Scientia Iranica D (2019) 26(3), 1567–1588

Sharif University of Technology
Scientia Iranica
Transactions D: Computer Science & Engineering and Electrical Engineering
http://scientiairanica.sharif.edu

Localizing exception faults in Android applications

H. Mirzaei and A. Heydarnoori*

Department of Computer Engineering, Sharif University of Technology, Tehran, Iran.

Received 8 August 2017; received in revised form 22 April 2018; accepted 24 September 2018

KEYWORDS
Fault localization;
Unhandled exceptions;
Exception faults;
Android applications.

Abstract. In software programs, most of the time, there is a chance for occurrence of faults in general, and exception faults in particular. Localizing those pieces of code that are responsible for a particular fault is one of the most complicated tasks, and it can produce 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 such as context-awareness, use of sensors, being executable on various mobile devices, limited hardware resources, etc. To this aim, in this paper, a semi-automated hybrid method is introduced that combines static and dynamic analyses to localize exception faults in Android applications. Our evaluations of nine open source Android applications of different sizes with various exceptions show that the technique proposed in this paper can correctly identify root causes of the occurred exceptions. These results indicate that our proposed approach is effective in practice in localizing exception faults in Android applications.

1. Introduction

The goal of software testing is to make sure that a program does what it was designed to do; conversely, it does not do anything unintended [1]. Due to the diversity of application configurations, platforms, and inputs, software testing is typically an unpleasant, complicated, difficult to automate, and costly process [2]. 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 [3,4]. *Fault localization* is the process of identifying the pieces of an application's source code responsible for the occurrence of 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 [5]. In recent years, *smart mobile devices*, 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 [6]. They are also *context-aware*, meaning that they can behave differently in different situations and locations [6]. Additionally, Mobile applications can use various sensors of mobile devices, can be executed on different smart mobile devices with various properties, and should be executed and tested with limited hardware resources available on smart mobile devices, new technologies are used in their developments, etc. [7]. All these special characteristics force mobile application developers to look for new fault localization methods [7,8].

To address the above issue, a number of approaches have been proposed in related literature

*. *Corresponding author.*
E-mail addresses: hamedmirzaei@ce.sharif.edu (H. Mirzaei); heydarnoori@sharif.edu (A. Heydarnoori)

doi: 10.24200/sci.2018.4966.1015

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. [9] proposed an approach to detect profile leakages in iOS applications. On the other hand, Gibler et al. [10] used a different approach for finding profile leakages in Android applications. To detect energy-related faults, Vekris et al. [11] defined some energy policies and examined Android applications to see whether or not those policies have been followed. In a different work, Pathak et al. [12] used the data-flow analysis to detect no-sleep energy faults, which keep the phone awake all the time. In another work, Gottschalk et al. [13] defined a number of energy-related code smells and attempted to remove them by re-engineering the source code. The *Graphical User Interface (GUI)* is an important part of any event-driven application. Some approaches, such as the ones done by Takala et al. [14], Yang et al. [15], and Hu et al. [16], focus on detecting GUI-related faults in mobile applications. More specifically, they attempt 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 too; however, we concentrate on what is occurring beyond the GUI and in the source code itself, not just the GUI. It is implied 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, occurring in specific unwanted situations and forcing the application to stop. Of note, these faults are very important since they can crash the whole application. To the best of our knowledge, no approaches have been proposed in the literature to localize this kind of faults in Android applications. To address this issue, in this paper, a semi-automated hybrid method is introduced that combines the dynamic and static analyses 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 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, lines of the application's source code based on their relevance to the occurrence of 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 is recorded by the approach, and the next test case is 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, three different scores are used: 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 is executed more in the failed test cases than that in the passed test cases, it is more likely to be faulty. To calculate this score, a combination of *Tarantula* [17] and *Jaccard* [18] metrics is used. The value pattern score is used to detect lines of the source code that are not related to the occurrence of that exception. Hence, this score helps to reduce the search space by removing unrelated statements; thus, fault localization can 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 the occurrence of that exception. This score reduces the search space by choosing related program statements.

The our proposed approach has been implemented as a tool for Java and used to localize various exceptions in nine open source Android applications of different sizes. Our evaluations indicate that the proposed 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 include (i) a new ranking metric to rank lines of the source code based on their relevance to the occurrence of 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 used throughout this paper. Then, Section 3 introduces the 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 with various exceptions. Section 6 provides discussion on 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, an example Android program is provided in order to clarify the problem that our proposed approach aims to tackle. This example is then used throughout this paper. However, first, the structure of an Android application is briefly explained 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 (http://developer. android.com):

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 have been downloaded to the device and is available for them to use. Thus, this is the broadcast receiver who will intercept this communication and 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 [19]. 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; 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 addition, the user can trigger any possible sequence of events and this would lead testers to a huge number of test scenarios.

Now that the reader is familiar with the structure of Android applications, consider an Android application with the following four activities: the running example: `MainActivity` (Listing 1),

```java
public class MainActivity extends AppCompatActivity {
    Button button1, button2;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main_activity);
        addListenerOnButton();
    }
    public void addListenerOnButton() {
        final Context context = this;
        button1 = (Button) findViewById(R.id.button1);
        button2 = (Button) findViewById(R.id.button2);
        button1.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View arg0) {
                Intent intent = new Intent(context, SubActivity1.class);
                startActivity(intent);
            }
        });
        button1.setOnLongClickListener(new View.OnLongClickListener() {
            public boolean onLongClick(View v) {
                Intent intent = new Intent(context, SubActivity2.class);
                startActivity(intent);
                return true;
            }
        });
        button2.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View arg0) {
                Intent intent = new Intent(context, SubActivity1.class);
                startActivity(intent);
            }
        });
        button2.setOnLongClickListener(new View.OnLongClickListener() {
            public boolean onLongClick(View v) {
                return true;
            }
        });
    }
}
```

Listing 1. MainActivity.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.setMovementMethod(LinkMovementMethod.getInstance());
31.        String text = "<a href='http://www.google.com'>Google</a>";
32.        textView2.setText(Html.fromHtml(text));
33.        button3.setOnClickListener(new View.OnClickListener() {
34.            @Override
35.            public void onClick(View arg0) {
36.                Intent intent = new Intent(context, SubActivity2.class);
37.                startActivity(intent);
38.            }
39.        });
40.        button3.setOnLongClickListener(new View.OnLongClickListener() {
41.            public boolean onLongClick(View v) {
42.                            num = 7;
43.                View view = null;
44.                if (num >= 8) {
45.                    view = findViewById(R.id.button3); //this line never executes
46.                }
47.                System.out.println(view.getWidth() + " and " + num); //it will throw exception
48.                return true;
49.            }
50.        });
51.        button4.setOnClickListener(new View.OnClickListener() {
52.            @Override
53.            public void onClick(View arg0) {
54.                int tempInteger = 31073;
55.                String tempString = "51268";
56.                XmlResourceParser a = Resources.getSystem().getLayout(10); //it will throw exception because
     the resource does not exist
57.            }
58.        });
59.    }
60.}
```

Listing 2. SubActivity1.java.

SubActivity1 (Listing 2), SubActivity2 (Listing 3), and SubActivity3 (Listing 4). MainActivity is the main activity that includes two *buttons* and works normally without any exceptions. However, SubActivity1 includes a *textView* and two *buttons*, and will throw a NullPointerException on the *long click* event of the first button and a ResourceNotFoundException 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* ($Exception_1$ in the rest of this paper) in SubActivity1 (Listing 2) occurs because of the following reasons: (i) an invalid assignment to variable num in *line 42* of Listing 2; (ii) an invalid assignment to variable view in *line 43*; (iii) the missing of the else statement in the if block of *line 44*; finally, (iv) the execution of *line 47* throws the exception; ResourceNotFoundException ($Exception_2$ 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 the 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, an overview is provided first. The approach

```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 3. SubActivity2.java.

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

Listing 4. SubActivity3.java.

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 responsible for raising exceptions. From the user's perspective, the proposed approach has three phases as depicted in Figure 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 is 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 is 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 the probability of causing the exception fault. In the following, the details of these phases are discussed by means of the example provided in Section 2.

3.2. Approach details

The proposed approach is a hybrid approach that statically analyzes the source code of an Android application and dynamically executes its test cases to locate those program statements responsible for the occurrence of 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 Figure 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 such as *click* and *long click* can be generated. Next, for

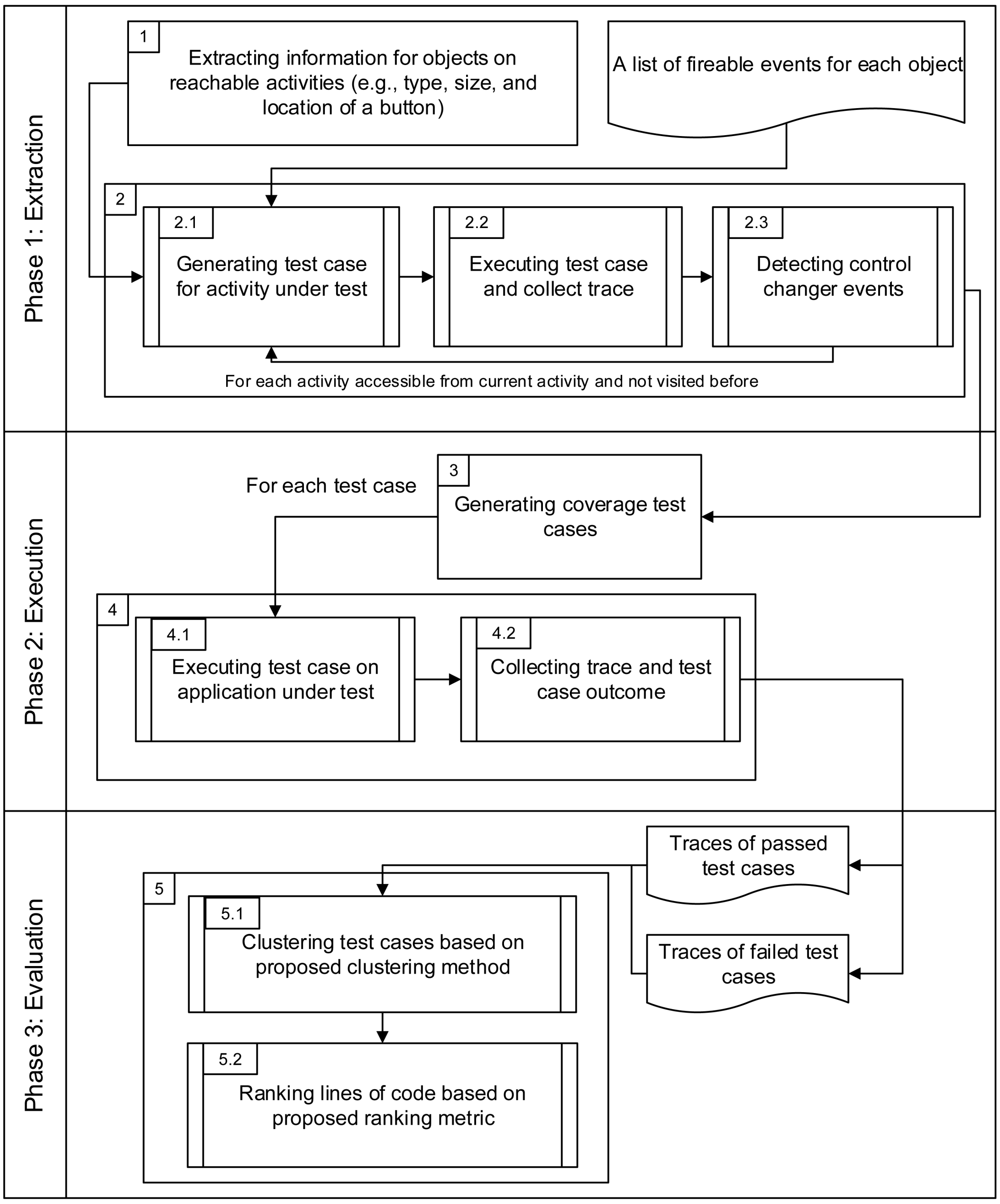
Phase 1: Extraction
1
Extracting information for objects on reachable activities (e.g., type, size, and location of a button)
A list of fireable events for each object
2
2.1
Generating test case for activity under test
2.2
Executing test case and collect trace
2.3
Detecting control changer events
For each activity accessible from current activity and not visited before
Phase 2: Execution
For each test case
3
Generating coverage test cases
4
4.1
Executing test case on application under test
4.2
Collecting trace and test case outcome
Phase 3: Evaluation
Traces of passed test cases
Traces of failed test cases
5
5.1
Clustering test cases based on proposed clustering method
5.2
Ranking lines of code based on proposed ranking metric

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

each activity of the application which is *reachable* from the main activity, all the information about its objects is extracted (e.g., information like *size*, *unique-id*, and *location* for the *button* object). Of note, activity B is *reachable* from activity A if there exists at least one sequence of events starting from activity A and ending at activity B.

To detect control changer events, for each activity, a test case that 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 investigating 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), *AndroidViewClient (https://github.com/dtmilano/AndroidViewClient)* (*AVC*) library of the *Python* programming language is used. Table 1 lists control changer events for our running example. We have basically two kinds of control changer events which are of 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

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

Event number	Source	Destination	Description
$event_1$	MainActivity	SubActivity1	Click on *button1*
$event_2$	MainActivity	SubActivity2	Long click on *button1*
$event_3$	MainActivity	SubActivity1	Click on *button2*
$event_4$	MainActivity	MainActivity	Long click on *button2*
$event_5$	MainActivity	MainActivity	Click on *textView1*
$event_6$	SubActivity1	SubActivity2	Click on *button3*
$event_7$	SubActivity1	NullPointerException	Long click on *button3*
$event_8$	SubActivity1	ResourceNotFoundException	Click on *button4*
$event_9$	SubActivity1	SubActivity1	Long click on *button4*
$event_{10}$	SubActivity1	InvalidActivity	Click on *textView2*
$event_{11}$	SubActivity2	SubActivity2	Click on *button5*
$event_{12}$	SubActivity2	InvalidActivity	Long click on *button5*
$event_{13}$	SubActivity2	SubActivity3	Click on *textView3*
$event_{14}$	SubActivity3	SubActivity3	Click on *textView4*

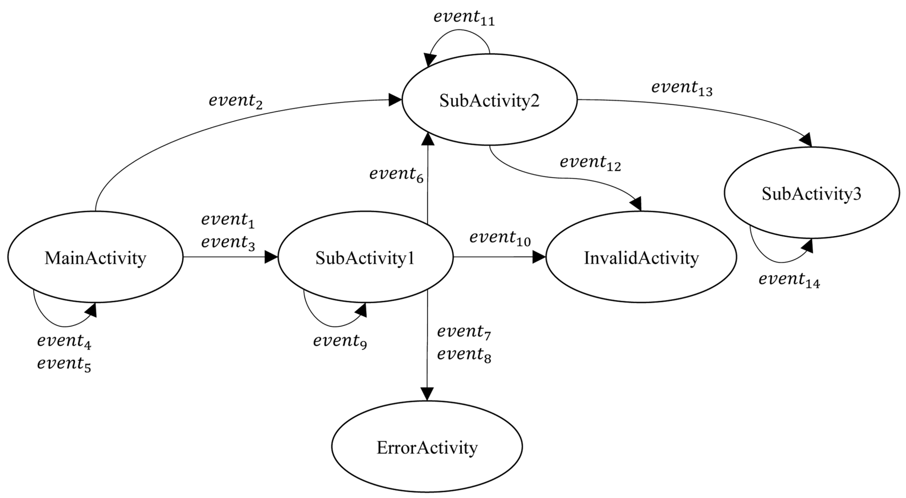

Figure 2. The labeled graph generated for the running example presented in Section 2.

our approach, for the sake of simplicity, it is assumed that there exists an activity named *InvalidActivity* such 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 as well as their source and destination activities. To accomplish the task of generating test cases in the next phase of our proposed approach, a labeled graph $G(V, L, E)$ (e.g., Figure 2) is created in which:

- V: The set of nodes such that each node is an activity of set of reachable activities from the *Main-Activity*, *InvalidActivity*, and *ErrorActivity*;
- L: The set of labels that denote the fireable events of the application;
- E: The set of edges. Such that an edge from activity A to activity B with a label l meaning that the firing of event l in activity A transfers the applications' control to activity B.

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. [20] carried out a statistical analysis, and showed that *redundant test cases* would decrease the precision of results. A

Table 2. The set of test cases generated for the running example presented in Section 2. Test cases whose event sequences are a prefix of another test case, which 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.

Status	**TCN**	**The sequence of events**
Failed	TC_1	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
	TC_2	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
	TC_3	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
	TC_4	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
	TC_5	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
	TC_6	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
	TC_7	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
	TC_8	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
Passed	TC_9	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
	TC_{10}	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
	TC_{11}	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
	TC_{12}	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
	TC_{13}	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
	TC_{14}	MainActivity $\rightarrow event_5 \rightarrow$ MainActivity $\rightarrow event_4 \rightarrow$ MainActivity $\rightarrow event_2 \rightarrow$ SubActivity2 $\rightarrow event_{11} \rightarrow$ SubActivity2 $\rightarrow event_{12} \rightarrow$ InvalidActivity
	TC_{15}	MainActivity $\rightarrow event_5 \rightarrow$ MainActivity $\rightarrow event_4 \rightarrow$ 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
	TC_{16}	MainActivity $\rightarrow event_5 \rightarrow$ MainActivity $\rightarrow event_4 \rightarrow$ MainActivity $\rightarrow event_3 \rightarrow$ SubActivity1 $\rightarrow event_9 \rightarrow$ SubActivity1 $\rightarrow event_{10} \rightarrow$ InvalidActivity
	TC_{17}	MainActivity $\rightarrow event_5 \rightarrow$ MainActivity $\rightarrow event_4 \rightarrow$ 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
	TC_{18}	MainActivity $\rightarrow event_5 \rightarrow$ MainActivity $\rightarrow event_4 \rightarrow$ MainActivity $\rightarrow event_1 \rightarrow$ SubActivity1 $\rightarrow event_9 \rightarrow$ SubActivity1 $\rightarrow event_{10} \rightarrow$ InvalidActivity
	TC_{19}	MainActivity $\rightarrow event_4 \rightarrow$ MainActivity $\rightarrow event_5 \rightarrow$ MainActivity $\rightarrow event_2 \rightarrow$ SubActivity2 $\rightarrow event_{11} \rightarrow$ SubActivity2 $\rightarrow event_{13} \rightarrow$ SubActivity3 $\rightarrow event_{14} \rightarrow$ SubActivity3
	TC_{20}	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 $\rightarrow event_{13} \rightarrow$ SubActivity3 $\rightarrow event_{14} \rightarrow$ SubActivity3
	TC_{21}	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 $\rightarrow event_{13} \rightarrow$ SubActivity3 $\rightarrow event_{14} \rightarrow$ SubActivity3
	TC_{22}	MainActivity $\rightarrow event_5 \rightarrow$ MainActivity $\rightarrow event_4 \rightarrow$ MainActivity $\rightarrow event_2 \rightarrow$ SubActivity2 $\rightarrow event_{11} \rightarrow$ SubActivity2 $\rightarrow event_{13} \rightarrow$ SubActivity3 $\rightarrow event_{14} \rightarrow$ SubActivity3
	TC_{23}	MainActivity $\rightarrow event_5 \rightarrow$ MainActivity $\rightarrow event_4 \rightarrow$ MainActivity $\rightarrow event_3 \rightarrow$ SubActivity1 $\rightarrow event_9 \rightarrow$ SubActivity1 $\rightarrow event_6 \rightarrow$ SubActivity2 $\rightarrow event_{11} \rightarrow$ SubActivity2 $\rightarrow event_{13} \rightarrow$ SubActivity3 $\rightarrow event_{14} \rightarrow$ SubActivity3
	TC_{24}	MainActivity $\rightarrow event_5 \rightarrow$ MainActivity $\rightarrow event_4 \rightarrow$ MainActivity $\rightarrow event_1 \rightarrow$ SubActivity1 $\rightarrow event_9 \rightarrow$ SubActivity1 $\rightarrow event_6 \rightarrow$ SubActivity2 $\rightarrow event_{11} \rightarrow$ SubActivity2 $\rightarrow event_{13} \rightarrow$ SubActivity3 $\rightarrow event_{14} \rightarrow$ SubActivity3

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 will be biased towards the pieces of code that are visited during their execution. This can affect the precision of results; hence, redundant test cases are ignored 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 [21]. 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 of generator module, the approach presented in [21] is adapted. More specifically, a two-step process is performed to generate test cases: (i) generating all the possible passed and failed test cases; and (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., Figure 2) that end at a node other than *ErrorActivity*; the set of failed test cases are those that end at *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 produce 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; however, we accept it and try to overcome this issue in the future work;
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 may decrease the performance of the approach [20].

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), the *AVC* library of the *Python* programming language is used. For instance, Table 2 indicates whether each test case is 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 that uses the information gathered so far to rank lines of the application's source code based on the probability of being faulty. Our approach's precision is highly dependent on the coverage criterion which is used to generate test cases out of the generated graph. Our coverage criterion, as explained earlier, is not complete in the case of loop edges; 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. Figure 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. Of note, a test case is related to a file `A.java` if at least one of the program statements of that file is visited during the execution of that test case. Consequently, test cases TC_9, TC_{14}, TC_{19}, and TC_{22} will be pruned since they are irrelevant to `SubActivity1.java`. Next, the related failed test cases will be clustered based on the occurred exceptions. For example, test cases TC_1, TC_2, TC_3, and TC_4 will become a cluster related to $Exception_1$ (i.e., *Failed* $Cluster_1$), and test cases TC_5, TC_6, TC_7, and TC_8 will become another cluster related to $Exception_2$ (i.e., *Failed* $Cluster_2$). At the end, to localize each exception, the related passed test cases are analyzed, and the corresponding failed test cases are clustered. For example, to localize $Exception_1$ in our running example, *Passed* and *Failed* $Cluster_1$ test cases are analyzed. Note that, by this definition, a passed test case relates to many exceptions, while a failed test

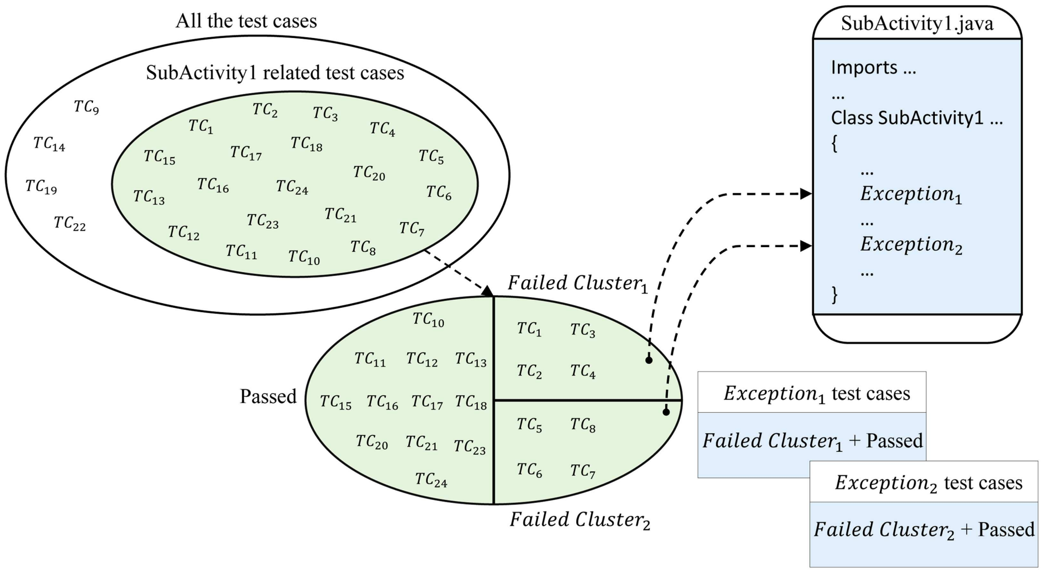

Figure 3. Clustering the test cases provided in Table 2 with respect to occurred exceptions.

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, the weaknesses of existing ranking metrics should be discussed.

Weaknesses of existing ranking metrics. The ranking metrics, *Tarantula* [17,22] and *Jaccard* [18], are widely used in related literature to rank suspicious lines of code. However, in our experiments, we noticed that these metrics could not necessarily rank the lines of code correctly in some situations. For instance, consider $Exception_2$ in our running example (i.e., *line 56* of `SubActivity1.java` (Listing 2)). Based on the generated test cases in Table 2 and the clustering results in Figure 3, we see that both of the ranking metrics *Tarantula* and *Jaccard* will not only return *line 56* (which is the real reason of occurring the exception), but also incorrectly introduce *line 54* and *line 55* as the main reasons of occurrence of the exception. This problem is that because these ranking metrics attempt to rank a program statement by only its execution frequency in test cases. As discussed below, we attempt to tackle this problem by proposing a ranking metric that employs other information, too.

Proposed ranking metric. As discussed above, existing ranking metrics, such as Tarantula and 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 such as the one occurred in $Exception_2$ of the running example (i.e., *line 56* of `SubActivity1.java` (Listing 2)). This problem emerged from irrelevant statements involved in the ranking process. However, based on our studies, 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 [23] to detect those statements that are relevant to faulty statements and to mark the others as irrelevant. The other factor, called 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 Eq. (1)) combines these two factors with the execution frequency of program statements in passed and failed test cases. The evaluations of our proposed ranking metric are provided in Section 5.

$$S(l) = VP(l) * (a * Prob(l) + (1 - a) * Slice(l)). \quad (1)$$

In the above formula, we have the following parameters:

- *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, too. $VP_{pass}(v)$ and $VP_{fail}(v)$ are defined as sets of value patterns generated for variable v from the passed and failed test cases, respectively. For a variable v, if $VP_{fail}(v)$ is a subset of $VP_{pass}(v)$, then variable v is marked as unrelated to the occurred exception. Based on the generated test cases for

Table 3. The value patterns for the variable num of SubActivity1.java.

Exception	**PVPs**[a]	**FVPs**[b]	**Description**
Exception$_1$	{<16,8> <24,8>}	{<16,8> <24,8> <42,7>}	The lines that use the variable `num` will get 1 for the *VP* score because *FVPs* are not a subset of *PVPs*
Exception$_2$	{<16,8> <24,8>}	{<16,8> <24,8>}	The variable `num` has no effect on the *VP* score because *FVPs* are a subset of *PVPs*

[a]PVP: Passed Value Pattern; [b]FVP: Failed Value Pattern.

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 *Exception*$_1$, the lines that use variable *num* will get 1 for the VP score since $VP_{fail}(num)$ is not a subset of $VP_{pass}(num)$. Nevertheless, for *Exception*$_2$, this variable is an unrelated variable and has no effect on VP score, since $VP_{fail}(num)$ is a subset of $VP_{pass}(num)$. In general, $VP(l)$ value is 0 if and only if all the variables used in line l are 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 [23]. Therefore, when an exception occurs in a line of code, it can be the result 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. Therefore, for each line l of the application's source code, if l is in the static slice of the faulty line, then $Slice(l)$ is 1, otherwise it would be 0. Therefore, in Listing 2, the backward static slice of *line 47* represents *lines 16, 24, 42, 43, 44,* and *47*; *Slice(l)* score for these lines is 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 criterion for test cases, for each exception, these frequencies are calculated on related test cases (not all of them). After analyzing the *Tarantula* and the *Jaccard* ranking metrics, we found that the *Tarantula and Jaccard* metrics respectively assigned a higher probability and a lower probability to some lines of code, which was beyond what was expected. Therefore, an average of these two (i.e., Eq. (2)) is used in our proposed ranking metric (i.e., Eq. (1)). In Eq. (2), $S_{Tarantula}(l)$ and $S_{Jaccard}(l)$ respectively denote the *Tarantula* and *Jaccard* ranking metrics of line l:

$$Prob(l) = (S_{Tarantula}(l) + S_{Jaccard}(l))/2. \qquad (2)$$

- *Parameter a:* Parameter a in Eq. (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 parameter a must be less than 0.5. To gain the best raking results, different values for this parameter were analyzed in our 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, however, 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

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

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

Table 5. Ranking metrics results for $Exception_1$ in the running example (i.e., `NullPointerException`).

Line numbers	PTCs[a]	FTCs[b]	$S_{Tarantula}(l)$	$S_{Jaccard}(l)$	$Prob(l)$	$Slice(l)$	$VP(l)$	$S(l)$	Rank
`MainActivity`: 9, 12, 13, 14, 17, 18, 19, 43	12	4	0.5	0.25	0.375	0	0	0	3
`MainActivity`: 23, 24	6	2	0.5	0.2	0.35	0	0	0	3
`MainActivity`: 37, 38	6	2	0.5	0.2	0.35	0	0	0	3
`SubActivity1`: 14, 15, 19, 20, 21, 25, 26, 27, 28, 29, 30, 31, 32	12	4	0.5	0.25	0.375	0	0	0	3
`SubActivity1`: 16, 24	12	4	0.5	0.25	0.375	1	1	0.875	2
`SubActivity1`: 36, 37	8	0	0	0	0	0	1	0	3
`SubActivity1`: 42, 43, 44 (false), 47	0	4	1	1	1	1	1	1	1
`SubActivity2`: 21, 22, 23	8	0	0	0	0	0	1	0	3
`SubActivity2`: 26, 27, 28	4	0	0	0	0	0	1	0	3
`SubActivity2`: 34, 35	4	0	0	0	0	0	1	0	3
`SubActivity3`: 7, 8	4	0	0	0	0	0	1	0	3
Others	-	-	-	-	-	-	-	0	3

[a]PTCs: Passed Test Cases; [b]FTCs: Failed Test Cases.

Table 6. Ranking metrics results for $Exception_2$ in the running example (i.e., `ResourceNotFoundException`).

Line Numbers	PTCs[a]	FTCs[b]	$S_{Tarantula}(l)$	$S_{Jaccard}(l)$	$Prob(l)$	$Slice(l)$	$VP(l)$	$S(l)$	Rank
`MainActivity`: 9, 12, 13, 14, 17, 18, 19, 43	12	4	0.5	0.25	0.375	0	0	0	3
`MainActivity`: 23, 24	6	2	0.5	0.2	0.35	0	0	0	3
`MainActivity`: 37, 38	6	2	0.5	0.2	0.35	0	0	0	3
`SubActivity1`: 14, 15, 19, 20, 21, 25, 26, 27, 28, 29, 30, 31, 32	12	4	0.5	0.25	0.375	0	0	0	3
`SubActivity1`: 16, 24	12	4	0.5	0.25	0.375	0	0	0	3
`SubActivity1`: 36, 37	8	0	0	0	0	0	1	0	3
`SubActivity1`: 54, 55	0	4	1	1	1	0	1	0.2	2
`SubActivity1`: 56	0	4	1	1	1	1	1	1	1
`SubActivity2`: 21, 22, 23	8	0	0	0	0	0	1	0	3
`SubActivity2`: 26, 27, 28	4	0	0	0	0	0	1	0	3
`SubActivity2`: 34, 35	4	0	0	0	0	0	1	0	3
`SubActivity3`: 7, 8	4	0	0	0	0	0	1	0	3
Others	-	-	-	-	-	-	-	0	3

[a]PTCs: Passed Test Cases; [b]FTCs: Failed Test Cases.

of different values for parameter a of an example case study. Thus, the proposed experimental results indicated that $a = 0.2$ gave the best results.

After applying the proposed ranking metric to our running example, the results presented in Table 5 for $Exception_1$ and the results in Table 6 for $Exception_2$ were obtained. These results illustrate that *lines 42, 43, 44(false),* and *47* of `SubActivity1.java` (Listing 2) 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 *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, the *AVC* library of the *Python* programming language was used. Interested readers can refer to [24] to access the source codes of our implementations.

5. Evaluation

In this section, the evaluations of our proposed approach are presented for localizing exceptions in Android applications. In particular, the objectives, setup, and results of our evaluations are presented.

5.1. Evaluation objectives

In performed evaluations, we are in favor of answering the following research questions to evaluate the effectiveness of the proposed 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 which are responsible for the occurrence of 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 would work better than the Tarantula and Jaccard metrics in ranking suspicious lines of code that might be responsible for the occurrence of an exception fault.

5.2. Evaluation setup

To answer the research questions raised in Section 5.1, have been 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 published in Android markets such as *Google Play* (https://play.google.com) and *CafeBazaar* (https://cafebazaar.ir/). Published applications are often exception-free; hence, some exceptions are injected 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 in `ErrorActivity` in the labeled graph; 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, two case studies have been chosen, i.e., the *Calculator* and the *Running Example* (see Section 2), including 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, our proposed approach has been applied to the case studies presented in Table 7. For this purpose, our prototype implementations

Table 7. The selected case studies for our evaluations.

Case study	LOC[a]	NOO[b]	NOA[c]	NOU[d]	Exceptions
Calculator	110	10	2	2	Two `NumberFormatExceptions`
Tippy Tipper	312	39	0	0	None
Gallery	129	11	4	1	`ActivityNotFoundException`
3000 Pishvaz Code	2162	86	1	1	`ResourceNotFoundException`
24Game	228	15	0	0	None
Running Example	251	9	2	2	`NullPointerException` and `ResourceNotFoundException`
Gestures Builder	176	7	0	0	None
Tomdroid	36708	76	1	1	`ArrayIndexOutOfBoundsException`
FBReader	76148	879	1	1	`ArrayIndexOutOfBoundsException`

[a]LOC: Line Of Code; [b]NOO: Number Of 975 Objects; [c]NOA: Number Of All exceptions; [d]NOU: Number Of Unique exceptions.

(see Section 4) have been used. 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 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 concerns executing those test cases over the application. For this purpose, the Python's AVC library facilitates the execution of each test case while the application is being executed in the ADT (Android Developer Tools) simulator. Traces of these test cases will be recorded; finally, they are used to rank lines of the application's source code based on their probability of being faulty.

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 [24], 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.

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 those 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, too.

5.4. Threats to validity

Several factors may potentially affect the validity of the results of this experiment. This section provides a description of these factors.

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 [25]. 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 with the ability to extend or change the lists of events. Another threat to internal validity relates 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 may be a chance of error in the results. This threat is minimized by adapting a well-established approach for generating test cases published in [21].

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 [25].

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

Table 8. Evaluations results (all line numbers is this table refer to the case studies available online at [1]).

Application	Number of test cases		Main causes of exceptions	Detected causes of exceptions
	Passed	**Failed**		
Calculator	8	8	The object `amount1` has no value	The highest scores are given to *lines 17, 49, and 75* which are responsible for assigning a value to object `amount1`
	8	4	The object `amount2` has no value	The highest scores are given to *lines 17, 50, and 76* which are responsible for assigning a value to object `amount2`
Tippy Tipper	44	0	None	None
Gallery	6	4	(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*	The highest scores are given to *lines 23, 25, and 26*
3000 Pishvaz Code	128	4	(i) The execution of *line 79* of `CategoryActivity` activity; (ii) The execution of the `switch` block with an incorrect value	The highest scores are given to *lines 79 and 72_2*
24Game	128	0	None	None
Running example	12	4	(i) *Line 47* of `SubActivity1` (Listing 2) 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`; and (iv) A bad assignment to variable `num` in *line 42* of `SubActivity1`	The highest scores are given to *lines 42, 43, 44_1, and 47* which are responsible for assigning a value to object `view`
	12	4	The resource with id 10 does not exist	The highest score is given to *line 56* of `SubActivity1` (Listing 2) which is trying to get a non-existing resource
Gestures builder	48	0	None	None
Tomdroid	127	5	(i) A bad assignment to variable `acceptedFileExtensions` at *line 107 or 118*; (ii) The execution of *line 116* with the true condition; and (iii) A bad assignment to variable `collection` at *line 117*	The highest scores are given to *lines 107, 117, 118 and 121* and after them to *lines 64, 70, 116_0, 116_1*
FBReader	798	84	(i) A bad assignment to variable `myEditPosition` at *line 105*; (ii) Calling the function with an incorrect argument at *line 106*; and (iii) The execution of the `switch` block with input zero at *line 94* and the execution of *line 96*	The highest scores are given to *lines 106, 105, 96, 94_0, 61, and 59*

Table 9. The graph size and the execution time of applying our approach to different case studies.

Case study	Graph(V,E)	EGT[a]	GTCT[b]	ETCT[c]	RT[d]
Calculator	(3, 13)	3.7	<1	42	<2
Tippy Tipper	(5, 48)	12.6	<1	66	<2
Gallery	(4, 24)	4.4	<1	66	<2
3000 Pishvaz Code	(12, 62)	27.9	<1	198	<3
24Game	(4, 16)	5.7	<1	192	<3
Running Example	(6, 14)	4.9	<1	48	<2
Gestures Builder	(4, 14)	3.53	<1	72	<2
Tomdroid	(9, 65)	24	<1	198	<3
FBReader	(42, 212)	256	<2	1323	<4

[a]EGT: Extracting Graph Time; [b]GTCT: Generating Test Cases Time;

[c]ETCT: Executing Test Cases Time; [d]RT: Ranking Time; all times are in minutes.

Table 10. A comparison of the results of applying our proposed ranking metric with the results of using the Tarantula and Jaccard metrics.

Application	Comparison factors											
	DMC[a]			IDMC[b]			DULOC[c]			IRAL[d]		
	Ranking Metric											
	P[e]	T[f]	J[g]	P	T	J	P	T	J	P	T	J
Running Example	√	√	√	-	√	√	√	√	√	-	√	√
Calculator	√	√	√	-	√	√	√	√	√	-	√	√
Gallery	√	√	√	-	-	-	√	√	√	-	√	√
3000 Pishvaz Code	√	√	√	-	-	-	√	-	-	-	√	√
Tomdroid	√	√	√	-	-	-	√	-	-	-	√	√
RBReader	√	-	-	√	√	√	√	-	-	√	√	√

[a]DMC: Detecting main reason; [b]IDMC: Incorrect detection as main reason;

[c]DULOC: Detecting unrelated lines; [d]IRAL: Incorrect ranking of alternative lines;

[e]P: Proposed metric; [f]T: Tarantula metric; [g]J: Jaccard metric.

not considered in our evaluations. In this paper, we did not claim that we could 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., [9,10,13]) 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 [25].

For this experiment, the main threat to constructing 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; thus, we knew the precise location of faults.

5.4.4. Replicability

Replicability validity investigates whether or not we get the same results [25] by rerunning the approach on the same inputs. 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, also, available online at [24]. Moreover, the prototype implementation of our approach can be downloaded from [24]. 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 Tables 8 and 10). Furthermore, as pointed out in executionphase, 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, too. 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; 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 are being executed in Android simulators.

6.2. Proposed ranking metric

The existing ranking metrics such as the *Tarantula* and the *Jaccard* only use the execution frequency of passed and failed test cases; 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 is required to analyze the application's source code 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, too.

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., [10,26-28]) and *fully-automated* (e.g., [9,11-15]) fault localization approaches have been proposed in the literature. Furthermore, these approaches can be classified into *static*, *dynamic*, or *hybrid* ones. Static approaches (e.g., [9-12]) only work with the source code of an application without taking into account its runtime information. On the other hand, dynamic approaches (e.g., [14,16,29]) only use the information collected at the runtime of an application and do not consider its source code. Nonetheless, hybrid approaches (e.g., [30-33]) 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, consider related work in three categories: (i) the fault localization approaches 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 that 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 [5] categorized fault localization approaches to *slice-based*, *spectrum-based*, *statistics-based*, *state-based*, *machine learning-based*, *model-based*, and *data mining-based* approaches. Therefore, to be compatible with this categorization, in the following, first, an overview of these categories of approaches is provided.

The idea of slice-based approaches, such as [34,35], 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 [36,37], compare the program spectra 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, such as the execution information for conditional branches.

Statistics-based methods (e.g., [17,18,30]) attempt to rank program statements of being the cause of a fault using the statistical metrics such as *Tarantula* and *Jaccard*. Similar to our proposed approach, these metrics are calculated by running a set of test cases.

In state-based approaches, such as [38], 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. In 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 [31,40], 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 analyses 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, such as [28,41], abstract the software fault localization problem to a data mining problem, especially when we have a huge number of code lines. For example, we may seek to identify the patterns of program statements execution that lead to 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; 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, such as [42,43], is that they are highly dependent on the previously reported faults; 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. This kind of approaches is named as *combined* techniques, and their examples are included [44,45].

The fault localization process in most of the approaches mentioned above is highly dependent on the input test cases. To mitigate this, Zeller and Monperrus in [46] introduced a technique named *test case purification* whose 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 were used during the fault localization, then more accurate and 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 to smart mobile applications, their implemented tools are not necessarily able to detect faults and exceptions that are specific to smart mobile applications, such as `ActivityNotFoundException` in Android applications. However, the 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 related 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, too. For instance, Takala et al. [14] used a tool named *TEMA* to extract the events of an Android application. The extracted events are then used to generate a Finite State Machine (FSM) for the application. Next, GUI faults are localized by means of this FSM. In another work, Yang et al. [15] implemented a tool, named *ORBIT*, that tests 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, such as [9-12], analyze the data flow graph of an application to localize various kinds of faults. For example, Vekris et al. [11] and Pathak et al. [12] did a similar work to detect energy faults using some defined policies. More specifically, they sought paths in the data flow graph that acquire a resource at some point of time and do not release it later. Egele et al. [9] considered the privacy threats that iOS applications pose to users. In particular, they provided 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 the third parties. Another work presents *AndroidLeaks* [10], 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 reach 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 and Neamtiu [16] and Gottschalk et al. [13] proposed 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, Hu and Neamtiu [16] performed a bug mining study to identify the patterns of GUI bugs that are quite common. On the other hand, Gottschalk et al. [13] sought 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 related literature. For instance, Pathak et al. [29] proposed a state-based approach to localize energy faults. In their work, the state of the application was recorded periodically, and when an energy bug occurred, the fault was localized by comparing the current state of the application with the previous ones.

Unlike our proposed approach that focuses 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, the proposed approach focuses on exceptions that 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. [19] introduced a tool named CrashScope. This tool explores a given Android application with the goal of triggering crashes. For this purpose, systematic input generation is used according to several strategies informed by static and dynamic analyses. 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, [47] presented 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. Therefore, this work motivates the need for highly automated tools that can answer this challenging task.

7.3. Exception localization approaches

There have been studies in the literature that localize exception faults in programs. In this section, an overview of these approaches of our particular interest is provided.

Barr et al. [48] used symbolic execution of programs to localize `FloatingPointException` in *C/C++* and *Fortran* programs. In another work, Payet et al. [6] analyzed 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; hence, this method is not a general approach.

Hu et al. [16] categorized faults in Android applications based on studying ten popular forums. Unhandled exceptions represent one of their proposed categories, although they do not discuss how to address them.

Sinha et al. [49] 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: *NullPointer*, *Arithmetic*, and *Type* exceptions. However, this method cannot localize other kinds of exceptions. In a similar approach, Jiang et al. [50] used 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 such as `ActivityNotFoundException`.

8. Conclusions and future work

This paper presented a new approach to localize 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, a statistical ranking metric was proposed 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 attempts 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 that analyzes the application's source code to remove unrelated lines from the list of suspicious lines. The approach to nine Android applications of different sizes with different number of various exceptions was evaluated. In all of our case studies, our technique was able to detect correctly the causes of occurred exceptions. Our experimental evaluations also indicated that our ranking metric outperformed two of the widely used Tarantula and Jaccard ranking metrics.

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

References

1. https://sites.google.com/site/exceptionfaultlocalization/, Supporting Materials (2016).
2. Barr, E.T., Vo, T., Le, V., and Su, Z. "Automatic detection of floating-point exceptions", In *ACM SIGPLAN Notices*, **48**, ACM, pp. 549-560 (2013).
3. Berkhin, P. "A survey of clustering data mining techniques", In *Grouping Multidimensional Data*. Springer, pp. 25-71 (2006).
4. Briand, L.C., Labiche, Y., and Liu, X. "Using machine learning to support debugging with tarantula", In *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on IEEE*, pp. 137-146 (2007).
5. Easterbrook, S., Singer, J., Storey, M.-A., and Damian, D. "Selecting empirical methods for software engineering research", In *Guide to Advanced Empirical Software Engineering*, Springer, pp. 285-311 (2008).
6. Egele, M., Kruegel, C., Kirda, E., and Vigna, G. "PiOS: Detecting privacy leaks in iOS applications", In *NDSS*, pp. 177-183 (2011).
7. Gibler, C., Crussell, J., Erickson, J., and Chen, H. "AndroidLeaks: automatically detecting potential privacy leaks in Android applications on a large scale", *International Conference on Trust and Trustworthy Computing*, Springer, pp. 291-307 (2012).
8. Gottschalk, M., Josefiok, M., Jelschen, J., and Winter, A. "Removing energy code smells with reengineering services", **208**, pp. 441-455 (2012).
9. Habibi, E. and Mirian-Hosseinabadi, S.-H. "Event-driven web application testing based on model-based mutation testing", *Information and Software Technology*, **67**, pp. 159-179 (2015).
10. Harrold, M.J., Rothermel, G., Sayre, K., Wu, R., and Yi, L. "An empirical investigation of the relationship between spectra differences and regression faults", *Software Testing Verification and Reliability*, **10**(3), pp. 171-194 (2000).
11. Hu, C. and Neamtiu, I. "Automating GUI testing for Android applications", In *Proceedings of the 6th International Workshop on Automation of Software Test ACM*, pp. 77-83 (2011).
12. Jiang, S., Zhang, H., Wang, Q., and Zhang, Y. "A debugging approach for Java runtime exceptions based on program slicing and stack traces", In *Quality Software (QSIC), 2010 10th International Conference on, IEEE*, pp. 393-398 (2010).
13. Jones, J.A. and Harrold, M.J. "Empirical evaluation of the tarantula automatic fault-localization technique", In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ACM*, pp. 273-282 (2005).
14. Jones, J.A., Harrold, M.J., and Stasko, J. "Visualization of test information to assist fault localization", In *Proceedings of the 24th International Conference on Software Engineering, ACM*, pp. 467-477 (2002).
15. Lee, H.J., Naish, L., and Ramamohanarao, K. "The effectiveness of using non redundant test cases with program spectra for bug localization", In *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on, IEEE*, pp. 127-134 (2009).
16. Linares-Vásquez, M., Vendome, C., Luo, Q., and Poshyvanyk, D. "How developers detect and fix performance bottlenecks in Android apps", In *Proceedings of the International Conference on Software Maintenance and Evolution, IEEE*, pp. 352-361 (2015).
17. Mao, X., Lei, Y., Dai, Z., Qi, Y., and Wang, C. "Slice-based statistical fault localization", *Journal of Systems and Software*, **89**, pp. 51-62 (2014).
18. Mirzaei, H. and Heydarnoori, A. "Exception fault localization in android applications", In *Mobile Software Engineering and Systems (MOBILESoft), 2015 2nd ACM International Conference on, IEEE*, pp. 156-157 (2015).
19. Moon, S., Kim, Y., Kim, M., and Yoo, S. "Ask the mutants: Mutating faulty programs for fault localization", In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on, IEEE*, pp. 153-162 (2014).

20. Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., Vendome, C., and Poshyvanyk, D. "Automatically discovering, reporting and reproducing android application crashes", In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on, IEEE*, pp. 33-44 (2016).

21. Muccini, H., Di Francesco, A., and Esposito, P. "Software testing of mobile applications: Challenges and future research directions", In *Proceedings of the 7th International Workshop on Automation of Software Test*, IEEE Press, pp. 29-35 (2012).

22. Myers, G.J., Sandler, C., and Badgett, T. *The Art of Software Testing*, John Wiley & Sons (2011).

23. Nessa, S., Abedin, M., Wong, W.E., Khan, L., and Qi, Y. "Software fault localization using N-gram analysis", In *Wireless Algorithms, Systems, and Applications.* Springer, pp. 548-559 (2008).

24. Papadakis, M. and Le Traon, Y. "Metallaxis-FL: mutation-based fault localization", *Software Testing, Verification and Reliability*, **25**(5-7), pp. 605–628 (2015).

25. Pathak, A., Hu, Y.C., and Zhang, M. "Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices", In *Proceedings of the 10th Workshop on Hot Topics in Networks, ACM*, p. 5 (2011).

26. Pathak, A., Jindal, A., Hu, Y.C., and Midkiff, S.P. "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps", In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, ACM*, pp. 267-280 (2012).

27. Payet, É., and Spoto, F. "Static analysis of Android programs", *Information and Software Technology*, **54**(11), pp. 1192-1201 (2012).

28. Platon, O. "Smart C# debugger: Debugging C# programs using model based diagnosis", University Politehnica of Bucharest Scientific Bulletin, *Series C: Electrical Engineering*, **69**(1), pp. 45-60 (2007).

29. Saha, R.K., Lease, M., Khurshid, S., and Perry, D.E. "Improving bug localization using structured information retrieval", In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, IEEE*, pp. 345-355 (2013).

30. Shu, T., Ye, T., Ding, Z., and Xia, J. "Fault localization based on statement frequency", *Information Sciences*, **360**, pp., 43-56 (2016).

31. Sinha, S., Shah, H., Görg, C., Jiang, S., Kim, M., and Harrold, M.J. "Fault localization and repair for Java runtime exceptions", In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ACM*, pp. 153-164 (2009).

32. Takala, T., Katara, M., and Harty, J. "Experiences of system-level model-based gui testing of an android application", In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on, IEEE*, pp. 377-386 (2011).

33. Tantithamthavorn, C., Teekavanich, R., Ihara, A., and Matsumoto, K.-I. "Mining a change history to quickly identify bug locations: A case study of the Eclipse project ", In *Software Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on, IEEE*, pp. 108-113 (2013).

34. Thomas, S.W., Nagappan, M., Blostein, D., and Hassan, A.E. "The impact of classifier configuration and classifier combination on bug localization", *IEEE Transactions on Software Engineering*, **39**(10), pp. 1427-1443 (2013).

35. Tip, F. "A survey of program slicing techniques", *Journal of Programming Languages*, **3**(3), pp. 121-189 (1995).

36. Vekris, P., Jhala, R., Lerner, S., and Agarwal, Y. "Towards verifying android apps for the absence of no-sleep energy bugs", In *HotPower*, USENIX Association, p. 3 (2012).

37. Wen, W. "Software fault localization based on program slicing spectrum", In *Software Engineering (ICSE), 2012 34th International Conference on, IEEE*, pp. 1511-1514 (2012).

38. Wong, W.E. and Debroy, V., *A Survey of Software Fault Localization, Tech. Rep. UTDCS-45*, Department of Computer Science, University of Texas at Dallas (2009).

39. Wong, W.E., and Qi, Y. "BP neural network-based effective fault localization", *International Journal of Software Engineering and Knowledge Engineering*, **19**(04), pp. 573–597 (2009).

40. Wu, R., Zhang, H., Cheung, S.-C., and Kim, S. "CrashLocator: locating crashing faults based on crash stacks", In *Proceedings of the International Symposium on Software Testing and Analysis, ACM*, pp. 204-214 (2014).

41. Xu, Z., Zhang, J., and Xu, Z. "Memory leak detection based on memory state transition graph", In *Software Engineering Conference (APSEC), 2011 18th Asia Pacific, IEEE*, pp. 33-40 (2011).

42. Xuan, J. and Monperrus, M. "Learning to combine multiple ranking metrics for fault localization", In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, IEEE*, pp. 191-200 (2014).

43. Xuan, J. and Monperrus, M. "Test case purification for improving fault localization", In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM*, pp. 52-63 (2014).

44. Yang, W., Prasad, M.R., and Xie, T. "A grey-box approach for automated GUI-model generation of mobile applications", In *Fundamental Approaches to Software Engineering*, Springer, pp. 250-265 (2013).

45. Yi, Q., Yang, Z., Liu, J., Zhao, C., and Wang, C. "Explaining software failures by cascade fault localization", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, **20**(3), p. 41 (2015).

46. Zeller, A. "Isolating cause-effect chains from computer programs", In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM*, pp. 1-10 (2002).

47. Zhang, X., Gupta, N., and Gupta, R. "Locating faults through automated predicate switching", In *Proceedings of the 28th International Conference on Software Engineering, ACM*, pp. 272–281 (2006).

48. Zhang, Y. and Santelices, R. "Prioritized static slicing and its application to fault localization", *Journal of Systems and Software*, **114**, pp. 38-53 (2016).

49. Zhong, H. and Su, Z. "An empirical study on real bug fixes", In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, IEEE Press, pp. 913-923 (2015).

50. Zhu, L.-Z., Yin, B.-B., and Cai, K.-Y. "Software fault localization based on centrality measures", In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual, IEEE*, pp. 37-42 (2011).

Biographies

Hamed Mirzaei holds a MSc 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.

Abbas Heydarnoori is an Assistant Professor in the Department of Computer Engineering at the Sharif University of Technology. He was a post-doctoral fellow at the University of Lugano, Switzerland. Dr. Heydarnoori 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.