Localizing Exception Faults in Android Applications

Hamed Mirzaei  
Department of Computer Engineering,  
Sharif University of Technology, Tehran, Iran  
hamedmirzaei@ce.sharif.edu

Abbas Heydarnoori  
Department of Computer Engineering,  
Sharif University of Technology, Tehran, Iran  
The Corresponding Author, heydarnoori@sharif.edu  
Mobile#: +98-912-8186631

Abstract

In software programs, most of the time, there is a chance of faults in general, and exception faults in particular. Localizing those pieces of code which are responsible for a particular fault is one of the most complicated tasks and it can make incorrect results if done manually. Semi-automated and fully-automated techniques have been introduced to overcome this issue. However, despite recent advances in fault localization techniques, they are not necessarily applicable to Android applications because of their special characteristics like context-awareness, use of sensors, being executable on various mobile devices, limited hardware resources, and so on. To this aim, in this paper, we introduce a semi-automated hybrid method that combines static and dynamic analysis to localize exception faults in Android applications. Our evaluations with nine open source Android applications of different sizes with various exceptions show that our proposed technique can correctly identify the root causes of occurred exceptions. These results indicate that our proposed approach is effective in practice in localizing exception faults in Android applications.  

Keywords: Fault localization; Unhandled exceptions; Exception faults; Android applications.

1 Introduction

The goal of software testing is to make sure that a program does what it was designed to do, and conversely, it does not do anything unintended [22]. Due to the diversity of application configurations, platforms, and inputs, software testing is typically an unpleasant, complicated, difficult to automate, and costly process [46]. Consequently, software systems are often not thoroughly tested and they usually include some faults. Hence, there is a high demand for automating the process of localizing faults in faulty software applications [39] [40]. Fault localization is the process of identifying the pieces of an application’s source code which are responsible for occurring that fault. Semi-automated and fully-automated fault localization methods have been introduced to decrease the developers’ involvement during the process of fault localization and to increase the precision of results [38]. In recent years, smart mobile devices, and specially Android ones, are widely used and millions of applications have been developed for them. Mobile applications are event-driven, i.e., they respond to user and/or system generated actions [24]. They are also context-aware which means that they can behave differently in different situations.
and locations \cite{27}. Additionally, they can use various sensors of mobile devices, can be executed on different smart mobile devices with various properties, should be executed and tested with limited hardware resources available on smart mobile devices, new technologies are used in their developments, and so on \cite{21}. All these special characteristics force mobile application developers to look for new fault localization methods \cite{21,18}.

To address the above issue, a number of approaches have been proposed in literature for fault localization in smart mobile applications. However, since different types of faults exist, each technique focuses on a particular type of fault, such as profile leakages, extremely energy usages, or wasting the memory spaces. In particular, Egele et al. \cite{6} propose an approach to detect profile leakages in iOS applications. On the other hand, Gibler et al. \cite{7} use a different approach for finding profile leakages in Android applications. To detect energy related faults, Vekris et al. \cite{36} define some energy policies, and examine Android applications to see whether or not those policies have been followed. In a different work, Pathak et al. \cite{26} use the data-flow analysis to detect no-sleep energy faults which keep the phone awake all the time. In another work, Gottschalk et al. \cite{8} define a number of energy-related code smells and try to remove them via re-engineering the source code. The Graphical User Interface (GUI) is an important part of any event-driven application. Some approaches like the ones done by Takala et al. \cite{32}, Yang et al. \cite{11}, and Hu et al. \cite{11} focus on detecting GUI-related faults in mobile applications. More specifically, they are trying to find objects on the screen with conflicting boundaries or out of bound layouts. On the other hand, in our proposed approach, we are using the GUI events as well, but our focus is on what is going on beyond the GUI and in the source code itself, not just the GUI. This means that our approach is unable to find those out of bound and conflicting boundary faults.

One of the most important kinds of faults is unhandled exception faults which occur in specific unwanted situations and force the application to stop. Although these faults are very important since they can crash the whole application, to the best of our knowledge, no approaches have been proposed in literature to localize this kind of faults in Android applications. To address this issue, in this paper, we introduce a semi-automated hybrid method that combines the dynamic and static analysis to localize unhandled exception faults in Android applications. Our approach mainly includes three phases: extraction, execution, and evaluation. In the extraction phase, a behavioral graph for the Android application under the test is automatically generated using that application’s activities, objects, and events. Then that graph is used to automatically generate a set of test cases for that application. In the execution phase, those test cases are executed over the application and their execution traces are profiled. If an exception occurs during the execution of a test case, in the evaluation phase, we rank lines of the application’s source code based on their relevance to occurring that exception fault. An application may have more than one unhandled exception fault. To localize multiple exception faults in an Android application, our approach executes every single test case of the application. If a test case throws an exception, it would be recorded by the approach and the next test case would be examined. This process keeps running until all the test cases are examined.

To rank lines of the application’s source code with respect to their relevance to causing an exception, we use three different scores: the test case score, the value pattern score, and the backward static slicing score. The test case score indicates the execution frequency of each line of the code in the failed and passed test cases. It is obvious that if a line of the code executes more in failed test cases than passed test cases, it is more likely to being faulty. To calculate this score, we use a combination of Tarantula \cite{13} and Jaccard \cite{3} metrics. We use the value pattern score to detect lines of the source code that are not related to occurring that exception. Hence, this score helps to reduce the search space by removing unrelated statements and thus, fault localization would be more precise and faster. In contrast to the value pattern score, backward
static slicing score detects those lines of the code which are related to occurring that exception. This score reduces the search space by choosing related program statements.

We have implemented our proposed approach as a tool for Java and used it to localize various exceptions in nine open source Android applications of different sizes. Our evaluations indicate that our approach works as expected in most cases and can localize lines of the code that are responsible for raising exceptions. We also compared our ranking metric with the Tarantula and Jaccard as two powerful ranking metrics, and noticed that ours practically outperforms them. The contributions of this work are: (i) a new ranking metric to rank lines of the source code based on their relevance to occurring an exception; (ii) the ability to localize multiple exception faults in a single run of the approach; and (iii) a prototype implementation of the proposed approach.

The rest of this paper is organized as follows: Section 2 provides a running example which is used throughout this paper. Section 3 then introduces our proposed approach and ranking metric. Next, Section 4 describes our prototype implementation of the proposed approach. Afterwards, Section 5 presents the results of our evaluations of the proposed technique on nine open source Android applications of different sizes and with various exceptions. Section 6 provides a discussion about various aspects of the proposed approach. Section 7 considers the related work. Finally, Section 8 concludes the paper and provides future research directions.

2 Running Example

In this section, we provide an example Android program in order to clarify the problem that our proposed approach aims to tackle. This example is then used throughout this paper. However, first, we briefly explain the structure of an Android application to make the paper more understandable for readers who are not familiar with Android programming.

Each Android application is composed of the following four main components:

1. **Activity**: An activity represents a single screen with a user interface. It is the main component in any Android application since it is the point where users interact with the application. Each action of the user with the application’s user interface is known as an event to the application which will be responded by an event listener.

2. **Service**: A service is a component that runs in the background to perform long-running operations. For example, a service might synchronize emails while the user is in a different application.

3. **Broadcast receivers**: Broadcast receivers simply respond to broadcast messages from other applications or from the system. For example, applications can initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use. Thus, this is the broadcast receiver who will intercept this communication and will initiate an appropriate action.

4. **Content providers**: A content provider component supplies data from one application to others on request.

Android applications are *event-driven* in nature [20]. The user works with the user interface and makes a request by triggers events. When an event is triggered, the user interface sends it to the back-end source code and after processing the event, the response is returned to the user. The main problem with testing these applications is the huge number of available events. In

[^http://developer.android.com]
addition, the user can trigger any possible sequence of events and this would lead testers to a huge number of test scenarios.

Listing 1: MainActivity.java

```java
8. public class MainActivity extends AppCompatActivity {
9.   Button button1, button2;
10. @Override
11.   public void onCreate(Bundle savedInstanceState) {
12.     super.onCreate(savedInstanceState);
13.     setContentView(R.layout.main_activity);
14.     addListenerOnButton();
15. }
16. public void addListenerOnButton() {
17.     final Context context = this;
18.     button1 = (Button) findViewById(R.id.button1);
19.     button2 = (Button) findViewById(R.id.button2);
20.     button1.setOnClickListener(new View.OnClickListener() {
21.         @Override
22.         public void onClick(View arg0) {
23.             Intent intent = new Intent(context, SubActivity1.class);
24.             startActivity(intent);
25.         }
26.     });
27.     button1.setOnLongClickListener(new View.OnLongClickListener() {
28.         public boolean onLongClick(View v) {
29.             Intent intent = new Intent(context, SubActivity2.class);
30.             startActivity(intent);
31.             return true;
32.         }
33.     });
34.     button2.setOnClickListener(new View.OnClickListener() {
35.         @Override
36.         public void onClick(View arg0) {
37.             Intent intent = new Intent(context, SubActivity1.class);
38.             startActivity(intent);
39.         }
40.     });
41.     button2.setOnLongClickListener(new View.OnLongClickListener() {
42.         public boolean onLongClick(View v) {
43.             return true;
44.         }
45.     });
46. }
47.}
```

Listing 2: SubActivity1.java

```java
13. public class SubActivity1 extends AppCompatActivity {
14.   TextView textView2;
15.   Button button3, button4;
16.   int num = 8;
17. @Override
18.   public void onCreate(Bundle savedInstanceState) {
19.     super.onCreate(savedInstanceState);
20.     setContentView(R.layout.sub_activity_1);
21.     addListeners();
22. }
23. private void addListeners() {
24.     num = 8;
25.     final Context context = this;
26.     button3 = (Button) findViewById(R.id.button3);
27.     button4 = (Button) findViewById(R.id.button4);
28.     textView2 = (TextView) findViewById(R.id.textView2);
29.     textView2.setClickable(true);
30.     textView2.setText(Html.fromHtml("<a href='http://www.google.com'>Google</a>"));
31.     button3.setOnClickListener(new View.OnClickListener() {
32.         @Override
33.         public void onClick(View arg0) {
34.             Intent intent = new Intent(context, SubActivity2.class);
35.             addListeners();
36.         }
37.     });
38.     button4.setOnClickListener(new View.OnClickListener() {
39.         @Override
40.         public void onClick(View arg0) {
41.             Intent intent = new Intent(context, SubActivity1.class);
42.             startActivity(intent);
43.         }
44.     });
45. }
46. }
47.}
```
Now that the reader is familiar with the structure of Android applications, suppose an Android application with the following four activities as the running example: MainActivity

```java
Listing 3: SubActivity2.java

public class SubActivity2 extends AppCompatActivity {
    TextView textView3;
    Button button5;
    Context context = this;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.sub_activity_2);
        addListeners();
    }
    private void addListeners() {
        button5 = (Button) findViewById(R.id.button5);
        textView3 = (TextView) findViewById(R.id.textView3);
        textView3.setClickable(true);
        button5.setOnLongClickListener(new View.OnLongClickListener() {
            public boolean onLongClick(View v) {
                Intent browserIntent = new Intent(Intent.ACTION_VIEW, Uri.parse("http://www.google.com"));
                startActivity(browserIntent);
                return true;
            }
        });
        textView3.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View arg0) {
                Intent intent = new Intent(context, SubActivity3.class);
                startActivity(intent);
            }
        });
    }
}

Listing 4: SubActivity3.java

public class SubActivity3 extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.sub_activity_3);
    }
}
```

Now that the reader is familiar with the structure of Android applications, suppose an Android application with the following four activities as the running example: MainActivity
Listing 1, SubActivity1 (Listing 2), SubActivity2 (Listing 3) and SubActivity3 (Listing 4). The MainActivity is the main activity that includes two buttons and works normally without any exceptions. However, the SubActivity1 includes a textView and two buttons, and will throw a NullPointerException on the long click event of the first button and a ResourceNotNotFoundException on the click event of the second one. Nevertheless, SubActivity2 has one textView and one button, and SubActivity3 includes only one textView. Both SubActivity2 and SubActivity3 work normally without any exceptions.

The NullPointerException (Exception1 in the rest of this paper) in SubActivity1 occurs because of the following reasons: (i) an invalid assignment to variable num in line 42; (ii) an invalid assignment to variable view in line 43; (iii) the missing of the else statement in the if block of line 44; and finally (iv) the execution of line 47 which throws the exception; The ResourceNotNotFoundException (Exception2 in the rest of this paper) in SubActivity1 occurs because of an invalid Resource at line 56. In the next section, we are going to introduce our proposed exception fault localization approach and apply it to our running example.

3 Proposed Approach

3.1 Approach Overview

Before delving into the details of the proposed approach, we first provide an overview. The approach is a hybrid one, i.e., it uses both of the Android application’s source code and the test cases’ runtime traces to detect lines of the program’s source code which are responsible for raising exceptions. From the user’s perspective, the proposed approach has three phases as depicted in Fig. 1. The extraction and execution phases are fully automated, while the evaluation phase consists of both manual and automated sub-phases. In the extraction phase, all the information about the application’s activities, objects, and events are extracted. In addition, the events that transfer the control of the application from one activity to another are identified. In the execution phase, a set of test cases are generated and their execution traces are profiled. Finally, in the evaluation phase, our technique uses the collected test cases’ traces and the application’s source code to rank program statements based on their probability of causing the exception fault. In the following, we discuss the details of these phases with the help of the example provided in Section 2.

3.2 Approach Details

The proposed approach is a hybrid approach which statically analyzes the source code of an Android application and dynamically executes its test cases to locate those program statement which are responsible for occurring an exception fault. The rest of this section discusses the details of the extraction, execution, and evaluation phases which are followed in this process as depicted in Fig. 1.

3.2.1 The Extraction Phase

The main goal of this phase is to extract all the information about the Android application’s activities, objects, and events. Furthermore, control changer events which are events that transfer the control of the application from one activity to another are identified in this phase. To this aim, first, a list of fireable events for each type of the objects of the application is generated. For instance, for a button object, events like click and long click can be generated. Next, for each activity of the application which is reachable from the main activity, all the information
Phase 1: Extraction

1. Extracting information for objects on reachable activities (e.g., type, size, and location of a button)

A list of fireable events for each object

Phase 2: Execution

2.1. Generating test case for activity under test

2.2. Executing test case and collect trace

2.3. Detecting control changer events

For each activity accessible from current activity and not visited before

Phase 3: Evaluation

3. Generating coverage test cases

For each test case

4.1. Executing test case on application under test

4.2. Collecting trace and test case outcome

5.1. Clustering test cases based on proposed clustering method

5.2. Ranking lines of code based on proposed ranking metric

Traces of passed test cases

Traces of failed test cases

Figure 1: The proposed exception fault localization approach for Android applications
## Table 1: The control changer events of the running example presented in Section 2

<table>
<thead>
<tr>
<th>Event Number</th>
<th>Source</th>
<th>Destination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>event₁</td>
<td>MainActivity</td>
<td>SubActivity₁</td>
<td>Click on button₁</td>
</tr>
<tr>
<td>event₂</td>
<td>MainActivity</td>
<td>SubActivity₂</td>
<td>Long click on button₁</td>
</tr>
<tr>
<td>event₃</td>
<td>MainActivity</td>
<td>SubActivity₁</td>
<td>Click on button₂</td>
</tr>
<tr>
<td>event₄</td>
<td>MainActivity</td>
<td>MainActivity</td>
<td>Long click on button₂</td>
</tr>
<tr>
<td>event₅</td>
<td>MainActivity</td>
<td>MainActivity</td>
<td>Click on textView₁</td>
</tr>
<tr>
<td>event₆</td>
<td>SubActivity₁</td>
<td>SubActivity₂</td>
<td>Click on button₃</td>
</tr>
<tr>
<td>event₇</td>
<td>SubActivity₁</td>
<td>NullPointerException</td>
<td>Long click on button₄</td>
</tr>
<tr>
<td>event₈</td>
<td>SubActivity₁</td>
<td>ResourceNotFoundException</td>
<td>Click on button₅</td>
</tr>
<tr>
<td>event₉</td>
<td>SubActivity₁</td>
<td>SubActivity₁</td>
<td>Long click on button₄</td>
</tr>
<tr>
<td>event₁₀</td>
<td>SubActivity₁</td>
<td>InvalidActivity</td>
<td>Click on textView₂</td>
</tr>
<tr>
<td>event₁₁</td>
<td>SubActivity₂</td>
<td>SubActivity₂</td>
<td>Click on button₅</td>
</tr>
<tr>
<td>event₁₂</td>
<td>SubActivity₂</td>
<td>InvalidActivity</td>
<td>Long click on button₅</td>
</tr>
<tr>
<td>event₁₃</td>
<td>SubActivity₂</td>
<td>SubActivity₃</td>
<td>Click on textView₃</td>
</tr>
<tr>
<td>event₁₄</td>
<td>SubActivity₃</td>
<td>SubActivity₃</td>
<td>Click on textView₄</td>
</tr>
</tbody>
</table>

about its objects are extracted (e.g., information like size, unique-id, and location for the button object). We say that an activity $B$ is reachable from the activity $A$ if there exists at least one sequence of events starting from the activity $A$ and ending at the activity $B$.

To detect control changer events, for each activity, a test case which triggers all the possible events on the activity is generated. Next, control changer events are detected by comparing the activities before and after triggering the events and seeing if the control of the application has been transferred from one activity to another. To perform this, in our implementations of the proposed approach (see Section 4), we use the [AndroidViewClient](https://github.com/dtmilano/AndroidViewClient) (AVC) library of the Python programming language. Table 1 lists control changer events for our running example.

We have basically two kinds of control changer events which are of our interest in this work: (i) some events transfer the control of the application to an activity outside of it (e.g., clicking on a button may open an external browser), and (ii) some events may transfer the control of the application to an exception fault. For the first kind of events, since all these events are treated equally in our approach, for the sake of simplicity, we assume that there exists an activity, named `InvalidActivity`, that all these events transfer the control to it. For the second category of events, we define an activity, named `ErrorActivity`, and assume that all the events causing an exception, transfer the control to it.

At the end of the extraction phase, we know all the events, and their source and destination activities. To accomplish the task of generating test cases in the next phase of our proposed approach, we create a labeled graph $G(V, L, E)$ (e.g., Fig. 2) in which:

- $V$: The set of nodes such that each node is an activity from the set of reachable activities from the `MainActivity`, `InvalidActivity`, `ErrorActivity`;

- $L$: The set of labels that denote the fireable events of the application; and

https://github.com/dtmilano/AndroidViewClient
• \( E \): The set of edges. An edge from the activity \( A \) to activity \( B \) with a label \( l \) means that the firing of event \( l \) in activity \( A \) transfers the applications’ control to activity \( B \).

Table 2: The set of test cases generated for the running example presented in Section 2. Test cases which their event sequences are a prefix of another test case are not mentioned here because they are redundant. Passed test cases \( TC_9, TC_{14}, TC_{19} \) and \( TC_{22} \) are not considered in the evaluation phase because they are irrelevant to the faulty activity \( SubActivity1 \).

<table>
<thead>
<tr>
<th>Status</th>
<th>TC</th>
<th>The sequence of events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passed</td>
<td>( TC_1 )</td>
<td>( MainActivity \rightarrow event_4 \rightarrow MainActivity \rightarrow event_5 \rightarrow MainActivity \rightarrow event_3 \rightarrow SubActivity1 \rightarrow event_9 \rightarrow SubActivity1 \rightarrow event_7 \rightarrow ErrorActivity )</td>
</tr>
<tr>
<td>Passed</td>
<td>( TC_2 )</td>
<td>( MainActivity \rightarrow event_4 \rightarrow MainActivity \rightarrow event_5 \rightarrow MainActivity \rightarrow event_1 \rightarrow SubActivity1 \rightarrow event_9 \rightarrow SubActivity1 \rightarrow event_7 \rightarrow ErrorActivity )</td>
</tr>
<tr>
<td>Passed</td>
<td>( TC_3 )</td>
<td>( MainActivity \rightarrow event_5 \rightarrow MainActivity \rightarrow event_4 \rightarrow MainActivity \rightarrow event_3 \rightarrow SubActivity1 \rightarrow event_9 \rightarrow SubActivity1 \rightarrow event_7 \rightarrow ErrorActivity )</td>
</tr>
<tr>
<td>Passed</td>
<td>( TC_4 )</td>
<td>( MainActivity \rightarrow event_5 \rightarrow MainActivity \rightarrow event_4 \rightarrow MainActivity \rightarrow event_1 \rightarrow SubActivity1 \rightarrow event_9 \rightarrow SubActivity1 \rightarrow event_7 \rightarrow ErrorActivity )</td>
</tr>
<tr>
<td>Failed</td>
<td>( TC_5 )</td>
<td>( MainActivity \rightarrow event_4 \rightarrow MainActivity \rightarrow event_5 \rightarrow MainActivity \rightarrow event_3 \rightarrow SubActivity1 \rightarrow event_9 \rightarrow SubActivity1 \rightarrow event_8 \rightarrow ErrorActivity )</td>
</tr>
<tr>
<td>Failed</td>
<td>( TC_6 )</td>
<td>( MainActivity \rightarrow event_4 \rightarrow MainActivity \rightarrow event_5 \rightarrow MainActivity \rightarrow event_1 \rightarrow SubActivity1 \rightarrow event_9 \rightarrow SubActivity1 \rightarrow event_8 \rightarrow ErrorActivity )</td>
</tr>
<tr>
<td>Failed</td>
<td>( TC_7 )</td>
<td>( MainActivity \rightarrow event_5 \rightarrow MainActivity \rightarrow event_4 \rightarrow MainActivity \rightarrow event_3 \rightarrow SubActivity1 \rightarrow event_9 \rightarrow SubActivity1 \rightarrow event_8 \rightarrow ErrorActivity )</td>
</tr>
<tr>
<td>Failed</td>
<td>( TC_8 )</td>
<td>( MainActivity \rightarrow event_5 \rightarrow MainActivity \rightarrow event_4 \rightarrow MainActivity \rightarrow event_1 \rightarrow SubActivity1 \rightarrow event_9 \rightarrow SubActivity1 \rightarrow event_8 \rightarrow ErrorActivity )</td>
</tr>
<tr>
<td>Passed</td>
<td>( TC_9 )</td>
<td>( MainActivity \rightarrow event_4 \rightarrow MainActivity \rightarrow event_5 \rightarrow MainActivity \rightarrow event_2 \rightarrow SubActivity2 \rightarrow event_{11} \rightarrow SubActivity2 \rightarrow event_{12} \rightarrow InvalidActivity )</td>
</tr>
<tr>
<td>Passed</td>
<td>( TC_{10} )</td>
<td>( MainActivity \rightarrow event_4 \rightarrow MainActivity \rightarrow event_5 \rightarrow MainActivity \rightarrow event_3 \rightarrow SubActivity1 \rightarrow event_9 \rightarrow SubActivity1 \rightarrow event_6 \rightarrow SubActivity2 \rightarrow event_{11} \rightarrow SubActivity2 \rightarrow event_{12} \rightarrow InvalidActivity )</td>
</tr>
<tr>
<td>Passed</td>
<td>( TC_{11} )</td>
<td>( MainActivity \rightarrow event_4 \rightarrow MainActivity \rightarrow event_5 \rightarrow MainActivity \rightarrow event_3 \rightarrow SubActivity1 \rightarrow event_9 \rightarrow SubActivity1 \rightarrow event_{10} \rightarrow InvalidActivity )</td>
</tr>
<tr>
<td>Passed</td>
<td>( TC_{12} )</td>
<td>( MainActivity \rightarrow event_4 \rightarrow MainActivity \rightarrow event_5 \rightarrow MainActivity \rightarrow event_1 \rightarrow SubActivity1 \rightarrow event_9 \rightarrow SubActivity1 \rightarrow event_6 \rightarrow SubActivity2 \rightarrow event_{11} \rightarrow SubActivity2 \rightarrow event_{12} \rightarrow InvalidActivity )</td>
</tr>
<tr>
<td>Passed</td>
<td>( TC_{13} )</td>
<td>( MainActivity \rightarrow event_4 \rightarrow MainActivity \rightarrow event_5 \rightarrow MainActivity \rightarrow event_1 \rightarrow SubActivity1 \rightarrow event_9 \rightarrow SubActivity1 \rightarrow event_{10} \rightarrow InvalidActivity )</td>
</tr>
</tbody>
</table>
### 3.2.2 The Execution Phase

In this phase, a set of test cases is automatically generated using the data from the extraction phase. For instance, Table 2 illustrates a set of test cases for our running example. This set should cover all the functionalities executable from the application’s GUI (Graphical User Interface). Lee et al. [15] did a statistical analysis and showed that redundant test cases will decrease the precision of results. A redundant test case is a test case that is a prefix of another one, and the execution of the bigger test case will guarantee the execution of all the functionalities executed by the redundant one. Many of the fault localization approaches, including ours, use ranking
metrics to detect faulty statements. The ranking metrics themselves are based on the execution frequency of program statements in passed and failed test cases. Consequently, if redundant test cases are taken into account, the results would be biased towards the pieces of code which are visited during their execution. This can affect the precision of results, and hence, we ignore redundant test cases in our approach.

As discussed in Section 2, Android applications are event-driven, i.e., they respond to user and/or system generated actions. The main challenge in testing event-driven applications which is very time-consuming is the huge number of events in the application [9]. Since an exception fault can occur in any of the event-handlers of an application, we need to test all of them in our approach. In our test case generator module, we adapt the approach presented in [9]. More specifically, we perform a two-step process to generate test cases: (i) generating all the possible passed and failed test cases, and then, (ii) pruning the set of generated test cases. In the first step, the set of passed test cases are those paths of the generated graph (e.g., Fig. 2) that end at a node other than the ErrorActivity; and the set of failed test cases are those which end at the ErrorActivity. While performing the first step, i.e., generating the test cases, four important rules should be considered. These rules are necessary to claim that the set of test cases will cover all the functionalities of the application:

1. Any of the edges of the generated graph, including loop edges, should be visited in at least one test case. This is because each edge represents an event of the application.

2. Since the execution order of events can make different results, all the possible permutations of events should be considered.

3. To avoid infinite paths in the case of loops, it has been decided to meet all the loop edges of each node just for once in the entrance of the node. This may decrease the final precision of our approach in some special cases but we accept it and try to overcome this issue in the future work.

Figure 2: The labeled graph generated for the running example presented in Section 2.
4. Any of the test cases, which is a prefix of another one, will be removed from the set of test cases because redundant test cases would decrease the performance of the approach. [15]

After employing the two-step process discussed above to automatically generate test cases for an Android application under the test, we automatically run those test cases on the application, and label each test case as passed or failed. To reach this goal, in our implementations of the proposed approach (see Section 4), we use the AVC library of the Python programming language. For instance, Table 2 indicates whether each test case was labeled as passed or failed for our running example. In addition to labeling each test case as passed or failed, their execution traces are also profiled. Each execution trace includes the program statements during the test case execution, and their order of execution. Additionally, all the value assignments to program variables are also profiled.

3.2.3 The Evaluation Phase

The evaluation phase is the final phase of our proposed approach which uses the information gathered so far to rank lines of the application’s source code based on their probability of being faulty. Our approach’s precision is highly dependent on the coverage criteria which is used to generate test cases out of the generated graph. Our coverage criteria, as explained earlier, is not complete in the case of loop edges, and thus, we cannot claim that our approach is capable of detecting all the exception faults. For example, if an application throws an exception whenever a button is continuously tapped for five times, our approach is unable to detect it. Nevertheless, in other situations, we can claim that our approach is able to detect all unhandled exception faults. Consequently, as discussed below, the first step in this phase is to cluster test cases with respect to occurred exception faults.

Clustering Test Cases Fig. 3 indicates how we benefit from the clustering mechanism to cluster test cases with respect to occurred exception faults for the running example presented in Section 2. At first, those test cases that are related to SubActivity1.java will be selected. We
say a test case is related to a file $\text{A.java}$ if at least one of the program statements of that file is visited during the execution of that test case. Consequently, the test cases $\text{TC}_9$, $\text{TC}_{14}$, $\text{TC}_{19}$, and $\text{TC}_{22}$ will be pruned since they are irrelevant to $\text{SubActivity1.java}$. Next, the related failed test cases will be clustered based on the occurred exceptions. For example, test cases $\text{TC}_1$, $\text{TC}_2$, $\text{TC}_3$, and $\text{TC}_4$ will become a cluster related to $\text{Exception}_1$ (i.e., Failed Cluster$_1$) and test cases $\text{TC}_5$, $\text{TC}_6$, $\text{TC}_7$, and $\text{TC}_8$ will become another cluster related to $\text{Exception}_2$ (i.e., Failed Cluster$_2$). At the end, to localize each exception, we analyze the related passed test cases and the corresponding failed test cases cluster. For example, to localize $\text{Exception}_1$ in our running example, we analyze the Passed and Failed Cluster$_1$ test cases. Note that by this definition, a passed test case can be related to many exceptions, while a failed test case is exactly related to one exception. The step after clustering the test cases is to rank lines of the application’s source code based on their probability of causing each exception. To this aim, we will propose a ranking metric. However, before introducing that, we discuss the weaknesses of existing ranking metrics.

### Weaknesses of Existing Ranking Metrics

The ranking metrics $\text{Tarantula}$ [13, 14] and $\text{Jaccard}$ [3] are widely used in literature to rank suspicious lines of code. However, in our experiments, we noticed that these metrics cannot necessarily rank the lines of code correctly in some situations. For instance, consider $\text{Exception}_2$ in our running example (i.e., line 56 of $\text{SubActivity1.java}$). Based on the generated test cases in Table 2 and the clustering results in Fig. 5, we see that both of the ranking metrics $\text{Tarantula}$ and $\text{Jaccard}$ not only will return line 56 (which is the real reason of occurring the exception), but also they will incorrectly introduce line 54 and line 55 as the main reasons of occurring the exception. This problem is because these ranking metrics try to rank a program statement by only its execution frequency in test cases. As discussed below, we try to tackle this problem by proposing a ranking metric that employs other information as well.

### Proposed Ranking Metric

As discussed above, existing ranking metrics, like $\text{Tarantula}$ and $\text{Jaccard}$, only use the execution frequency of each statement in passed and failed test cases to rank it as relevant to an occurred fault. However, this might lead to incorrect results in cases like the one occurred in $\text{Exception}_2$ of the running example (i.e., line 56 of $\text{SubActivity1.java}$). This problem is coming from irrelevant statements which participate in the ranking process. However, our studies showed that there exist two other factors that complement each other and can be used to detect irrelevant statements. These factors together try to remove irrelevant statements by reducing the search space for faulty statements. The first one uses the slicing techniques [35] to detect those statements that are relevant to faulty statements, and to mark the others as irrelevant. The other factor, which we called it the value-pattern, uses the value assignments of variables in passed and failed test cases to partition the program statements as relevant or irrelevant. Thus, as discussed below, our ranking metric (i.e., $S(l)$ in Equation 1) combines these two factors with the execution frequency of program statements in passed and failed test cases. We will provide the evaluations of our proposed ranking metric in Section 5.

$$S(l) = VP(l) \ast (a \ast \text{Prob}(l) + (1 - a) \ast \text{Slice}(l)) \tag{1}$$

1. **Value Pattern Score ($VP(l)$):** For each variable in the application source code and each test case, a sequence of pairs < line number, value > which is called a value pattern is generated from the traces of test cases. Please note that as discussed before, we profile the assignments to variables as well. We define $VP_{\text{pass}}(v)$ and $VP_{\text{fail}}(v)$ as the set of value patterns generated for variable $v$ from the passed and failed test cases respectively. For a variable $v$, if $VP_{\text{fail}}(v)$ be a subset of $VP_{\text{pass}}(v)$, then that variable $v$ is marked as unrelated to the
Table 3: The value patterns for the variable num of SubActivity1.java. PVP: Passed Value Pattern; FVP: Failed Value Pattern.

<table>
<thead>
<tr>
<th>Exception</th>
<th>PVPs</th>
<th>FVPs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception₁</td>
<td>{&lt;16,8&gt; &lt;24,8&gt;}</td>
<td>{&lt;16,8&gt; &lt;24,8&gt; &lt;42,7&gt;}</td>
<td>The lines that use the variable num will get 1 for the VP score because FVPs is not a subset of PVPs.</td>
</tr>
<tr>
<td>Exception₂</td>
<td>{&lt;16,8&gt; &lt;24,8&gt;}</td>
<td>{&lt;16,8&gt; &lt;24,8&gt;}</td>
<td>The variable num has no effects on the VP score because FVPs is a subset of PVPs.</td>
</tr>
</tbody>
</table>

occurred exception. Based on the generated test cases for our running example (Table 2) and the source code of SubActivity1.java (Listing 2), the value patterns for variable num are listed in Table 3. We conclude from the calculated value patterns that for the Exception₁, the lines that use variable num will get 1 for the VP score since VP_menu(num) is not a subset of VP_pass(num). Nevertheless, for the Exception₂, this variable is an unrelated variable and has no effects on the VP score since VP_menu(num) is a subset of VP_pass(num). In general, the VP(l) value would be 0 if and only if all the variables used in line l have been marked as unrelated variables and it would be 1 otherwise.

- **Backward static slicing score (Slice(l))**: According to the definition of slicing, the static slice of line l is the set of program statements that may affect the values of variables in line l. Therefore, when an exception occurs in a line of code, it can be because of some faults in some parts of the static slice of that line of code. Thus, this score is used to highlight related lines of code to the line where the exception has occurred. So, for each line l of the application’s source code, if l be in the static slice of the faulty line, then Slice(l) would be 1, otherwise it would be 0. So, in Listing 2, the backward static slice of line 47 are lines 16, 24, 42, 43, 44, and 47, and the Slice(l) score for these lines would be 1.

- **Test Case Score (Prob(l))**: The execution frequency of each line of code in failed and passed test cases is a useful metric to rank them. Notice that based on our proposed clustering criteria for test cases, for each exception, these frequencies are calculated on related test cases, and not all of them. After analyzing the Tarantula and the Jaccard ranking metrics, we found out that the Tarantula metric assigns a higher and the Jaccard metric assigns a lower probability to some lines of code than what is expected. So, we use an average of these two (i.e., Equation 2) in our proposed ranking metric (i.e., Equation 1). In Equation 2, STarantula(l) and SJaccard(l) respectively denote the Tarantula and Jaccard ranking metrics of line l.

\[
Prob(l) = \frac{(STarantula(l) + SJaccard(l))}{2}
\]  

- **The a Parameter**: The parameter a in Equation 1 controls the effects of the test case and the backward static slicing scores in the final ranking score. Because the backward static slice score outweighs the test case score, the value of the a parameter must be less than 0.5. To gain the best ranking results, we analyzed different values for this parameter in our
Table 4: Different values of the $a$ parameter used to calculate the probability of being faulty for some lines of the Gallery case study.

<table>
<thead>
<tr>
<th>Gallery line numbers</th>
<th>$a=0.5$</th>
<th>$a=0.45$</th>
<th>$a=0.4$</th>
<th>$a=0.35$</th>
<th>$a=0.3$</th>
<th>$a=0.25$</th>
<th>$a=0.2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>MainActivity: 19</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MainActivity: 22</td>
<td>0.556</td>
<td>0.602</td>
<td>0.646</td>
<td>0.69</td>
<td>0.73</td>
<td>0.779</td>
<td>0.82</td>
</tr>
<tr>
<td>MainActivity: 23(true)</td>
<td>0.567</td>
<td>0.699</td>
<td>0.73</td>
<td>0.766</td>
<td>0.79</td>
<td>0.833</td>
<td>0.87</td>
</tr>
<tr>
<td>MainActivity: 23(false)</td>
<td>0.5</td>
<td>0.55</td>
<td>0.6</td>
<td>0.65</td>
<td>0.7</td>
<td>0.75</td>
<td>0.8</td>
</tr>
<tr>
<td>MainActivity: 25</td>
<td>0.567</td>
<td>0.699</td>
<td>0.73</td>
<td>0.766</td>
<td>0.79</td>
<td>0.833</td>
<td>0.87</td>
</tr>
<tr>
<td>MainActivity: 26</td>
<td>0.567</td>
<td>0.699</td>
<td>0.73</td>
<td>0.766</td>
<td>0.79</td>
<td>0.833</td>
<td>0.87</td>
</tr>
</tbody>
</table>

evaluations (Section 5) on all the case studies, and compared the ranking results with what we have expected. In particular, we expected to find the main cause of faults and rank the candidate lines correctly. Our results showed that for values of $a$ greater than 0.2, their ranking may be acceptable, but the probability assigned to each line is not so realistic. For values of $a$ near 0.5, the assigned probability is lower than what it should be. For example, Table 4 illustrates the effect of different values for the $a$ parameter for an example case study. Thus, our experimental results indicated that $a = 0.2$ gives the best results.

After applying the proposed ranking metric on our running example, we got the results presented in Table 5 for Exception 1, and the results in Table 6 for Exception 2. These results illustrate that lines 42, 43, 44(false), and 47 of SubActivity1.java are the most probable sources of Exception 1, and line 56 of SubActivity1.java is the root cause of Exception 2. The line number 44(false) means that the line 44 is executed with the false condition.

4 Prototype Implementation

We prototyped our approach as a Java project in Android Developer Tools (ADT). In order to localize exception faults in a desired Android application, that application should be first imported into the ADT. Next, while it is being executed in the ADT simulator, our tool automatically applies the approach presented in Section 3.2 to generate test cases for that application. Next, our tool analyzes the profiled execution traces of those test cases to localize exception faults in that Android application. To implement the test cases generator module, we used the AVC library of the Python programming language. Interested readers can refer to [1] to access the source codes of our implementations.

5 Evaluation

In this section, we present the evaluations of our proposed approach for localizing exceptions in Android applications. In particular, we present the objectives, the setup, and the results of our evaluations.
Table 5: Ranking metrics results for Exception_1 in the running example (i.e., NullPointerException). PTCs: Passed Test Cases; FTCs: Failed Test Cases.

<table>
<thead>
<tr>
<th>Line Numbers</th>
<th>PTCs</th>
<th>FTCs</th>
<th>(S_{\text{Tarantula}}(l))</th>
<th>(S_{\text{Jaccard}}(l))</th>
<th>(\text{Prob}(l))</th>
<th>(\text{Slice}(l))</th>
<th>(\text{VP}(l))</th>
<th>(S(l))</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>MainActivity: 9, 12, 13, 14, 17, 18, 19, 43</td>
<td>12</td>
<td>4</td>
<td>0.5</td>
<td>0.25</td>
<td>0.375</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>MainActivity: 23, 24</td>
<td>6</td>
<td>2</td>
<td>0.5</td>
<td>0.2</td>
<td>0.35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>MainActivity: 37, 38</td>
<td>6</td>
<td>2</td>
<td>0.5</td>
<td>0.2</td>
<td>0.35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>SubActivity1: 14, 15, 19, 20, 21, 25, 26, 27, 28, 29, 30, 31, 32</td>
<td>12</td>
<td>4</td>
<td>0.5</td>
<td>0.25</td>
<td>0.375</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>SubActivity1: 16, 24</td>
<td>12</td>
<td>4</td>
<td>0.5</td>
<td>0.25</td>
<td>0.375</td>
<td>1</td>
<td>1</td>
<td>0.875</td>
<td>2</td>
</tr>
<tr>
<td>SubActivity1: 36, 37</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SubActivity1: 42, 43, 44(false), 47</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SubActivity2: 21, 22, 23</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SubActivity2: 26, 27, 28</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SubActivity2: 34, 35</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SubActivity3: 7, 8</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Others</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6: Ranking metrics results for Exception_2 in the running example (i.e., ResourceNotFoundException). PTCs: Passed Test Cases; FTCs: Failed Test Cases.

<table>
<thead>
<tr>
<th>Line Numbers</th>
<th>PTCs</th>
<th>FTCs</th>
<th>(S_{\text{Tarantula}}(l))</th>
<th>(S_{\text{Jaccard}}(l))</th>
<th>(\text{Prob}(l))</th>
<th>(\text{Slice}(l))</th>
<th>(\text{VP}(l))</th>
<th>(S(l))</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>MainActivity: 9, 12, 13, 14, 17, 18, 19, 43</td>
<td>12</td>
<td>4</td>
<td>0.5</td>
<td>0.25</td>
<td>0.375</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>MainActivity: 23, 24</td>
<td>6</td>
<td>2</td>
<td>0.5</td>
<td>0.2</td>
<td>0.35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>MainActivity: 37, 38</td>
<td>6</td>
<td>2</td>
<td>0.5</td>
<td>0.2</td>
<td>0.35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>SubActivity1: 14, 15, 19, 20, 21, 25, 26, 27, 28, 29, 30, 31, 32</td>
<td>12</td>
<td>4</td>
<td>0.5</td>
<td>0.25</td>
<td>0.375</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>SubActivity1: 16, 24</td>
<td>12</td>
<td>4</td>
<td>0.5</td>
<td>0.25</td>
<td>0.375</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>SubActivity1: 36, 37</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SubActivity1: 54, 55</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.2</td>
<td>2</td>
</tr>
<tr>
<td>SubActivity1: 56</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SubActivity2: 21, 22, 23</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>SubActivity2: 26, 27, 28</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>SubActivity2: 34, 35</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>SubActivity3: 7, 8</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Others</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
5.1 Evaluation Objectives

In our evaluations, we are in favor of answering the following research questions to evaluate the effectiveness of our approach as well as our ranking metric:

1. **RQ1:** Is the proposed approach capable of identifying the lines of an Android application’s source code who are responsible for occurring an exception fault?

2. **RQ2:** How precise the proposed ranking metric is compared to two widely used Tarantula and Jaccard ranking metrics?

RQ1 intends to ensure that the proposed approach localizes exception faults correctly. In addition, we expect that it can localize multiple exceptions in a single run of the approach. The goal of RQ2 is to compare the precision of our ranking metric with the existing ones. We claim and expect that our ranking metric works better than the Tarantula and Jaccard metrics in ranking suspicious lines of code that might be responsible for occurring an exception.

5.2 Evaluation Setup

To answer the research questions provided in Section 5.1, we pursued the following steps to perform the evaluations of our proposed technique.

**Selection of Case Studies** To evaluate our proposed approach and our ranking metric, we chose nine real-world open-source Android applications of different sizes with various exceptions. These applications are either used in evaluating other fault localization methods, or are published in Android markets like Google Play and CafeBazaar. Published applications are often exception free, hence we injected some exceptions into their source codes manually. Table 7 lists our selected case studies. As can be seen in this table, we also chose a number of exception-free applications (i.e., Tippy Tipper, Gestures Builder, and 24Game) to see whether our approach can distinguish them. For these applications, our approach finishes after generating the labeled graph (see Section 3.2). If there are no exceptions in an application, there should not be any edges to the ErrorActivity in the labeled graph, and thus our approach finishes without generating the test cases. Additionally, to evaluate the applicability of our approach in localizing multiple exception faults in a single execution of the approach, we chose two case studies, i.e., the Calculator and the Running Example (see Section 2), that include two exceptions. Interested readers are referred to [1] to access the source codes of our case studies.

**Experimental Design** To answer the research questions provided in Section 5.1, we applied our proposed approach to the case studies presented in Table 7. For this purpose, we used our prototype implementations (see Section 4). Our approach utilizes the three main phases described in Section 3.2 to localize exceptions in each case study. At first, the approach analyzes the application under the test to extract its structure including the activities, objects, and their properties. Then, it makes a graph out of the activities as nodes and the events as edges. The test cases generator module then uses the obtained graph to generate a set of test cases that cover all the possible events in the application. The next step is executing those test cases over the application. For this purpose, the Python’s AVC library comes to help and executes each test case while the application is being executed in the ADT (Android Developer Tools) simulator. Traces of these test cases will be recorded, and finally, they are used to rank lines of the application’s source code based on their probability of being faulty.

Table 7: The selected case studies for our evaluations. LOC: Line Of Code; NOO: Number Of Objects; NOA: Number Of All Exceptions; NOU: Number Of Unique Exceptions.

<table>
<thead>
<tr>
<th>Case Study</th>
<th>LOC</th>
<th>NOO</th>
<th>NOA</th>
<th>NOU</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculator</td>
<td>110</td>
<td>10</td>
<td>2</td>
<td>2</td>
<td>Two NumberFormatException</td>
</tr>
<tr>
<td>Tippy Tipper</td>
<td>312</td>
<td>39</td>
<td>0</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>Gallery</td>
<td>129</td>
<td>11</td>
<td>4</td>
<td>1</td>
<td>ActivityNotFoundException</td>
</tr>
<tr>
<td>3000 Pishvaz Code</td>
<td>2162</td>
<td>86</td>
<td>1</td>
<td>1</td>
<td>ResourceNotFoundException</td>
</tr>
<tr>
<td>24Game</td>
<td>228</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>Running Example</td>
<td>251</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>NullPointerException and ResourceNotFoundException</td>
</tr>
<tr>
<td>Gestures Builder</td>
<td>176</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>None</td>
</tr>
<tr>
<td>Tomdroid</td>
<td>36708</td>
<td>76</td>
<td>0</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>FBReader</td>
<td>76148</td>
<td>879</td>
<td>1</td>
<td>1</td>
<td>ArrayIndexOutOfBoundsException</td>
</tr>
</tbody>
</table>

Table 8: Evaluation results.

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of Test Cases</th>
<th>Main Causes of Exceptions</th>
<th>Detected Causes of Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculator</td>
<td>8</td>
<td>The object amount1 has no value</td>
<td>The highest scores are given to lines 17, 49 and 75 which are responsible for assigning a value to object amount1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>The object amount2 has no value</td>
<td>The highest scores are given to lines 17, 50 and 76 which are responsible for assigning a value to object amount2</td>
</tr>
<tr>
<td>TippyTipper</td>
<td>44</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>---</td>
<td>---</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td>Gallery</td>
<td>6</td>
<td>4</td>
<td>(i) The variable Component is not declared in line 26, (ii) The object Intent is not instantiated correctly in line 25, (iii) The true condition for the if block in line 23</td>
</tr>
<tr>
<td>3000 Pishvaz Code</td>
<td>128</td>
<td>4</td>
<td>(i) The execution of line 79 of CategoryActivity activity, (ii) The execution of the switch block with an incorrect value</td>
</tr>
<tr>
<td>24Game</td>
<td>128</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>Running Example</td>
<td>12</td>
<td>4</td>
<td>(i) Line 47 of SubActivity1 which throws an exception, (ii) Missing else statement in the if block of line 44 of SubActivity1, (iii) A bad assignment to variable view in line 43 of SubActivity1, (iv) A bad assignment to variable num in line 42 of SubActivity1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>4</td>
<td>The resource with id 10 does not exist</td>
</tr>
<tr>
<td>Gestures Builder</td>
<td>48</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>------------------</td>
<td>----</td>
<td>---</td>
<td>------</td>
</tr>
<tr>
<td>Tomdroid</td>
<td>127</td>
<td>5</td>
<td>(i) A bad assignment to variable acceptedFileExtensions at line 107 or 118, (ii) The execution of line 116 with the true condition, (iii) A bad assignment to variable collection at line 117</td>
</tr>
<tr>
<td>FBReader</td>
<td>798</td>
<td>84</td>
<td>(i) A bad assignment to variable myEditPosition at line 105, (ii) Calling the function with an incorrect argument at line 106, (iii) The execution of the switch block with input zero at line 94 and the execution of line 96</td>
</tr>
</tbody>
</table>

5.3 Evaluation Results

Table 8 provides the results of applying our approach to the case studies presented in Table 7. In particular, Table 8 indicates the number of generated passed and failed test cases, the real causes of faults, and the detected causes of faults by our approach, and Table 9 indicates the generated graph size and the execution time of different phases for all the case studies. As mentioned before, the source codes of our case studies are available online at [1] and interested readers can consider the results themselves. As the results show, for all the case studies, our approach works as expected and is able to detect the main causes of occurred exceptions. These results answer our first evaluations’ research question, i.e., RQ1, and confirm that our proposed approach is capable of correctly localizing exception faults in the source codes of Android applications.
Table 9: The graph size and the execution time of applying our approach on different case studies. EGT: Extracting Graph Time; GTCT: Generating Test Cases Time; ETCT: Executing Test Cases Time; RT: Ranking Time; All times are in minutes.

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Graph(V,E)</th>
<th>EGT</th>
<th>GTCT</th>
<th>ETCT</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculator</td>
<td>(3, 13)</td>
<td>3.7</td>
<td>&lt;1</td>
<td>42</td>
<td>&lt;2</td>
</tr>
<tr>
<td>Tippy Tipper</td>
<td>(5, 48)</td>
<td>12.6</td>
<td>&lt;1</td>
<td>66</td>
<td>&lt;2</td>
</tr>
<tr>
<td>Gallery</td>
<td>(4, 24)</td>
<td>4.4</td>
<td>&lt;1</td>
<td>66</td>
<td>&lt;2</td>
</tr>
<tr>
<td>3000 Pishvaz Code</td>
<td>(12, 62)</td>
<td>27.9</td>
<td>&lt;1</td>
<td>198</td>
<td>&lt;3</td>
</tr>
<tr>
<td>24Game</td>
<td>(4, 16)</td>
<td>5.7</td>
<td>&lt;1</td>
<td>192</td>
<td>&lt;3</td>
</tr>
<tr>
<td>Running Example</td>
<td>(6, 14)</td>
<td>4.9</td>
<td>&lt;1</td>
<td>48</td>
<td>&lt;2</td>
</tr>
<tr>
<td>Gestures Builder</td>
<td>(4, 14)</td>
<td>3.53</td>
<td>&lt;1</td>
<td>72</td>
<td>&lt;2</td>
</tr>
<tr>
<td>Tomdroid</td>
<td>(9, 65)</td>
<td>24</td>
<td>&lt;1</td>
<td>198</td>
<td>&lt;3</td>
</tr>
<tr>
<td>FBReader</td>
<td>(42, 212)</td>
<td>256</td>
<td>&lt;2</td>
<td>1323</td>
<td>&lt;4</td>
</tr>
</tbody>
</table>

Regarding our second evaluation’s research question, i.e., RQ2, we claim that our proposed ranking metric is better than existing ones. To prove this, we compared the results of applying our ranking metric with the results of using the Tarantula and Jaccard metrics as two powerful ranking metrics. This comparison is based on the following four factors:

1. **Detecting the Main Causes of Exceptions (DMC):** Is the ranking metric capable of identifying the main causes of occurred exceptions?

2. **Incorrectly Detecting the Main Causes of Exceptions (IDMC):** Does the ranking metric detect some lines of the application’s source code as the most probable causes of occurred exceptions when they are not?

3. **Detecting Unrelated Lines of Code (DULOC):** In each application, there are often many lines of code that are unrelated to occurred exceptions. The DULOC factor considers whether the ranking metric can detect unrelated lines?

4. **Incorrect Ranking of Alternative Lines (IRAL):** In addition to detecting the main causes of an exception, a powerful ranking metric should also correctly rank other lines of the application’s source code with respect to their probability of being related to an occurred exception.

Table 10 compares the results of applying our ranking metric with the results of using the Tarantula and Jaccard metrics on the case studies from Table 7 that contain some exceptions. As can be seen in Table 10 our ranking metric outperforms the Tarantula and Jaccard metrics in ranking suspicious lines of code. More specifically, our metric is capable of detecting the main causes of exceptions (i.e., DMC) in all the cases, while others cannot necessarily do it. Moreover, it is a big weakness for a ranking metric to detect incorrect lines as the main causes of an exception (i.e., the IDMC factor). However, the results in Table 10 show that the Tarantula and Jaccard metrics suffer from this problem in some case studies. Moreover, as can be seen in Table 10 our proposed metric outperforms other metrics in DULOC and IRAL factors as well.

### 5.4 Threats to validity

There are several factors that may potentially affect the validity of the results of this experiment. This section provides a description of these factors.
Table 10: Comparing the results of applying our proposed ranking metric with the results of using the Tarantula and Jaccard metrics. P: Proposed metric; T: Tarantula metric; J: Jaccard metric; DMC: Detecting main reason; IDMC: Incorrect detection as main reason; DULOC: Detecting unrelated lines; IRAL: Incorrect ranking of alternative lines.

<table>
<thead>
<tr>
<th>Application</th>
<th>DMC</th>
<th>IDMC</th>
<th>DULOC</th>
<th>IRAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Example</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Calculator</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Gallery</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>3000 Pishvaz Code</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Tomdroid</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>RBReader</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>

5.4.1 Internal Validity

Internal validity relates to the extent to which the design and analysis may have been compromised by the existence of confounding variables and other unexpected sources of bias [5]. The main threat to internal validity is the list of events generated for each object before the execution of the proposed approach. An incorrect list of events can cause uncertainty in the final results. This threat is minimized by providing the user the ability to extend or change the lists of events. Another threat to internal validity is related to the test cases generated by the test case generator module. More specifically, if the generated test cases do not cover all the program statements, there would be a chance of error in the results. This threat is minimized by adapting a well-established approach for generating test cases that was published in [9].

5.4.2 External Validity

External validity relates to the extent to which the research questions capture the objectives of the research and the extent to which any conclusions can be generalized [5]. The main threat to external validity is that whether the proposed approach can be generalized to localize other kinds of exceptions and faults that were not considered in our evaluations. In this paper, we did not claim that we can localize all kinds of faults. For example, energy-related faults are out of the scope of this paper and there are a large body of work (e.g., [6, 7, 8]) that aim to address this kind of faults. Regarding the exception faults which are the target of this paper, we tried in our evaluations to localize various kinds of exceptions in different applications. However, there is still room for further evaluations with more sample applications and exception faults. However, as our evaluations indicate, if the source code of an application and the traces of test cases are available, one can use our proposed approach to detect suspicious lines of code. Another threat to external validity relates to selection of sample applications that directly influence the results. We minimized this threat by selecting open source applications from widely-used Android stores.
5.4.3 Construct Validity

The test of construct validity questions whether the theoretical constructs are interpreted and measured correctly [5].

For this experiment, the main threat to construct validity is that whether the location of faults that the proposed approach detects are correctly interpreted. In other words, the actual location of faults might be different from the detected location of faults, and we incorrectly accepted the results of the approach. This threat was addressed by manually localizing the exceptions before applying the approach. In addition, for a number of sample applications, we ourselves injected the faults and thus, we knew the precise location of faults.

5.4.4 Replicability

Replicability validity checks that by rerunning the approach on the same inputs, whether or not we get the same results [5].

We provided all the details of our exception fault localization approach as well as the setup of our evaluations, including the data collection and data analysis procedures. The sample applications used in this study are open-source and are also available online at [1]. Moreover, the prototype implementation of our approach can be downloaded from [1]. Consequently, it should be possible to replicate the results.

6 Discussions

6.1 Strengths and Weaknesses of the Approach

The suggested approach has the ability to localize multiple exception faults. In addition, because of using the value pattern and the backward static slicing scores in its calculations, it can detect related lines of code to the line in which the exception has actually occurred. This makes the results more precise (see Table 8 and Table 10). Furthermore, as pointed out in Section 3.2.2, redundant test cases can decrease the accuracy of results. Hence, the suggested approach detects and eliminates them to improve the reliability of results.

Nevertheless, the proposed approach has some drawbacks as well. Most importantly, it relies on the generated test cases to localize occurred exception faults. Thus, if the generated test cases do not cover all the application’s program statements, the fault might be in those uncovered lines, and hence, the approach would not detect it. The suggested approach is also relatively slow like almost any other similar approaches because running test cases over the application is being executed in Android simulators.

6.2 Proposed Ranking Metric

The existing ranking metrics like the Tarantula and the Jaccard only use the execution frequency of passed and failed test cases, and hence, the results can be inaccurate in some cases (see Section 5). Our ranking metric overcomes this issue by using two other scores: the value pattern score and the backward static slicing score. A difference between the suggested ranking metric and existing ones is that existing ones are only based on traces of test cases without analyzing the source code itself. Nevertheless, the suggested approach requires to analyze the application’s source code as well to calculate the backward static slicing score. Although the backward static slicing score makes the results more accurate, the need for analyzing the source code can be considered as a weakness for the suggested ranking metric as well.
7 Related Work

Originally, fault localization was performed manually. This means that when an error occurs, a human agent should manually analyze the source code and the error report to localize that fault. However, manual approaches are time-consuming, require a lot of effort, and are error-prone because of human involvements. To address these challenges, semi-automated (e.g., [7, 24, 45, 29]) and fully-automated (e.g., [32, 11, 25, 6, 8]) fault localization approaches have been proposed in literature. These approaches can be furthermore classified into static, dynamic, or hybrid ones. Static approaches (e.g., [36, 26, 6, 7]) only work with the source code of an application without taking into account its runtime information. On the other hand, dynamic approaches (e.g., [32, 11, 25]) only use the information collected at the runtime of an application and do not consider its source code. Nonetheless, hybrid approaches (e.g., [4, 50, 28, 23]) use both static and dynamic information.

This section provides an overview of automated fault localization approaches and compares them with our proposed approach for localizing exception faults in Android applications. In particular, we consider related work in three categories: (i) the fault localization approaches which have been introduced for traditional applications, (ii) the approaches that have been recently introduced particularly for smart mobile applications, and (iii) the approaches that specifically localize exception faults which are of particular interest in this paper. However, different approaches may localize various kinds of faults as will be discussed in this section.

7.1 Fault Localization Approaches for Traditional Applications

Wong and Debroy in [38] categorize fault localization approaches to slice-based, spectrum-based, statistics-based, state-based, machine learning-based, model-based, and data mining-based approaches. So, to be compatible with this categorization, in the following, first we provide an overview of these categories of approaches.

The idea of slice-based approaches, like [17, 37], is that if a test case fails because of an incorrect variable value at a statement, then the cause of that fault should be identified in the program slice associated with that variable-statement pair.

Spectrum-based techniques, such as [30, 19], compare the program spectrum of passed and failed test cases to localize faults. A program spectrum records the execution information or the dynamic behavior of an application in certain situations like the execution information for conditional branches.

Statistics-based methods (e.g., [4, 13, 3]) try to rank program statements of being the cause of a fault using the statistical metrics like Tarantula and Jaccard. Similar to our proposed approach, these metrics are calculated by running a set of test cases.

In state-based approaches, like [47], the program states are recorded repeatedly, and when a fault occurs, the program states before and after the fault are compared to localize that fault. These approaches, a program state is defined as a collection of program variable assignments in a specific point of execution. By this definition, a program may have innumerable states which can make the fault localization process challenging.

In machine learning-based techniques (e.g., [39]), the problem at hand can be expressed as trying to learn the location of a fault based on input data such as the statements coverage. These approaches usually create a model (e.g., a neural network) and train it with a plenty of failed and passed test cases. When a fault occurs, for each line of the code, a test case is generated and evaluated over that model. The model then evaluates the input test case with the test cases learned so far, and detects the cause of the fault. The main strength of this kind of techniques is that they are robust and adaptive, and the generated models can become stronger by feeding...
new test cases. Model-based techniques, such as [41, 50], are amongst the most popular fault localization approaches. A model is a behavioral or structural representation of the program. Different model-based techniques apply various analysis over the generated models to localize occurred faults. For example, the fault localization process can be interpreted as finding a specific kind of path in a graph.

Data mining techniques can unveil hidden patterns in samples of data (particularly in large volumes of data) that may not be recognized by the human analysis alone. Data mining-based approaches for fault localization, like [34, 29], abstract the software fault localization problem to a data mining problem, specially when we have a huge number of lines of code. For example, we may want to identify the patterns of program statements execution that lead to a program failure. For this purpose, these approaches may use different data mining techniques, such as VSM, UM, LSA, LDA, and CBDM to localize faults.

Bug repositories keep historical information about a program including the previous faults and their solutions. When a fault occurs, the history of the program could be analyzed and based on previous similar faults, the new fault could be localized. We call this category of approaches as history-based techniques. The main characteristic of this kind of approaches, like [10, 33], is that they are highly dependent on the previously reported faults and if a similar fault is not reported before, they would not work.

Sometimes, the results of a combination of the approaches mentioned above can be stronger than each one separately. For example, a fault localization approach may use both of the spectrum-based and history-based techniques together to localize faults. We name this kind of approaches as combined techniques and examples of them include [42, 48].

The fault localization process in most of the approaches mentioned above is highly dependent on the input test cases. To mitigate this, Xuan et al. in [45] introduce a technique named test case purification which its goal is to break each failed test case into more atomic test cases such that each atomic test case includes only one assertion. They show that if their purification process be used during the fault localization, then more accurate and more relevant program statements would be returned as the reasons of faults.

As described in Section 2, Android applications are event-driven, i.e., they respond to user and/or system generated actions. Therefore, their test process is different from traditional applications. Consequently, many of the fault localization approaches introduced in this section are not applicable to smart mobile applications in general, and Android applications in particular. Moreover, even if some of them might be applicable for smart mobile applications, their implemented tools are not necessarily able to detect faults and exceptions that are specific to smart mobile applications like ActivityNotFoundException in Android applications. However, our proposed approach and prototype implementation look for Android specific exceptions.

### 7.2 Fault Localization Approaches for Smart Mobile Applications

This section provides an overview of existing fault localization techniques introduced in literature for smart mobile applications.

As mentioned before in Section 7.1, the advantage of model-based fault localization approaches is that the generated models are easier to analyze than the source codes themselves. Hence, a number of model-based fault localization approaches have been introduced for smart mobile applications as well. For instance, Takala et al. in [32] use a tool named TEMA to extract the events of an Android application. The extracted events are then used to generates a Finite State Machine (FSM) for the application. Next, GUI faults are localized with the help of this FSM. In another work, Yang et al. [44] implement a tool, named ORBIT, that test the GUI of an Android
application in a two step process. First, it analyzes the source code statically to extract the set of events supported by the GUI. It then dynamically exercises those events on the application to obtain a behavioral model of the application. This model can be analyzed next to localize faults. Data flow analysis-based approaches, like [36, 26, 6, 7], analyze the data flow graph of an application to localize various kinds of faults. For example, Vekris et al. in [36] and Pathak et al. in [26] did a similar work to detect energy faults using some defined policies. More specifically, they look for paths in the data flow graph which acquire a resource at some point of time and do not release it later. Egele et al. in [6] consider the privacy threats that iOS applications pose to users. In particular, they provide a tool, named PiOS, that allows developers to analyze the data flow graphs of iOS applications for possible leaks of sensitive information from a mobile device to third parties. Another work presents AndroidLeaks [7], a static analysis framework for automatically finding potential leaks of sensitive information in Android applications. For this purpose, it benefits from data flow analysis to see if data from a source method reaches a sink method.

Besides generating models out of programs, there are a number of techniques that directly work with the source code itself. For instance, Hu et al. [11] and Gottschalk et al. [8] propose techniques that map the fault localization problem to the issue of finding the pieces of the source code which follow some defined special patterns. More specifically, [11] performs a bug mining study to identify the patterns of GUI bugs that are quite common. On the other hand, [8] looks for energy-wasting patterns. Those pieces of code that follow the defined patterns are marked as faulty and the rest of the code is known as fault free. The pattern-based approaches are often fast. Nevertheless, the problem with them is that they are unable to detect those faults that do not follow the defined patterns.

In addition to fault localization techniques that use various kinds of models, exercise the application’s test cases, and/or analyze the application’s source code, state-based techniques that localize faults by comparing different states of the application are also introduced in literature. For instance, Pathak et al. [25] propose a state-based approach to localize energy faults. In their work, the state of the application is recorded periodically, and when an energy bug happens, the fault is localized by comparing the current state of the application with the previous ones. Unlike our proposed approach that focus on exception faults, most of the existing techniques discussed above pay attention to user-interested faults such as profile leakages, GUI faults, or extremely energy usages. However, these types of faults often do not stop the application from working and mainly waste the resources of the smart mobile device. In some cases, these faults can be avoided by allocating more resources to the application. Nevertheless, Our approach focuses on exceptions which are a very common type of faults. Exceptions are important since they may stop the whole application and they cannot be necessarily avoided by allocating more resources to the application.

Recently, Moran et al. [20] introduced a tool named CrashScope. This tool explores a given Android application with the goal of triggering crashes. For this purpose, it uses systematic input generation according to several strategies informed by static and dynamic analysis. When a crash happens, the tool produces a crash report that includes screenshots, detailed crash reproduction steps, the captured exception stack trace, and a script that automatically regenerates the crash on a target device. However, unlike our approach that localizes exception faults, this tool only produces augmented crash reports and does not localize them.

Finally, [16] presents the results of an empirical study conducted to understand actual developers’ practices for detecting and fixing performance bottlenecks in mobile applications. In general, it indicates that developers heavily depend on user reviews and manual execution of the applications for detecting performance bugs. So, this work motivates the need for highly automated tools that can answer this challenging task.
7.3 Exception Localization Approaches

There has been studies in literature which localize exception faults in programs. In this section, we provide an overview of these approaches which are of our particular interest.

Barr et al. [2] use symbolic execution of programs to localize the `FloatingPointException` in C/C++ and Fortran programs. In another work, Payet et al. [27] analyze the source code of an Android application statically to localize `NullPointerException` faults. It is clear that static analysis is not strong enough for localizing unhandled exceptions because of the dynamic nature of exceptions, and hence, this method is not a general approach.

Hu et al. [11] categorize faults in Android applications based on studying ten popular forums. Unhandled exceptions is one of their proposed categories, although they do not discuss how to address them.

Sinha et al. [31] introduced a hybrid method that uses both dynamic analysis on stack trace information and static backward data-flow analysis to localize three specific kinds of exceptions: `NullPointerException`, `Arithmetic`, and `Type` exceptions. However, this method cannot localize other kinds of exceptions. In a similar approach, Jiang et al. [12] use program slicing, backward data flow analysis, and stack trace information to localize runtime exceptions.

To summarize, unlike our proposed approach, all of the techniques introduced above try to localize some specific kinds of exceptions and they are often unable to localize Android specific exceptions like `ActivityNotFoundException`.

8 Conclusions and Future Work

This paper presented a new approach for localizing exception faults in Android applications. Our approach is a hybrid approach that statically analyzes an Android application’s source code as well as the execution traces of its test cases. To rank lines of source code based on their probability of being faulty, we proposed a statistical ranking metric that uses the following three scores: (i) the test case score that uses test cases’ traces and gives each line a score based on its participation in test cases’ execution; (ii) the value pattern score that tries to detect unrelated lines of code by examining the differences between passed and failed test cases for each line; and (iii) the backward static slicing score which analyzes the application’s source code to remove unrelated lines from the list of suspicious lines. We evaluated our approach over nine Android applications of different sizes with different number of various exceptions. In all of our case studies, our technique was able to correctly detect the causes of occurred exceptions. Our experimental evaluations also indicated that our ranking metric outperforms two of the widely used Tarantula and Jaccard ranking metrics.

In future, we plan to extend our technique to support other mobile platforms like iOS and Windows. In addition, localizing other types of faults (e.g., user interface and security faults) will be considered in future works.

References


About the Authors

**Hamed Mirzaei** holds a master’s degree from the Department of Computer Engineering at the Sharif University of Technology. His primary research interest is in the areas of reverse engineering and re-engineering of software systems and mining software repositories. Contact him at hamedmirzaei@ce.sharif.edu.

**Abbas Heydarnoori** is an assistant professor in the Department of Computer Engineering at the Sharif University of Technology. Before, he was a post-doctoral fellow at the University of Lugano, Switzerland. Abbas did his PhD in the School of Computer Science at the University of Waterloo, Canada. His research interests focus on reverse engineering and re-engineering of software systems, mining software repositories, and recommendation systems in software engineering. Contact him at heydarnoori@sharif.edu.

9 Captions

Listing 1 MainActivity.java

Listing 2 SubActivity1.java

Listing 3 SubActivity2.java

Listing 4 SubActivity3.java

Fig. 1 The proposed exception fault localization approach for Android applications.

Table 1 The control changer events of the running example presented in Section 2.

Table 2 The set of test cases generated for the running example presented in Section 2. Test cases which their event sequences are a prefix of another test case are not mentioned here because they are redundant. Passed test cases $TC_9$, $TC_{14}$, $TC_{19}$ and $TC_{22}$ are not considered in
the evaluation phase because they are irrelevant to the faulty activity SubActivity1.

Fig. 3 Clustering the test cases provided in Table 2 with respect to occurred exceptions.

Table 3 The value patterns for the variable num of SubActivity1.java. PVP: Passed Value Pattern; FVP: Failed Value Pattern.

Table 4 Different values of the a parameter used to calculate the probability of being faulty for some lines of the Gallery case study.

Table 5 Ranking metrics results for Exception1 in the running example (i.e., NullPointerException). PTCs: Passed Test Cases; FTCs: Failed Test Cases.

Table 6 Ranking metrics results for Exception2 in the running example (i.e., ResourceNotFoundException). PTCs: Passed Test Cases; FTCs: Failed Test Cases.

Table 7 The selected case studies for our evaluations. LOC: Line Of Code; NOO: Number Of Objects; NOA: Number Of All Exceptions; NOU: Number Of Unique Exceptions.

Table 8 The graph size and the execution time of applying our approach on different case studies. EGT: Extracting Graph Time; GTCT: Generating Test Cases Time; ETCT: Executing Test Cases Time; RT: Ranking Time; All times are in minutes.

Table 9 Comparing the results of applying our proposed ranking metric with the results of using the Tarantula and Jaccard metrics. P: Proposed metric; T: Tarantula metric; J: Jaccard metric; DMC: Detecting main reason; IDMC: Incorrect detection as main reason; DULOC: Detecting unrelated lines; IRAL: Incorrect ranking of alternative lines.

Table 8 Evaluation results.

10 Figures and Tables