutorials on async programming [13]; and (3) fault studies on Android [18, 46, 55, 86].

Answer: We identified 3 async programming rules (Fig. 1 annotated these rules as *R1*, *R2* and *R3*), which are also common to other modern GUI frameworks.

- ***Rule 1 (Async threads should not update GUI objects)***: Since Android UI toolkit is not thread-safe, the single-GUI-thread model requires that async threads should not directly manipulate GUI objects. GUI objects include *visible components* (*e.g.*, update a Dialog's message) and *data models* (*e.g.*, change the content of ArrayAdapter that fits in between a ListView and an ArrayList). Instead, they should designate the UI thread to handle GUI objects via UI-safe methods like Activity#runOnUiThread [40]. This rule is also enforced by many GUI frameworks, *e.g.*, Swing, SWT, Qt, and Cocoa.
- ***Rule 2 (Async threads should not create GUI components in the background)***: The UI thread by default is created with a Handler and a Looper. The Handler enqueues events (*e.g.*, messages or runnable objects) into a message queue. These events come from different app components and GUIs. The Looper dequeues and dispatches the events to the Handler for processing. However, an async thread (except HandlerThread) by default is not associated with a Handler, thereby it should not directly create GUI components (*e.g.*, Toast#show, Dialog#create) in the background. Instead, they should post GUI creations via UI thread's Handler into Looper for processing. Qt's event loop and wxErlang's mailbox queue enforce this similar rule.
- ***Rule 3 (Async threads should avoid accessing GUIs or performing transactions inside async callbacks)***: Async callbacks such as onPostExecute run on the UI thread, but they have no knowledge of the current states of GUIs. Because they are called when the async thread returns. As a result, accessing GUIs or performing transactions for Fragments in async callbacks (*e.g.*, AysncTask#onPostExecute, LoaderCallbacks#onLoadFinished) has the risks of sending GUI updates to destroyed GUIs and losing app state [31]. Cocoa Touch (GUI framework of iOS) also enforces this similar rule [6].

Q3: Do developers violate these rules? Are there any challenges to solve such APEs? To answer this, we utilize Github and Google Code APIs to scrawl the issue reports of 1091 apps that uses async constructs in ***Q1***. To identify APEs, we only collect issues that are reported with exception traces, which contain the callbacks of async constructs; then inspect their issue descriptions, comments, patches if available to confirm valid issues that do violate the three rules.

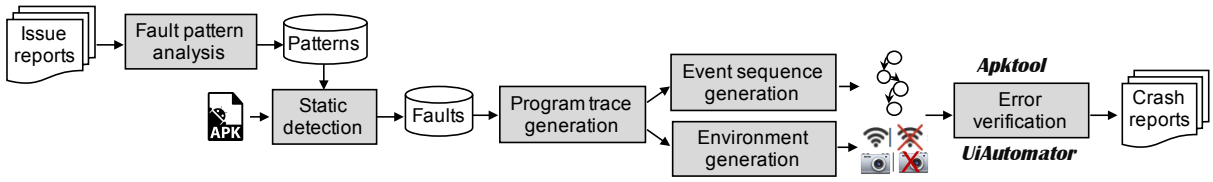

Figure 3: Workflow of APEChecker

Answer: We finally got 375 valid APE issues. The number of valid issues is not large since many issues are reported without exception traces. It is reported only 16% Github issues have exception traces [18]. But our evaluation (Section 5) reveals APEs affect more than 25% apps. These issues violate the rules in different forms, involving 9 different types of fatal exceptions, *e.g.*, CalledFromWrongThread, IllegalStateException, and BadTokenException. The violations of *Rule 1* and *3* account for the majority in these 375 issues (91.2 %). By investigating the fixing process of these APEs on Github, we note developers face three main challenges: (1) Due to lack of adequate understanding, they usually use simple try-catches to fix APEs; (2) Many APEs reside in the releases since developers fail to discover them during development; (3) Due to the lack of reproducing tests, several developers complain about the difficulties of debugging.

2.3 Problem Definition

We name the errors that violate the three basic rules as *aysnc programming errors*, and formulate our problem as follows.

UI thread. A UI thread u is the main thread that is created when an app starts, and manages GUI components.

Async thread. An async thread w is a worker thread that is instantiated from an async construct (*e.g.*, Thread), started by the UI thread (*e.g.*, Thread#run), and executes a task.

UI-accessing and UI-safe methods. A UI-accessing method (i.e., $m_{uiaccess}$) may create a GUI component (*e.g.*, Toast#show), change a GUI component's state (*e.g.*, Dialog#dismiss), or commit fragment transactions. A UI-safe method m_{uisafe} permits safe GUI access, *e.g.*, Activity#runOnUiThread, Handler#post, or GUI state check, *e.g.*, Activity#isFinishing().

Program trace and its event sequence. A program trace t is a sequence of method calls, $m_1, \ldots, m_i, \ldots, m_n$, on the call graph of an app. The event sequence s *w.r.t.* t is a set of user events, $e_1, \ldots, e_i, \ldots, e_k$ ($k \leq s$ usually holds), that can execute out t under the given environment E (*e.g.*, thread scheduling, network status, app permissions, sensor inputs and platform versions).

Problem Definition. Our problem is to check whether there exists any trace t on which u creates an async thread w, and w invokes any UI-accessing method that is *not control-dependent* on a UI-safe method. If t exists, we find an event sequence l that follows t to exhibit the error under given environment E.

2.4 Prior Work

Prior work has only tackled parts of this problem. One close work is from Zhang *et al.* [88], which finds invalid thread access errors in Java GUI applications (Swing, SWT, Android), and gives warnings in the form of method call chains. However, this work has several significant differences from ours: First, they only handle a subset of errors *w.r.t. Rule 1*, *i.e.*, an async thread directly accesses GUI components. Second, they use static analysis to check whether a thread spawning can reach a GUI object accessing method, which is determined by the ViewRoot#checkThread method in ADF. This is ad hoc and may miss many errors. Third, they do not confirm errors, which may bring many false positives. In contrast, we conduct a systematic study to understand APEs, and combine static and dynamic analysis to confirm errors with real tests. Lin *et al.* [54] investigate the uses of AsyncTask, and observe the invalid thread access errors. But they aim to solve another different problem of automatically refactoring long-running tasks in UI thread into AysncTasks.

Another area of related work is detecting and reproducing concurrency bugs (data race in particular) [10, 45, 52, 54, 59, 66, 80]. They treat some APEs *w.r.t. Rule 3* as data races. For example, the two APEs in Fig. 2 can be interpreted as data races since the mPanels variable inside the framework class PhoneWindow can be accessed simultaneously by the UI and async thread in which one access is a write operation. Existing data race detection tools [10, 45, 59] exploit dynamically explored event traces to build a happen-before graph, and then query the graph to find potential data races. But they have several limitations in practice to detect APEs: First, their effectiveness heavily relies on the traces for building the graph. If the traces have not fully covered app code, the detection ability is limited. Second, they usually generate a large number of false positives due to conservative analysis [47]. Existing data race reproducing tools [66, 80] are also impractical since they either require user-provided traces or depend on the results of data race detection tools when confirming bugs. Section 5 compares APEChecker with existing data race detection tools to confirm these observations.

3 OUR APPROACH APECHECKER

This section details our approach APEChecker. Figure 3 shows its workflow, which is composed of five key steps: APEChecker (1) summarizes a set of fault patterns from the collected APE issues, (2) encodes these fault patterns into a static analyzer to locate faulty code, (3) generates a set of program traces that can reach the faulty code from the app entry, (4) maps the program traces into real event sequences (tests) with appropriate environment; and (5) verifies APEs with the tests, and dumps crash reports for fixing.

3.1 Fault Pattern Analysis

We categorize 375 APE issues into three groups *w.r.t.* the rules summarized in the formative study. By analyzing these issues, we characterize these APEs as three fault patterns.

• ***Fault Pattern 1:*** If an async thread w is started, and w calls a UI-access method $m_{uiaccess}$, which is not control-dependent on a UI-safe method m_{uisafe}. This pattern violates ***Rule 1***, which is represented as $start(w) \rightarrow \neg\, m_{uisafe} \rightarrow m_{uiaccess}$. Fig. 4 shows such an error of *Pedometer* [17], an app that stores the user's step count per hour (the symbol "+" denotes the corresponding patch

```
1  // privacy-friendly-pedometer. Revision: 9a031d6
2  public class MonthlyReportFragment extends Fragment{
3    private void generateReports(boolean updated){
4    AsyncTask.execute(new Runnable(){
5       public void run(){
6         if (mAdapter != null && …){
7  +        getActivity().runOnUiThread(new Runnable(){
8  +           public void run(){
9                …
10               mAdapter.notifyDataSetChanged();
11 +          }
12 }}}}};}}
```

Figure 4: Example of Fault Pattern 1

```
1  // android_gisapp. Revision: 2ef12a7
2    // new ExportTask(…).execute();
3    public static class ExportTask extends AsyncTask<…>{
4     protected File doInBackground(Void... voids){
5      Cursor cursor = query(…);
6      if(cursor == null && …){
7  -     Toast.makeText(msg).show();
8  +     publishProgress();
9     }}
10 +  protected void onProgressUpdate(Void... values){
11 +    super.onProgressUpdate(values);
12 +    Toast.makeText(mLayer.getContext()).show(); }
13 }
```

Figure 5: Example of Fault Pattern 2

to fix this error). In fragment MonthlyReportFragment, it starts an aysnc thread to generate the monthly report and refresh the GUI by invoking notifyDataSetChanged, which crashes the app.

• ***Fault Pattern 2:*** If an async thread w is started, and w calls a GUI creation method $m_{uicreate}$, which is not posted on the UI thread by $m_{postlooper}$ to execute. This pattern violates ***Rule 2***, which is represented as $start(w) \rightarrow \neg\, m_{postlooper} \rightarrow m_{uicreate}$. Fig. 5 shows such an error of *gisapp* [16], which is a user interface controls library for Android geo applications (the symbols "+" and "-" denote the corresponding patch to fix this error). *gisapp* uses an async thread ExportTask to export the data by retrieving the database, and use Toast#makeText to show a message if no data is available. However, ExportTask has not referred to the UI thread's Handler to post messages, which crashes the app.

• ***Fault Pattern 3:*** If an async thread w is started, and before w returns, the target activity or fragment A is destroyed or stopped, and after w returns, its async callback calls a UI-access method $m_{uiacess}$, which is not control-dependent on a UI-safe method m_{uisafe}. This pattern violates ***Rule 3***, which is represented as $start(w) \rightarrow destroy(A)$ or $stop(A) \rightarrow return(w) \rightarrow \neg\, m_{uisafe} \rightarrow m_{uiaccess}$. In Fig. 2, there are two such errors. When users rotate the screen right after the start of the AsyncTasks SearchByPartName and DownloadDatasheet but before they finish, the dismiss of mSearchDialog (Line 17) and mDownloadDialog (Line 46) can crash the app.

3.2 Static Fault Detection

Algorithm 1 details static fault detection. It takes as input an app, the lists of UI-accessing and UI-safe methods (summarized from Android docs), and outputs the APE locations (including the starting points of async threads). Algorithm 1 works on the static call graph of an app, where each node denotes a method, and each edge $\langle f, g\rangle$ denotes the call from the method f to g. We first identify all method nodes that start async threads (Line 3) and check whether the use of corresponding async constructs violate the rules (Lines

Algorithm 1: Static Fault Detection

Input: apk: an Android app, $uiAccessAPIs$: list of UI-access methods, $uiSafeAPIs$: list of UI-safe methods
Output: $APEs$: List<methodStartThread, stmtStartThread, methodAccessUI, stmtAccessUI>

```
1  APEs ← ∅
2  cg ← buildCallGraph(apk)
3  asyncStartNodes ← getReachableAsyncStarts(cg)
4  foreach Node node ∈ asyncStartNodes do
5      async ← getAasyncClass(node)
6      o ← getOverriddenMethods(async)
7      nodeList ← getCalledMethodNodes(o, cg)
8      while nodeList ≠ ∅ do
9          node ← nodeList.dequeue()
10         if isVisited(node) then
11             continue
12         if node ∈ uiAccessAPIs then
13             if ¬ controlDepdent(node, uiSafeAPIs) then
14                 m ← getCallingMethod(async)
15                 n ← getCallingMethod(node)
16                 APEs ← ⟨m, async, n, node⟩
17         nodeList.enqueue(getCalledMethodNodes(node, cg))
18 return APEs
```

4-17). For each aysnc construct, we get the overridden methods according to its class type (Lines 5-6). Take AsyncTask as an example, we consider its callback methods onPreExecute, doInBackground and onPostExecute, *etc.* By querying the call graph, we gets all the nodes called by these methods (Line 7). If a UI-accessing method node is not control-dependent on any UI-safe methods (*e.g.*, Activity#runOnUiThread), this node will be tagged as suspicious APE (Lines 12-16). Note that we conduct both intra- and inter-procedural control-dependent analysis to reduce false alarms. Algorithm 1 also checks the successor methods that are called by this node (Line 17) to avoid false negatives.

Example. In Fig. 2, APEChecker identifies that searchByPartName starts an AsyncTask, SearchByPartName. In the onPostExecute callback, it locates a UI-accessing method ProgressDialog#dismiss (Line 17), which is not control-dependent on any UI-safe methods, *e.g.*, Activity#isFinishing(), to safely check activity state. So APEChecker reports a suspicious APE.

3.3 Program Trace Generation

Algorithm 2 details the program trace generation (the idea is similar to backward symbolic execution [56, 77]). It takes as input a target method and statement (*i.e.*, methodAccessUI and stmtAccessUI from Algorithm 1), and returns the traces (*i.e.*, method call sequences) that can reach the APE. It starts with an empty trace t, and backtracks to reach the entry activity (Lines 5-30). Specifically, the maximum number of generated traces (MaxTraceCnt, Line 17) and the maximum trace length (MaxTraceLen, Line 5) is configurable.

During the analysis, each trace t may have one of the three states, *i.e.*, pending (t is under propagation), terminated (t reaches the entry activity), failed (the propagation fails due to dead code or limitations of static analysis tools). If no traces are pending, the analysis stops (Lines 6-7). Otherwise, the analysis continues until the maximum trace length is reached. At each iteration, the algorithm uses two important variables, *i.e.*, ptrMethod and ptrStmt, pointing to the current backtrack point of t. When t is pending, the

algorithm queries the callers of t via getAcyclicCallers (we details this function later). If there are no callers, t will be set as *failed* due to the propagation cannot proceed (Lines 12-14). Otherwise, we update ptrMethod and ptrStmt of t (Lines 16-26). Specifically, if t's ptrMethod has multiple callers, the algorithm will fork out a new trace t' from t (copying all previous backtrack information from t to t'), and add t' into the list of traces (Lines 16-22).

Here, updateTrace (Line 21 and 25) performs three operations: (1) check whether t's ptrMethod has reached the entry activity and update t's state accordingly. In particular, if ptrMethod is a callback method (*e.g.*, user event handler) of the entry activity, t will be set as terminated. (2) update the method call chain of t by adding ptrMethod. The method call chain will be later used to generate event sequences. (3) record the conditions in the body of ptrMethod that ptrStmt are control-dependent on. These conditions will be analyzed later to create necessary environment to improve the hit rate. Section 3.4 will discuss the details of (2) and (3).

The function getAcyclicCallers queries and returns the immediate callers of t's ptrMethod. It removes the visited callers to ensure the traces are acyclic. When backtrack, we differentiate three types of call relations that widely exist in Android.

• ***Explicit Calls***. The function f immediately calls function g, *i.e.*, there exists an edge $\langle f, g\rangle$ on the call graph. For example, SearchByPartName#doInbackground is explicitly called by *searchByPartName* via AsyncTask#execute (see Fig. 2).

• ***Implicit Calls***. Android systems have many implicit calls through its framework. An implicit call from function f to g indicates f calls g asynchronously. For example, the callback onPostExecute of SearchByPartName is implicitly called when the callback doInbackground returns (see Fig. 2).

• ***Inter-Component Transition Calls***. Android system uses Intents to start activities or services. For example, in SearchByPartName, the activity *PartList* is called by the callback onPostExecute via startActivity (see Fig. 2).

Example. We explain Algorithm 2 on *ADSdroid* (Fig. 2). Algorithm 1 reports two suspicious faults, one of which is in the onPostExecute callback, which dismisses mDownloadDialog without any UI-safe methods (Line 46). Algorithm 2 first initializes a pending trace, whose ptrMethod is set as onPostExecute and ptrStmt as the dismiss statement. It then finds doInBackground is an implicit caller of onPostExecute (see ⑤). Next, it finds onListItemClick explicitly calls doInBackground via execute (see ④). Next, it finds the activity PartList is called by SearchByPartName#onPostExecute (see ③), and update ptrMethod as onPostExecute and ptrStmt as the startActivity statement (Line 21). Similarly, it finally finds the method searchByPartName starts the SearchByPartName (see ①). The final trace is searchByPartName → onListItemClick (only user event handlers are shown).

Algorithm 2: Program Trace Generation

Input: $method$: the target method, $stmt$: the target statement
Output: $traces$: the list of candidate traces

```
1  let t be an initially empty trace
2  let traceLen be the current trace length (initialized as 0)
3  t.ptrMethod ← method, t.ptrStmt ← stmt
4  traces ← {t}
5  while traceLen < MaxTraceLen do
6      if ¬ hasPendingTrace(traces) then
7          return traces
8      new_traces ← {}
9      foreach Trace t ∈ traces do
10         if isPending(t) then
               // t is a pending trace
11             t.callers ← getAcyclicCallers(t.ptrMethod)
12             if t.callers == ∅ then
                   // set t's state as failed
13                 updateTrace(t)
14                 continue
15             let c be the first caller of t.callers
16             foreach Caller c' ∈ t.callers\{c} do
17                 if count(traces) < MaxTraceCnt then
18                     t' ← fork(t)
19                     t'.ptrMethod ← getMethod(c')
20                     t'.ptrStmt ← getCallsite(c')
21                     updateTrace(t')
22                     new_traces ← new_traces ∪ {t'}
23             t.ptrMethod ← getMethod(c)
24             t.ptrStmt ← getCallsite(c)
25             updateTrace(t)
26             new_traces ← new_traces ∪ {t}
27         else
               // t is a terminated or failed trace
28             new_traces ← new_traces ∪ {t}
29     traces ← new_traces
30     traceLen ← traceLen + 1
```

3.4 Event Sequence and Environment Generation

This step converts the program traces from Section 3.3 into actionable event sequences with appropriate environment.

Event Sequence Generation. APEChecker considers two main types of callbacks when converting a program trace.

• ***User event handler callbacks.*** APEChecker maps each event handler on the trace to a corresponding action *w.r.t.* a UI widget or view. For example, in Fig. 2, the event handler onListItemClick(...) is mapped to a click action on a ListView item; searchByPartName (declared in the XML layout file) is mapped to a click action on the "Search" button. To achieve this, APEChecker maintains a view-handler mapping table, which supports event handler callbacks registered in both app code and XML layouts [70, 75].

• ***Activity/Fragment lifecycle callbacks.*** For an activity or fragment lifecycle callbacks (*e.g.*, onRestart, onResume, onDestroy) on the trace, APEChecker automatically substitutes the callback with special events that can force app to execute it. For example, for onRestart, APEChecker generates a long-press "Home" action (show the list of recently-opened apps) followed by a touch action on this app (switch back to the previous app again), triggering the lifecycle transition onStop → onRestart; for onDestroy, APEChecker generates a device-rotation event, triggering the transition onStop → onDestroy → onStart. Currently, APEChecker considers Activity, Fragment and other lifecycle-aware components (*e.g.*, Loader).

Environment Generation. APEChecker automatically constructs appropriate environment for event sequences. During the trace generation, APEChecker records the control-dependent conditions of an APE, and analyzes them to derive the environment. APEChecker currently focuses on three types of environment, which we find they can cover most of cases.

• *Specific user inputs.* APEChecker tracks the constraints on user inputs that can affect error verification. It currently supports two lightweight strategies: (1) infer the required input formats (*e.g.*, email address, phone number) from the property android:inputType in the UI layout files; and (2) track the intra-procedural data-flows of inputs and infer the required contents by analyzing simple string APIs (*e.g.*, equal to a constant string, contain a specific character).
• *Explicit system settings or permissions.* APEChecker analyzes specific APIs in the recorded conditions to infer necessary system settings or permissions. For example, if ¬WifiManager#isWifiEnabled() is the condition, APEChecker will disable WiFi before replaying the test; if Camera#open() is the condition, APEChecker will grant the camera access permission at runtime when required.
• *Specific Exception Handling.* Some APEs reside in exception handling code, and they cannot be manifested without triggering the corresponding exception. For example, if the APE can be only reached by an IOException when the app fails to access a remote server via network, APEChecker will disconnect the network before replaying the test to simulate the exception. APEChecker only handles specific cases of IOException (*e.g.*, cannot access network or files), but it can be extended to support other exceptions if required.

Currently, APEChecker supports limited environment (*e.g.*, specific user inputs, network connection, camera access, file access) in our evaluated subjects, and does not consider external events (*e.g.*, sensor inputs and intents). The environment can be extended in the future by using more sophisticated techniques like symbolic analysis [69] and exception handling [87].

3.5 Error Verification

To confirm an APE, we replay the generated event sequence with appropriate environment on the target app, and monitor the intended 9 exception types via Android Debugging Bridge (adb logcat). For those APEs whose manifestations require specific activity/fragment state and thread scheduling (*Fault Pattern 3* in particular), we instrument the original app A to A' by adding semaphore operations P (waiting) and V (release) at appropriate program locations. For a suspicious faulty statement i in the callback of async thread w, we insert a P operation (in the background callback) before i to wait for a signal; at the right activity or fragment lifecycle callback on the UI thread U, we insert a V operation to send the signal. As a result, we are able to control the thread scheduling and make sure the scheduling happen at the right lifecycle state. Note that if an app correctly follows the 3 async programming rules, the control of thread scheduling will not introduce APEs or force the app to crash. The instrumentation method only amplifies the possibility of the long execution time to simulate real possible scenarios (*e.g.*, delay of network access, wait of data download), and thus will not introduce new behaviors or change original behaviors. Additionally, the instrumentation locations depend on the fault types. For example, to manifest activity state loss, the V operation will be added in onStop since state loss always happens after onStop.
Example. The APE in SearchByPartName#onPostExecute in Fig 2 violates *Rule 3*. To manifest this error, APEChecker instruments a P operation in the end of doInBackground, and instruments a V operation at the beginning of SearchPanel#onDestroy. By doing this, APEChecker can easily manifest this error by a two-event sequence, *i.e.*, click the "Search" button and rotate the screen.

4 IMPLEMENTATION

APEChecker is implemented in Java (5K LOC) and Python (1K LOC), and built on several existing tools to automatically manifest APEs. It uses Soot [81] to build call graph, statically detects APEs based on the three fault patterns, and generates program traces. When generating program traces, it extends IC3 [65] to handle inner classes, fragments, and other public classes to reduce false negatives. It currently considers acyclic traces, and sets the maximum number of traces to 10 and maximum trace length to 20. It utilizes Gator [70] (the GUIHierarchyPrinterClient client in particular) to set up the mapping relations between user event handlers and UI elements, and converts the handlers (in the generated trace) to an event sequence (*e.g.*, click a button, choose an item in the list). The mapped UI elements usually have unique IDs or texts, which enables APEChecker to interact with them. It now supports back, rotate, Home (send the app to background), long-press-Home-and-back, Screen (screen on/off) to tweak lifecycle. Apktool [14] instruments semaphore P/V operations into the UI thread and async threads to control thread scheduling. UIAutomator [26] is used to execute tests, and Android Debugging Bridge (adb) [22] monitors whether the app throws the intended 9 exception types.

5 EVALUATION

We applied APEChecker on 40 real-world Android apps, and compared it with three state-of-the-art GUI testing tools (Monkey, Sapienz, and Stoat), the Google official static analysis tool Lint, and the state-of-the-art data race detection tool, EventRacer, to measure its effectiveness. We aim to answer these research questions.

- ***RQ1:*** How effective is APEChecker for detecting APEs in Android apps? Can Lint detect them?
- ***RQ2:*** How effective is APEChecker against existing GUI testing techniques (Monkey, Sapienz, and Stoat) for APEs?
- ***RQ3:*** How effective is APEChecker against existing data race detection techniques, for detecting specific types of APEs?

5.1 Evaluation Setup

Subjects. We choose subjects from (1) F-droid, the largest repository for open-source apps; and (2) Google Play Store, the official app store from Google. We crawled all 2097 unique apps from F-droid, and 3107 popular apps from Google Play with over 10K installations. Soot successfully processed 1654 F-droid apps and 2719 Google Play apps. Among them, 930 F-droid apps and 1274 Google Play apps use async constructs. Next, APEChecker identifies 866 APEs in 234 F-droid apps (25.2%=234/930) and 1161 APEs in 201 Google Play apps (15.8%=201/1274) that contain suspicious APEs, respectively.
Environment. APEChecker runs on a 64-bit Ubuntu 14.04 machine with 12 cores (3.50GHz Intel CPU) and 32GB RAM. We verify APEs on both an Android emulator (SDK 4.4.2) and an LG Nexus 5X mobile phone (SDK 7.1.1).
Studies. We conducted three case studies. In ***Study 1***, we answer ***RQ1***. We need to manually analyze each reported APE, and determine the true positives and false positives, which requires a lot of human efforts. Therefore, we randomly selected around 10% F-droid and Google Play apps, respectively (the apps requiring user credentials are excluded): (1) 25 F-droid apps from 234 suspicious faulty apps, with 30374 executable lines of code (LOC) (in Jimple), 143 classes, 680 methods and 6 activities on average; and (2) 15

Table 1: Subjects used in the experiment and evaluation results of APEChecker and Lint.

App Name	#ELOC	#Classes	#Methods	#Activities	#Time (min)	#APEs (detected)	#APEs (processed)	#Repro.	#FP	Fault Type	#APEs by Lint
Open Manager	11477	56	227	6	1.0	2	2	2	0	3	0
ADSdroid	2310	17	60	2	1.0	2	2	2	0	3	0
DeskCon	11264	80	317	5	9.8	7	5	4	0	3	0
TuCanMobile	24264	113	494	10	3.6	2	2	2	0	3	0
RadioDroid	19248	109	443	2	2.8	3	1	1	2	3	0
filmChecker	2739	28	67	2	2.9	1	1	1	0	3	0
QuranForMyAndroid	14170	99	414	11	2.5	4	4	2	0	2, 3	0
MoTAC	30857	135	748	6	1.8	3	2	2	0	2	0
MaximaOnAndroid	11357	43	211	4	2.5	2	2	2	0	3	0
DebianDroid	14133	88	413	3	3.1	8	0	0	0	-	0
NextGIS Mobile	12150	71	300	3	3.2	1	0	0	0	-	0
A2DP Volume	21502	115	555	6	3.7	6	6	5	0	2, 3	0
Andor	87818	445	2198	19	3.0	4	3	3	1	3	0
Mitzuli	23615	135	517	2	3.4	2	1	1	0	3	0
Commons	32767	177	912	9	4.1	3	1	1	0	3	0
AdAway	21443	131	535	8	4.2	2	0	0	0	-	0
ServeStream	64866	300	1777	8	4.8	1	1	1	0	3	0
Navit	19336	69	360	3	1.7	2	0	0	1	-	0
JKU App	52654	223	1263	7	5.2	2	0	0	0	-	0
HomeManager	9184	54	185	2	1.8	1	1	1	0	1	0
Transports Bordeaux	21840	180	668	16	7.3	3	1	0	1	-	0
MTG Familiar	84544	341	1529	12	7.2	1	0	0	0	-	0
Cowsay	4149	22	92	1	1.1	1	1	1	0	1	1
AeonDroid	23037	128	604	4	3.6	3	1	1	0	3	0
Addi	138716	404	2119	2	1.4	1	0	0	0	-	0
Average	**30374**	**143**	**680**	**6**	**3.5**	**2.8**	-	-	-	-	-
MalayalamNewspaper	7093	51	181	4	8.2	1	1	1	0	3	0
Drum Solo	17819	80	270	4	2.6	1	0	0	0	-	0
Smart Poker	38835	219	1020	7	5.1	3	3	1	0	3	0
Lojas Renner	24758	210	756	11	5.6	1	1	1	0	3	0
Messaging	114001	446	3103	13	13.2	1	0	0	0	-	0
Recarga Vivo	61913	369	1809	25	13.8	1	0	0	0	-	0
InstaCartoonPhoto	1693	26	60	6	6.8	1	1	1	0	3	0
Fingerprint Lock	17833	123	475	13	6.9	1	0	0	0	-	0
Salmos	19527	110	402	4	5.9	2	2	2	0	3	0
Santander	166101	902	3457	5	9.8	3	2	1	0	3	0
Biblia Sagrada	41170	153	648	10	16.6	7	2	2	1	3	0
Trade Accounting	85485	450	2018	34	11.8	2	0	0	0	-	0
PremiumWallpaper	52156	203	1321	10	8.3	4	4	4	0	2, 3	0
Tebak Lagu	23017	158	667	12	11.5	9	7	5	0	2, 3	0
WorldNews Live24	104436	515	2298	10	19.8	3	1	1	0	3	0
Average	**51722**	**267**	**1232**	**12**	**9.9**	**2.7**	-	-	-	-	-

Google Play apps with 51722 LOC, 267 classes, 1232 methods and 12 activities on average, shown in Table 1. We measured the static analysis time (the time of replay tests is omitted, since it only takes a few seconds), the number of detected APEs, the number of APEs that can be processed (limited by the abilities of Soot, IC3, Gator), the number of APEs that can be reproduced (#Repro.), false positives (#FP), the types of faults, and the number of APEs that can be detected by Lint.

In ***Study 2***, we answer ***RQ2*** by comparing APEChecker with Monkey, Sapienz, and Stoat, which have proven effectiveness on bug detection [12, 60, 78]. 10 apps are randomly selected from 234 F-droid apps that contain APEs. Each tool is allocated with one hour with default settings. The number of confirmed (*i.e.*, successfully triggered) APEs, the analysis time and the length of tests (*i.e.*, event sequences) are recorded. From the perspective of their approach workflow, all tools are given the original rather than the instrumented apps (*cf.* Section 3.5) as input to achieve fair comparison — Monkey, Sapienz and Stoat do not require the instrumentation for bug detection as their algorithms do not need instrumentation (otherwise they may be adversely affected, *e.g.*, stuck by the thread scheduling), and also do not have idea of where to instrument. To alleviate the randomness, we run each tool 10 times to average data.

In ***Study 3***, we answer ***RQ3*** by comparing APEChecker with EventRacer [10]. We did not choose other data race tools, *e.g.*, CAFA [45] (not available) and DroidRacer [29, 59] (less effecitve and precise than EventRacer [10]). In detail, we (1) randomly selected 10 F-droid apps (since the source code is required to confirm the APEs reported by EventRacer, listed in Table 2) with at least one APE of *Fault Pattern 3* (treated as data races by EventRacer); (2) only consider the races reported between UI thread and async threads; and (3) allocated 10,000 events for EventRacer (by default, only 1,000 events) to generate event sequences, which on average costs 15 minutes per app (comparable to the running time of APEChecker). EventRacer generates a lot of false positives, but to our knowledge, no reproducing tools can facilitate our analysis: AsyncDroid [66] requires user-provided event traces, RacerDroid [80] adopts manually analysis and ERVA [47] is not available. Therefore, we manually analyzed the races, and resorted to developers for confirmation.

5.2 Study 1: Effectiveness of Detecting APEs

Table 1 shows the results of APEChecker on the 40 apps. It identified 107 APEs in total with 67 and 40 from F-droid and Google Play apps, respectively. Among them, due to the limitations of underlying tools, 61 APEs can be processed, of which 51 are successfully confirmed with real tests, achieving 83.6% (51/61) hit rate. On average, it takes

6.7 minutes for one app, with 3.5 minutes for one F-droid app and 9.9 minutes for one Google Play app. In contrast, Lint only detects one APE, which is ineffective. We detail the results below.

Effectiveness. APEChecker successfully reproduces all APEs of *Fault Pattern 1*. HomeManager manages and switches between home applications. Once started, it scans the installed apps on the device in an async thread, which modifies the data list of app info asychronously (with a ListView). When the scan is finished, it notifies the UI thread to refresh the ListView with the new list. However, if the list is changed again (another async thread is created) during the refresh, the app can be crashed by an IllegalStateException.

In *Fault Pattern 2*, *MTG Familiar* is an offline database app for magic cards. It takes the photos of cards to identify the magic card. After taking a photo, it starts loading the photo with a ProgressDialog and calls Toast#makeText in an async thread. However, it fails to post Toast creation onto the UI thread, which causes a RuntimeException.

APEChecker successfully reproduces most APEs of *Fault Pattern 3*. For example, *Mitzuli*, a translator app, starts an async thread to check version update, and shows a dialog to notify the results. However, if users rotate the device before the thread returns, a fatal BadTokenException occurs.

Hit Rate. APEChecker achieves an 83.6% (51/61) hit rate. We analyzed the failed cases, and found most of them require special constraints. We summarize the main scenarios below.

(1) External environment. Transports Bordeaux, a transportation app, searches for paths from the starts to destinations. APEChecker can generate a program trace but cannot generate an event sequence to trigger the error, since the app registers a Broadcast Receiver for receiving specific intents. APEChecker cannot infer this now. *QuranForMyAndroid* is a translator for "The Quran". If "Back" button is pressed after editting the text, it starts a thread to save the text into storage and shows a message via Toast#makeText only when no data storage is available. APEChecker cannot reproduce such errors.

(2) Complicated Exceptions. APEChecker currently cannot infer complicated exceptions such as file not found and database corruption. One APE in *A2DP Volume* is that it tries to show a message via Toast#makeText in the exception handling's catch statement which, however, is only reachable when the database to be loaded is corrupted in try statement.

False Positives. APEChecker only reports 6 false positives (5.6%=6/107). The main reason is that some methods or classes are actually unreachable. For example, in *RadioDroid*, APEChecker identifies 3 potential APEs, 2 of which are false positive since they reside in a dead activity (ActivityRadioStationDetail) that cannot be started by any other activities.

In summary, on the evaluated apps, APEChecker can efficiently manifest APEs in a few minutes with 83.6% hit rate and 5.6% false alarms.

5.3 Study 2: Comparison with Testing Tools

Fig. 6 compares APEChecker and the three testing tools in the number of confirmed APEs, the analysis time (in minutes) and the length of event sequence (in logarithmic scale) that triggers the errors. The results are presented in three bar plots with error bars (shown as black lines), where each bar's height indicates the average value, and the error bar represents the standard deviation of uncertainty.

APEChecker confirms many more APEs than the testing tools. In the 10 runs, APEChecker totally confirms 32 unique APEs from these

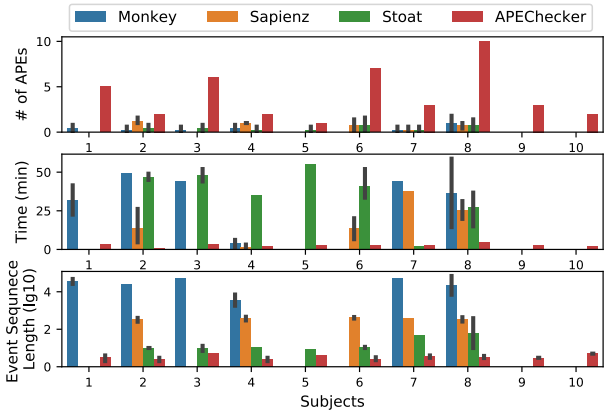

Figure 6: Comparison between APEChecker and testing tools.

10 apps, while Monkey, Sapienz and Stoat, respectively, confirm 7, 8, and 9. Moreover, none of the testing tools uncover APEs in subjects *9* and *10* while APEChecker finds 3 and 2 ones, respectively. The error bars of testing tools show obvious data variance, *i.e.*, testing tools are very likely to miss APEs.

For the testing tools, the analysis time of an error is its first occurrence time since one error can be triggered multiple times during one run. We can see APEChecker takes much less time than the testing tools except for the subjects *4* and *7*. On average, APEChecker takes 2.9 minutes, while Monkey, Sapienz, and Stoat require 32.7, 14.3 and 37.1 minutes, respectively. Sapienz can easily hit one of the errors in the subject *4* but take more time for the other twos. Stoat takes less time on the subject *7*, but only hits that APE once during the 10 runs. Additionally, in terms of the event sequence length, APEChecker provides shorter (more usable) tests than the testing tools. The average event sequence length of Monkey, Sapienz, Stoat and APEChecker is 35725, 411, 12, 4, respectively.

In summary, on the evaluated apps, APEChecker detects 3X more APEs than testing tools, reduces detection time from half an hour to a few minutes, and provides more usable reproducing tests.

5.4 Study 3: Comparison with EventRacer

Table 2 compares APEChecker and EventRacer on 10 F-droid apps in the number of detected APEs. For EventRacer, we give the number of reported data races and true positives under two configurations (1K-event and 10K-event). To confirm true positives, we followed the approach of EVRA [47] and remove duplicated races: if data races are reported on different variables in ADF but refer to the same location of app code, we treat them as one true positive. Because the races actually indicate the same error in the app.

From Table 2, we can see APEChecker is more effective than EventRacer for the specific APEs of *Fault Pattern 3*. APEChecker detects and reproduces 32 out of 38 APEs while EventRacer (10K-event) only finds 6 ones. By comparing the results between 1K-event and 10K-event, we can note the effectiveness of EventRacer heavily depends on the number of given events. Although EventRacer can detect 5 more APEs with 10K-event, it reports 14X more false positives, which overwhelms users. This is also observed by EVRA [47] that only 3% data races reported by EventRacer are harmful. We also observe that the reported races are highly random, which further undermines its effectiveness for detecting these specfic APEs.

Table 2: Comparison between APEChecker and EventRacer

App Name	#APEs (Patt.3)	APECh. #Repro.	EventRacer (1K)		EventRacer (10K)	
			#Reported	#TP	#Reported	#TP
A2DPVolume	5	5	41	0	1249	1
TuCanMobile	2	2	12	0	53	0
OpenManager	2	2	0	0	27	1
Andor	4	3	0	0	0	0
Maxima	2	2	1	0	36	1
filmChecker	1	1	36	0	139	0
DeskCon	7	5	0	0	29	1
RadioDroid	3	1	15	1	48	1
ADSdroid	2	2	2	0	1	0
AnyMemo	10	9	19	0	206	1
Total	**38**	**32**	**126**	**1**	**1788**	**6**

In summary, on the evaluated apps, APEChecker detects 5X more APEs than EventRacer on a specific APE type with very few false alarms.

5.5 Limitations and Threats to Validity

Limitations. APEChecker builds on several popular research tools, *i.e.*, Soot, IC3, and Gator, which represent the state-of-the-art. However, they still have several limitations in practice, which directly fail APEChecker on program trace generation. For example, even if Soot successfully processes an app, the call graph can still be incomplete (21.3% cases in our evaluation). Moreover, IC3 cannot handle ICCs in inner classes or fragments, and Gator cannot handle third-party UI controls. To counter this, we have already extended IC3 and Gator in these aspects (Section 4). But APEChecker still cannot generate valid program traces for 46 APEs detected in the evaluated apps (*e.g.*, *DebianDroid*) due to the limitations of underlying tools. We believe more engineering efforts on these tools can significantly improve APEChecker. Additionally, APEChecker now only infers simple environment, which lowers the hit rate. Future work will integrate more sophisticated analysis techniques to overcome this. Aditionally, applying APEChecker to detect APEs for other GUI frameworks like Qt requires proper extensions [72].

Threats to Validity. APEChecker is only evaluated on 40 apps, and thus our conclusion may not be general to all apps. The summarized async programming rules and fault patterns may be incomplete, and thus APEChecker may suffer from false negatives. But we have thoroughly inspected all relevant resources, and the fault patterns have covered all 375 real APEs from 1091 apps. To ease the verification of APEs, APEChecker simulates specific execution environment by controlling thread scheduling (*e.g.*, increase the wait time of network access). So some triggered APEs may not be easily manifested by users although they are real faults under specific conditions.

6 RELATED WORK

Automated GUI Testing Many GUI testing techniques, *e.g.*, random testing [57], search-based testing [58, 60], symbolic execution [5, 63, 82], model-based testing [2, 3, 8, 11, 76, 78, 85] and other approaches [61, 62, 75, 83], have been proposed for Android apps. As we discussed in Section 1 and demonstrated in the evaluation, these techniques are ineffective for APEs due to lack of prior knowledge (fault patterns). In contrast, APEChecker leverages this knowledge to efficiently manifest APEs.

APEChecker tackles specific bugs that violate the rules of single-GUI-thread model. QUANTUM [86] uses model-based testing to tackle app-agnostic bugs. For example, when rotation happens, the screen should show the same content and support the actions as before. Thor [1] amplifies existing tests by randomly injecting neutral event sequences (*e.g.*, double rotation, pause-and-resume) that should not affect the outputs of original tests. Amalfitano *et al.* [4] propose a similar idea by exhaustively injecting orientation changes to expose GUI failures. However, these techniques may not be effective for APEs. First, their effectiveness depends on the quality of an existing GUI model or test suite, which are usually not available [51]. Second, these techniques are random, and do not explicitly consider the interactions between UI and aysnc threads. Graziussi [44] encodes some specific patterns into Lint to detect lifecycle bugs. KREfinder [74] finds restart-and-resume errors that cause app data loss. APEChecker tackles APEs which have not been systematically investigated before.

Concurreny Bugs As discussed in Section 2.4, some APEs can be treated as concurrency bugs. To tackle them, a number of data race detection tools [10, 45, 59, 71] and data race reproducing tools [47, 66, 80] are developed. However, as our evaluation shows, data race detection tools are ineffective for these APEs. AsyncDroid [66] and RacerDroid [80] are impractical since they either require user-provided traces or depend on the results of data race detection tools. ERVA [47] utilizes dependency graph and event flipping to reproduce races, and uses state comparison techniques to identify harmful ones. AATT [52] manifests concurrency bugs whose race points at the app code level. However, it can only manifest races whose conflicting events are in the same page or race points in the user code. Other bug reproducing tools like RERAN [20] and CrashScope [64] require user interactions. In contrast, APEChecker automatically generates event sequences to verify APEs.

Guided Test Generation Our work uses static analysis to locate suspicious APEs, and then guides GUI interactions to verify them. This idea is also adopted in other application domains of Android. SmartDroid [89] guides tests to reach certain APIs to manifest the malicious behaviors. However, it explores every UI element on each page to find the right view to click. Brahmastra [9] drives apps to reach third-party components to test security issues; MAMBA [50] guides tests to reveal potential accesses to privacy-sensitive data; FuzzDroid [69] utilizes a search-based algorithm combined with static and dynamic analysis to manifest malicious behaviors. However, none of these tools can be directly compared with APEChecker for detecting APEs. Other targeted testing tools include Collider [49], ConDroid [73], A^{3}E [8]. However, they are either not available or requiring human intervention.

7 CONCLUSION

This paper introduces APEChecker, a technique to efficiently manifest APEs. First, we conduct a formative study to understand the aysnc programming rules implied by the single-GUI-thread model, and collect a set of real APE issues. Second, informed by these results, we distill three fault patterns, and locate suspicious APEs via static fault pattern analysis, and then verify them with real tests. The evaluation shows APEChecker is useful and effective.

ACKNOWLEDGMENTS

We appreciate the reviewers' constructive feedback. This work is partially supported by NSFC Grant 61502170, NTU Research Grant NGF-2017-03-033 and NRF Grant CRDCG2017-S04. Lingling Fan is partly supported by ECNU Project of Funding Overseas Short-term Studies, Ting Su is partially supported by NSFC Grant 61572197 and 61632005, and Geguang Pu is partially supported by MOST NKTSP Project 2015BAG19B02 and STCSM Project 16DZ1100600.

REFERENCES

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic Execution of Android Test Suites in Adverse Conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 83–93.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*. 258–261.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2015), 53–59.

[4] Domenico Amalfitano, Vincenzo Riccio, Ana C. R. Paiva, and Anna Rita Fasolino. 2018. Why does the orientation change mess up my Android application? From GUI failures to code faults. *Softw. Test., Verif. Reliab.* 28, 1 (2018).

[5] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*. 59.

[6] Apple. 2018. App Programming Guide for iOS. Retrieved 2018-7 from https://developer.apple.com/library/content/navigation/

[7] Apple. 2018. MacOS Cocoa. Retrieved 2018-7 from https://developer.apple.com/macos/

[8] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 641–660.

[9] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving Apps to Test the Security of Third-party Components. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, USA, 1021–1036.

[10] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2015. Scalable Race Detection for Android Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 332–348.

[11] Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 623–640.

[12] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 429–440.

[13] CodePath. 2018. CodePath Android Cliffnotes. Retrieved 2018-7 from http://guides.codepath.com/android

[14] Apktool developers. 2018. A tool for reverse engineering Android apk files. Retrieved 2018-7 from https://ibotpeaches.github.io/Apktool/

[15] Adsdroid Developers. 2018. Adsdroid. Retrieved 2018-7 from https://github.com/dnet/adsdroid

[16] Android Gisapp Developers. 2018. Android Gisapp. Retrieved 2018-7 from https://github.com/nextgis/android_gisapp

[17] Privacy Friendly Pedometer Developers. 2018. Privacy Friendly Pedometer. Retrieved 2018-7 from https://github.com/SecUSo/privacy-friendly-pedometer

[18] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in Android apps. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 408–419.

[19] FindBugs. 2018. FindBugs. Retrieved 2018-7 from http://findbugs.sourceforge.net/

[20] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd D. Millstein. 2013. RERAN: timing- and touch-sensitive record and replay for Android. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 72–81.

[21] Google. 2018. Activity Lifecycle. Retrieved 2018-7 from https://developer.android.com/guide/components/activities/activity-lifecycle.html

[22] Google. 2018. Android Debug Bridge. Retrieved 2018-7 from https://developer.android.com/studio/command-line/adb.html

[23] Google. 2018. Android Developers Documentation. Retrieved 2018-7 from https://developer.android.com

[24] Google. 2018. Android Lint Checks. Retrieved 2018-7 from http://tools.android.com/tips/lint-checks

[25] Google. 2018. Android Platform. Retrieved 2018-7 from https://www.android.com/

[26] Google. 2018. Android UI Automator. Retrieved 2018-7 from http://developer.android.com/tools/help/uiautomator/index.html

[27] Google. 2018. AsyncTask. Retrieved 2018-7 from https://developer.android.com/reference/android/os/AsyncTask.html

[28] Google. 2018. Dialogs. Retrieved 2018-7 from https://developer.android.com/guide/topics/ui/dialogs.html

[29] Google. 2018. DroidRacer. Retrieved 2018-7 from https://bitbucket.org/iiscseal/droidracer

[30] Google. 2018. Fragment Lifecycle. Retrieved 2018-7 from https://developer.android.com/guide/components/fragments.html

[31] Google. 2018. Fragment Transactions and Activity State Loss. Retrieved 2018-7 from https://www.androiddesignpatterns.com/2013/08/fragment-transaction-commit-state-loss.html

[32] Google. 2018. Handler. Retrieved 2018-7 from https://developer.android.com/reference/android/os/Handler.html

[33] Google. 2018. HandlerThread. Retrieved 2018-7 from https://developer.android.com/reference/android/os/HandlerThread.html

[34] Google. 2018. IntentService. Retrieved 2018-7 from https://developer.android.com/reference/android/app/IntentService.html

[35] Google. 2018. Lint. Retrieved 2018-7 from https://developer.android.com/studio/write/lint.html

[36] Google. 2018. List View. Retrieved 2018-7 from https://developer.android.com/guide/topics/ui/layout/listview.html

[37] Google. 2018. Loaders. Retrieved 2018-7 from https://developer.android.com/guide/components/loaders.html

[38] Google. 2018. Looper. Retrieved 2018-7 from https://developer.android.com/reference/android/os/Looper.html

[39] Google. 2018. Monkey. Retrieved 2018-7 from http://developer.android.com/tools/help/monkey.html

[40] Google. 2018. Processes and Threads. Retrieved 2018-7 from https://developer.android.com/guide/components/processes-and-threads.html

[41] Google. 2018. Thread. Retrieved 2018-7 from https://developer.android.com/reference/java/lang/Thread.html

[42] Google. 2018. ThreadPoolExecutor. Retrieved 2018-7 from https://developer.android.com/reference/java/util/concurrent/ThreadPoolExecutor.html

[43] Google. 2018. Toasts. Retrieved 2018-7 from https://developer.android.com/guide/topics/ui/notifiers/toasts.html

[44] Simone Graziussi. 2016. *Lifecycle and Event-Based Testing for Android Applications*. Master's thesis. School Of Industrial Engineering and Information, Politecnico.

[45] Chun-Hung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. 2014. Race detection for event-driven mobile applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 326–336.

[46] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 77–83.

[47] Yongjian Hu, Iulian Neamtiu, and Arash Alavi. 2016. Automatically Verifying and Reproducing Event-based Races in Android Apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 377–388.

[48] JDK. 2018. JDK Swing Platform. Retrieved 2018-7 from https://docs.oracle.com/javase/7/docs/technotes/guides/swing/

[49] Casper Svenning Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*. 67–77.

[50] Joseph Chan Joo Keng, Lingxiao Jiang, Tan Kiat Wee, and Rajesh Krishna Balan. 2016. Graph-aided Directed Testing of Android Applications for Checking Runtime Privacy Behaviours. In *Proceedings of the 11th International Workshop on Automation of Software Test (AST '16)*. ACM, New York, NY, USA, 57–63.

[51] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the Test Automation Culture of App Developers. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. 1–10.

[52] Qiwei Li, Yanyan Jiang, Tianxiao Gu, Chang Xu, Jun Ma, Xiaoxing Ma, and Jian Lu. 2016. Effectively Manifesting Concurrency Bugs in Android Apps. In *23rd Asia-Pacific Software Engineering Conference, APSEC 2016, Hamilton, New Zealand, December 6-9, 2016*. 209–216.

[53] Yu Lin, Semih Okur, and Danny Dig. 2015. Study and Refactoring of Android Asynchronous Programming (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 224–235.

[54] Yu Lin, Cosmin Radoi, and Danny Dig. 2014. Retrofitting Concurrency for Android Applications Through Refactoring. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 341–352.

[55] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys

Poshyvanyk. 2017. Enabling Mutation Testing for Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 233–244.

[56] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*. 95–111.

[57] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 224–234.

[58] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 599–609.

[59] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race detection for Android applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 316–325.

[60] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. 94–105.

[61] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd intelligence enhances automated mobile testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 16–26.

[62] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 559–570.

[63] Nariman Mirzaei, Sam Malek, Corina S. Pasareanu, Naeem Esfahani, and Riyadh Mahmood. 2012. Testing Android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[64] Kevin Moran, Mario Linares Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*. 33–44.

[65] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-component Communication Analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 77–88.

[66] Burcu Kulahcioglu Ozkan, Michael Emmi, and Serdar Tasiran. 2015. Systematic Asynchrony Bug Exploration for Android Apps. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 455–461.

[67] PMD. 2018. PMD. Retrieved 2018-7 from https://pmd.github.io/

[68] Qt. 2018. Qt. Retrieved 2018-7 from https://www.qt.io/

[69] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. 300–311.

[70] Atanas Rountev and Dacong Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*. 143.

[71] Gholamreza Safi, Arman Shahbazian, William G. J. Halfond, and Nenad Medvidovic. 2015. Detecting Event Anomalies in Event-based Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 25–37.

[72] Anirudh Santhiar, Shalini Kaleeswaran, and Aditya Kanade. 2016. Efficient race detection in the presence of programmatic event loops. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. 366–376.

[73] Julian Schütte, Rafael Fedler, and Dennis Titze. 2015. ConDroid: Targeted Dynamic Analysis of Android Applications. In *29th IEEE International Conference on Advanced Information Networking and Applications, AINA 2015, Gwangju, South Korea, March 24-27, 2015*. 571–578.

[74] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 2016. Finding Resume and Restart Errors in Android Applications. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 864–880.

[75] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: Beyond GUI Testing for Android Applications. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 27–37.

[76] Ting Su. 2016. FSMdroid: guided GUI testing of android apps. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. 689–691.

[77] Ting Su, Zhoulai Fu, Geguang Pu, Jifeng He, and Zhendong Su. 2015. Combining Symbolic Execution and Model Checking for Data Flow Testing. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 654–665.

[78] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 245–256.

[79] SWT. 2018. SWT. Retrieved 2018-7 from https://www.eclipse.org/swt/

[80] Hongyin Tang, Guoquan Wu, Jun Wei, and Hua Zhong. 2016. Generating Test Cases to Expose Concurrency Bugs in Android Applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 648–653.

[81] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 13.

[82] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2012. Verifying Android Applications Using Java PathFinder. *SIGSOFT Softw. Eng. Notes* 37, 6 (Nov. 2012), 1–5.

[83] Mario Linares Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2015. Mining Android App Usages for Generating Actionable GUI-Based Execution Scenarios. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. 111–122.

[84] wxErlang. 2018. wxErlang. Retrieved 2018-7 from http://erlang.org/doc/apps/wx/index.html

[85] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 250–265.

[86] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST '14)*. IEEE Computer Society, Washington, DC, USA, 183–192.

[87] Pingyu Zhang and Sebastian G. Elbaum. 2014. Amplifying Tests to Validate Exception Handling Code: An Extended Study in the Mobile Application Domain. *ACM Trans. Softw. Eng. Methodol.* 23, 4 (2014), 32:1–32:28.

[88] Sai Zhang, Hao Lü, and Michael D. Ernst. 2012. Finding Errors in Multithreaded GUI Applications. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. 243–253.

[89] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '12)*. ACM, New York, NY, USA, 93–104.