Applying Spectrum-Based Fault Localization to Android Applications

Euler Horta Marinho
Federal University of Minas Gerais
Brazil
eulerhm@dcc.ufmg.br

Fischer Ferreira
Federal University of Ceara
Brazil
fischer.ferreira@sobral.ufc.br

João P. Diniz
Federal University of Minas Gerais
Brazil
jpaulo@dcc.ufmg.br

Eduardo Figueiredo
Federal University of Minas Gerais
Brazil
figueiredo@dcc.ufmg.br

ABSTRACT

The pressing demand for high-quality mobile applications has a major influence on Software Engineering practices, such as testing and debugging. The variety of mobile platforms is permeated with different resources related to communication capabilities, sensors, and user-controlled options. As a result, applications may exhibit unexpected behaviors and resource interactions can introduce failures that manifest themselves in specific resource combinations. These failures can affect the quality of mobile applications and degrade the user experience. To reduce human effort of manual debugging, several techniques have been proposed and developed aiming to partially or fully automate fault localization. Fault localization techniques, such as Spectrum-based Fault Localization (SBFL), identify suspicious faulty program elements related to a software failure. However, we still lack empirical knowledge about the applicability of fault localization techniques in the context of mobile applications, specifically considering resource interaction failures. To address this problem, this paper evaluates the use of SBFL aiming to locate faults in 8 Android applications and verify the sensitivity of SBFL to variations in resource interactions. We rely on mutation testing to simulate faults and on the Ochiai coefficient as an indicator of the suspicious faulty code. Our results indicate that SBFL is able to rank more than 75% of the faulty code in 6 out of 8 applications. We also observed that the ranking of suspicious code varies depending on the combination of enabled resources (e.g., Wi-Fi and Location) in the mobile applications.

CCS CONCEPTS

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

KEYWORDS

mobile applications, resource interactions, fault Localization, SBFL

SBES 2023, September 25–29, 2023, Campo Grande, Brazil

ACM ISBN 979-8-4007-0787-2/23/09…$15.00
https://doi.org/10.1145/3613372.3613397

ACM Reference Format:
Euler Horta Marinho, Fischer Ferreira, João P. Diniz, and Eduardo Figueiredo. 2023. Applying Spectrum-Based Fault Localization to Android Applications. In *XXXVII Brazilian Symposium on Software Engineering (SBES 2023), September 25–29, 2023, Campo Grande, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3613372.3613397

1 INTRODUCTION

The growth of the mobile application market, for instance, due to the popularity of application store, has brought new challenges to their development and testing. For instance, the pressing demand for high quality applications has an important influence on Software Engineering practices, such as testing and debugging [10]. Testing is one of the most important approaches to quality assurance in the field of mobile applications, as evidenced by several secondary studies [22, 27, 45, 47]. For a proper software testing, we also need debugging which is another quality assurance activity aimed at the localization and removal of faults [55]. Nevertheless, manual debugging can be extremely challenging, tedious, and costly, since it relies heavily on the software developer experience, judgment, and intuition to identify and prioritize code that is likely to be faulty [6]. Therefore, developing techniques have been proposed aiming to partially or fully automate fault localization while reducing human effort [52].

Fault localization techniques aim to identify faulty program elements related to software failures using static or run-time information to determine the root cause of the failure [57]. Some of these techniques, such as Spectrum-based Fault Localization (SBFL), can produce a ranked list of suspicious code elements for developers, reducing their effort for manual fault checking [6]. Intuitively, the more a code element is executed by failing test cases, the more suspicious it is [19]. An SBFL technique often calculates suspiciousness scores using a ranking metric also known as a risk evaluation formula [57]. Ochiai [2], DStar [51], and Tarantula [21] are among the most common metrics for this purpose [36].

Several techniques have been proposed and evaluated for fault localization [40, 52, 54]. Nonetheless, these techniques must be assessed for mobile applications, since they may demand tailored quality assurance approaches due to mobile specific characteristics [20, 42, 43, 49]. For instance, debugging mobile applications is challenging and the localization of the faulty code may not even be apparent from the stack trace [23].

Mobile applications typically run on a variety of platform configurations [15]. Each platform configuration relies on a different set of enabled platform resources making application testing and debugging more challenging. Application resources can be related to communication features (e.g., Wi-Fi and GPS), sensors (e.g., Accelerometer and Gyroscope), and user-controlled options (e.g., Battery Saving and Do Not Disturb). Some of these resources can be managed directly through system-level settings, such as the Android Quick Settings [1], which allow the user to customize many system or application behaviors [26]. However, applications may exhibit unexpected behavior due to failures that manifest themselves in certain combinations of enabled resources [29, 43]. According to Sun et al. [43], failures involving two resources are critical but not very common in mobile applications. Another study [29] found a greater number of this type of failure.

In this work, we evaluate the use of the SBFL technique [1] aiming to locate faults in Android applications and verify the sensitivity to resource interaction failures. We use faults seeded from mutation operators in order to conduct the experimental study and rely on the Ochiai coefficient as an indicator of the suspicious faulty code [2]. Although there are many metrics for calculating the suspiciousness score, the Ochiai coefficient is considered one of the metrics with the best performance [36, 57]. Despite the Ochiai coefficient was preliminary used in mobile applications[28], we still lack knowledge about its applicability in the context of open source mobile applications, especially with respect to resource interaction failures, since the SBFL was not designed for these types of failures. Testers and developers may neglect to properly test and debug mobile applications considering resource interactions because they lack knowledge about such failures [29]. In consequence, these failures may occur during everyday use of the mobile application, while they are not noticeable during the testing and debugging activities.

To achieve our goal in this study, we follow four steps. First, we select 8 open source applications from GitHub used in our previous study [29]. Second, we use a tool [12] to generate mutants for each target application. We generate mutants for two groups of classes, i.e., classes that use APIs [2] of resources (resource-related classes) and classes that do not use such APIs (general classes). Third, we execute the test suites for each mutant and collect code coverage metrics. We then investigate the sensitivity of SBFL when there are variations in resource settings, a known source of application failures as demonstrated in previous studies [26, 29, 43]. Finally, we analyse the test reports to calculate the Ochiai coefficient for each application aiming to locate the faulty code. Our analysis is performed at the method-level. Therefore, SBFL reports the suspiciousness score for the methods.

Our results indicate that SBFL is able to rank more than 75% of the faulty methods for 6 applications. However, there is no evidence of a difference in the ranking coefficient between faults in resource-related classes and faults in general classes. Regarding the sensitivity of SBFL to variations in resource settings, we found a major influence of resource settings on the suspiciousness score. That is, for the same failure (i.e., mutant), the ranking of suspicious methods varies depending on the combination of enabled resources.

The remainder of this paper is organized as follows. Section 2 presents background information on spectrum-based fault localization, mutation testing, and resource interactions in mobile applications. Section 3 describes the study design. Section 4 discusses the results of our empirical study. Section 5 presents discussions about the results. Section 6 describes the threats to validity of this work. Section 7 presents some related work. Finally, Section 8 concludes this study and shows directions for future work.

2 BACKGROUND

In this section, we present an overview of concepts of Spectrum-based fault localization (Section 2.1), mutation testing (Section 2.2), and resource interactions in mobile applications (Section 2.3).

2.1 Spectrum-based fault localization

SBFL is a technique based on the analysis of the program spectra or coverage [1], i.e., the program elements covered during a test execution [18]. These elements can be of different granularity level, e.g., statements, blocks, predicates, methods, etc. We focus on method-level fault localization in this paper. Many SBFL techniques use ranking metrics to associate a suspiciousness score to the program elements. These techniques produce as output a list of elements ranked in descending order of suspiciousness [11].

The metrics used to calculate the suspiciousness score are the major concern for the design of a SBFL technique. Several metrics have been proposed to indicate faulty elements [19]. Tarantula[21] was the first metric proposed exclusively for fault localization. The Ochiai [2] coefficient was adapted from the Molecular Biology. Often, the metrics are defined in terms of four values collected of the execution of the tests [57]:

- e_f: number of failed tests that execute the program element.
- e_p: number of passed tests that execute the program element.
- n_f: number of failed tests that do not execute the program element.
- n_p: number of passed tests that do not execute the program element.

For example, Ochiai uses the following formula for calculating the suspiciousness of a program element:

$$Ochiai(element) = \frac{e_f}{\sqrt{(e_f + n_f) * (e_f + e_p)}} \quad (1)$$

Figure 1 presents a code snippet of the signatures of some methods implemented by the `GPSLogger` class of OSMTracker [3], to illustrate the use of the Ochiai coefficient for SBFL. This application is a trip tracker that works with data from GPS and updates the location at regular intervals, defined by `gpsLoggingInterval`. `onLocationChanged` is a faulty method, whose signature is defined in the `LocationListener` interface [4] of the Android Location API. At the top, ten test cases (t1 to t10) are presented. For each row, a • is used to indicate that the method is covered by the test case. At the bottom, we present the information showing whether the

[1] support.google.com/android/answer/9083864?hl=en
[2] LocationManager (https://developer.android.com/reference/android/location/LocationManager) is an example of API for location resources
[3] https://github.com/labexp/osmtracker-android
[4] https://developer.android.com/reference/android/location/LocationListener

test case passed or failed. The last column indicates the values of the Ochiai coefficient. Test cases 1, 3, 4, and 7 have failures. The Ochiai coefficient calculated for the `onLocationChanged` method is 1.00. Intuitively, the coefficient for this method is high because it is more covered by failed tests than by passed tests. We can observe that SBFL encourages the developer to inspect the most suspicious method first. A faulty method cannot be ranked if the values e_f and n_f are both zero.

2.2 Mutation testing

Mutation analysis is the process of introducing syntactic variations in a program aiming to produce program variants (mutants), i.e., generating artificial faults [34]. Mutation testing refers to the use of mutation analysis in order to assess the quality of a test suite. When a test case shows the behavior of a mutant to be different from that of the original program, the mutant is said to have been "killed" or "detected" [34]. Otherwise, the mutant is said to be "live". During this analysis, we measure the number of mutants that are killed and calculates the ratio of those over the total number of mutants. This ratio is called mutation score [24].

The syntactic variations of mutation analysis is performed by means of "mutation operators" [34]. A basic set of mutant operators, usually considered as a minimum standard for mutation testing [24] is the five-operator set proposed for the Mothra mutation system [30]. This set includes the Relational Operator Replacement (ROR), Logical Connector Replacement (LCR), Arithmetic Operator Replacement (AOR), Absolute Value Insertion (ABS), and Unary Operator Insertion (UOI) operators. For example, let's consider the snippet of the `onLocationChanged` method in Figure 2. This method receives a Location object as an argument and updates the location in line 12. A time stamp defined by `lastGPSTimestamp` is maintained to control the location update (line 11). We use the ROR operator for mutating the relational operator of line 3 and generate 5 mutants (lines 5-9) [5]: $expr1 <= expr2$, $expr1 > expr2$, $expr1 >= expr2$, $expr1 == expr2$, and $expr1 != expr2$.

Recent studies [13, 38, 41] have presented specific approaches for the mutation testing of mobile applications. For example, specialized mutation operators have been proposed from the taxonomy of real faults of Android applications [13]. Moreover, cost reduction techniques for the mutation testing of Android applications were catalogued as a set of good practices [38].

2.3 Resource interactions in mobile applications

Resource interaction failures have only been recently explored in mobile applications testing [26, 29, 43]. These failures occur when resources affect the behavior of other resources, similarly to the feature interaction problem in configurable software systems [5] and telecommunication systems [7]. An example of resource interaction failure occurs for `Wikimedia Commons app` [43]. Figure 3 presents a code snippet involved in a failure described in an issue [6]. The Android platform uses the GPS or the network (Wi-Fi/Mobile data) to determine the device location. This application fails if both GPS and the network are disabled. The failure is caused by calling `getLastKnownLocation` to get the current location over the network (line 3). However, this call returns a `null` value, which is later used to store the location-based values when constructing an object (line 5). As a result, the application crashes due to a `NullPointerException`. The issue was closed with a proper correction [7].

The high number of input combinations is a challenging aspect for testing software systems in general, since the effort of the exhaustive testing is generally prohibitive. Particularly, it is also the case of configurable systems [4, 9, 14] in which all tests must be executed with several configurations. Our previous work [29] named a input combination as a *setting*, i.e. a set of resources whose states (enabled or disabled) are previously defined. For instance, our previous study [29] considered a set of 14 common resources: `Auto Rotate`, `Battery Saver`, `Bluetooth`, `Camera`, `Do Not Disturb`, `Location`, `Mobile Data`, `Wi-Fi`, `Accelerometer`, `Gyroscope`, `Light`, `Magnetometer`, `Orientation`, and `Proximity`.

In this study, we evaluate the sensitivity of SBFL to variations in resource settings considering the same set of resources. That is, we investigate if SBFL is able to detect variations of combinations of resources, allowing us to locate the faulty code behind the resource failures.

3 STUDY DESIGN

This section presents the experimental design of our study. Section 3.1 shows the research questions and Section 3.2 delineates each phase of the study.

3.1 Research Questions

The goal of this study is to evaluate the use of SBFL to locate faults in Android applications and verify the sensitivity to resource interactions. To achieve this goal, we address the following research questions:

RQ1: To what extent SBFL can be used for mobile applications?

RQ2: Is there a difference in the ranking coefficient for faults in resource-related classes and faults in general classes?

RQ3: How sensitive is SBFL to variations in resource settings?

The first research question can be answered by applying the SBFL and measuring the suspiciousness score using the Ochiai coefficient. We use faults seeded by mutation operators to control the fault localization. To answer the second research question, the faults are seeded in two groups of classes, resource-related classes and general classes. The resource-related classes are identified based on the analysis of the imported packages [31]. For the third research question, we compare the suspiciousness scores in the context of the variation of resource settings.

3.2 Study Steps

Figure 4 depicts an overview of the steps of the study. First, we select the first application set to answer RQ1 and RQ2 (Step 1). These applications must be implemented in Java due to the constraints of the mutants generator tool used. From the first application set, we select the second application set to answer RQ3 and instrumented the test suites to control the resources (Step 2). These applications must have failures occurring in three executions for the selection of

[5] $expr1$ replaces $lastGPSTimestamp + gpsLoggingInterval$ and $expr2$ replaces System.currentTimeMillis()

[6] https://github.com/commons-app/apps-android-commons/issues/1735

[7] https://github.com/commons-app/apps-android-commons/pull/1791

Application: OSMTracker	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	Ochiai
class GPSLogger {...											
(1) public void onCreate() {...}	●	●	●	●	●	●	●	●	●	●	0.63
(2) public int onStartCommand(Intent intent, int flags, int startId) {...}	●	●	●	●	●	●	●		●	●	0.67
(3) public void onDestroy() {...}	●	●	●	●	●	●	●	●	●	●	0.63
(4) private void startTracking(long trackId) {...}	●	●	●	●	●	●			●	●	0.53
(5) private void stopTrackingAndSave() {...}	●	●	●	●	●	●			●	●	0.53
(6) public void onLocationChanged(Location location) {...} /* FAULT */	●		●	●			●				1.00
(7) private Notification getNotification() {...}	●	●	●	●	●	●	●		●	●	0.67
(8) private void createNotificationChannel() {...}					●	●					0.00
...}											
Test case outcomes (pass=✓, fail=✗)	✗	✓	✗	✗	✓	✓	✗	✓	✓	✓	

Figure 1: Example of Ochiai coefficient.

```
1  void onLocationChanged(Location location) {
2  ...
3   if((lastGPSTimestamp + gpsLoggingInterval) < System.current_TimeMillis()) {
4
5   //mut1:  if((lastGPSTimestamp + gpsLoggingInterval) <= System.current_TimeMillis())
6   //mut2:  if((lastGPSTimestamp + gpsLoggingInterval) > System.current_TimeMillis())
7   //mut3:  if((lastGPSTimestamp + gpsLoggingInterval) >= System.current_TimeMillis())
8   //mut4:  if((lastGPSTimestamp + gpsLoggingInterval) == System.current_TimeMillis())
9   //mut5:  if((lastGPSTimestamp + gpsLoggingInterval) != System.current_TimeMillis())
10
11    lastGPSTimestamp = System.current_TimeMillis();
12    lastLocation = location;
13    if (isTracking) {...}
14   }
15 }
```

Figure 2: Code Snippet of the `onLocationChanged` method of the `GPSLogger` class.

```
1  Location lastKL = locationManager.
       getLastKnownLocation(LocationManager.
       GPS_PROVIDER);
2  if (lastKL == null) {
3       lastKL = locationManager.
            getLastKnownLocation(LocationManager.
            NETWORK_PROVIDER);
4  }
5  return LatLng.from(lastKL); //An object is
       constructed from the latitude and
       longitude coordinates
```

Figure 3: Code Snippet from the Wikimedia Commons Android app.

the settings, since other studies [29] showed that this number of test executions was sufficient to detect flaky tests. The first application set was used to generate mutants (Step 3). For each mutant of the first application set, we execute the test suites (Step 4). This phase also includes the execution of the test suites of the second application set. Finally, we analyse the recorded test reports to calculate the Ochiai coefficient (Step 5). In the following sections, we detail each step.

3.3 Application Selection

Based on our previous work [29], we randomly selected 8 applications that meet the following criterion: implemented in Java and with test code size greater than 500 LOC. Table 1 depicts an overview of the selected applications. These applications are from different categories with a large variation of size (from 14,499 LOC to 347,897 LOC), test code size (from 525 LOC to 3,674 LOC), and test cases (from 4 to 164). We can observe a relative low instruction coverage of the test suites (from 2% to 51%). The column "Resources" presents resources declared in the Manifest file [8] that are used for the application selection. Some resources are not declared since they are not directly used by the application (for instance, `Auto Rotate` and `Battery Saver`). Other resources do not demand a `uses-permission` tag and may not be explicitly required by the developer with a `uses-feature` tag (for instance, `Accelerometer`

[8] https://developer.android.com/guide/topics/manifest/manifest-intro

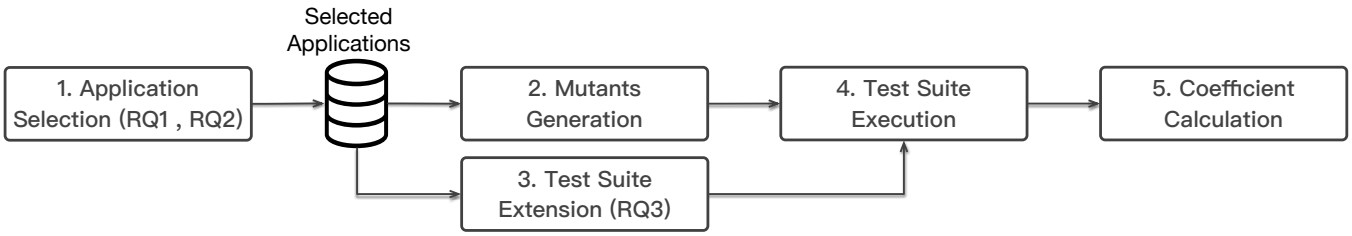


Figure 4: The steps of the study.

and `Gyroscope`). In this case, other approaches for code analysis could be used for identifying additional resources.

3.4 Mutants Generation

We were not able to find specific mutation tools or operators for resource interaction failures. Therefore, we decided to use a generator prototype tool [12] that implements four of the five operators of the Mothra mutation system [30]. This tool is able to generate mutants for Java code using the set of three mutants operators shown in Table 2. We opt for this tool since it requires less effort for setup execution and log generation and makes it simple to control the generation and execution of mutants.

These mutation operators (AOR, ROR, and LCR) are a subset of the five representative ones [30]. They act on binary expressions and replace the language operator (arithmetic, relational, or logical) with other syntactically similar operators. Although the mutation operator SBR (Statement Block Removal) was implemented in the used tool [12], we do not use this operator because SBFL techniques have limitations in locating faults related to missing code [40]. These mutation operators are capable of of reproducing faults related to resource interactions in mobile applications because we generate mutants for resource related classes. This strategy allows the modification of the code related to the resource, as seen in the example of Figure 2 in a similar way to the mutation generation strategies in other studies [14].

We generate all mutants to each target application based on the selected operators, since the tool uses the concept of metamutant [46] to encode all mutants and the original source code into one application. In this way, the compilation and loading time is reduced, because all mutants can be enabled/disabled at runtime. Thereafter, we conducted a previous analysis to identify mutants covered by at least one test case.

The number of mutants generated is constrained by the SBFL approach, where each test case is executed individually, increasing the testing effort. We attempt to select 20 mutants (10 mutants for resource-related classes and 10 mutants for general classes) in each target application. The number of mutants for each application can be found in Table 3. However, we could not generate an uniform number of mutants for some applications due to a lack of mutation points. The mutants were randomly selected from the mutation operators set (AOR, ROR, and LCR). We restricted our study to 20 mutants due to experimental time constraints.

As we can notice, we are not able to seed all mutants for resource-related classes for Ground and Threema. For Ground, we only identify 5 mutants for this kind of class. A possible reason for Threema is the low coverage of the test suite (2% in Table 1), although this application has a large number of resource-related classes.

3.5 Test Suite Extension

Similar to our previous work [29], we instrumented code aiming to control 14 common resources of the Android Platform: `Auto Rotate`, `Battery Saver`, `Bluetooth`, `Camera`, `Do Not Disturb`, `Location`, `Mobile Data`, `Wi-Fi`, `Accelerometer`, `Gyroscope`, `Light`, `Magnetometer`, `Orientation`, and `Proximity`. The instrumentation is based on Android instrumented tests, i.e., a type of functional test [9]. They execute on devices or emulators and can interact with Android framework APIs.

Each class of the test suites is extended with the instrumentation, allowing the control of specific contextual information of the resource states. The control of resources is based on settings. A setting is defined as a 14-tuple of pairs `(resource, state)` where `state` can be `True` or `False` depending on whether the `resource` is enabled or disabled.

3.6 Test Suite Execution

For each mutant of the first set of applications and the applications of the second set, we execute the test suites with the coverage reports enabled. We used a Xiaomi Pocophone F1 with 6 GB RAM, running Android 10. Since the calculation of the coefficient is based on the output of each test case, we execute test cases separately.

For illustrating the experimental effort, we collect a sample of the approximate execution time for all 20 mutants of each application in "Execution Time" column of Table 1. The CPU time was randomly sampled, since from our observation, we could not verify a great variation of time in the execution of the mutants. We can see that the execution time varies between 1h45m (OpenScale) to 1d3h (WordPress).

3.7 Coefficient Calculation

We analyzed test reports to identify failed test cases and the coverage reports to get the coverage information. In order to decrease the complexity of the analysis, we use the method coverage data for calculating the Ochiai coefficient of each method in target applications.

4 RESULTS

This section presents the study results and discusses them focusing on providing answers to the research questions. Section 4.1 provides

[9] https://developer.android.com/training/testing/instrumented-tests

Table 1: Characteristics of the Selected Applications.

Application	Description	Category	LOC	Test LOC	Test cases	Coverage (%)	Execution Time	Resources
AnkiDroid [3]	A flashcard-based study aid	Education	158,607	2,770	164	17	~15h00m	Camera, Mobile Data, Wi-Fi
Ground [17]	A map-first data collection platform	Productivity	19,906	525	4	17	~3h40m	Camera, Mobile Data, Location, Wi-Fi
OpenScale [32]	A weight and body metrics tracker	Health, Fitness	27,781	1,451	14	33	~1h45m	Bluetooth, Location
OwnTracks [33]	A location tracker	Travel, Local	14,499	889	27	51	~4h15m	Location, Mobile Data, Wi-Fi
PocketHub [37]	An application for managing GitHub repositories	Productivity	29,001	1,663	107	13	~8h15m	Mobile Data, Wi-Fi
Radio-Droid [39]	A radio streaming application	Music, Audio	22,815	1,735	23	28	~2h50m	Bluetooth, Mobile Data, Wi-Fi
Threema [44]	An instant message application	Communication	238,045	1,931	54	2	~8h10m	Bluetooth, Camera, Location, Mobile Data, Wi-Fi
WordPress [53]	A content management application	Productivity	347,897	3,674	115	19	~1d3h	Camera, Mobile Data, Wi-Fi

Table 2: Mutation Operators.

Mutation operator		Original	Mutations	
AOR	Arithmetic Operator Replacement	a - b	a + b a - b	a * b a % b
ROR	Relational Operator Replacement	a <= b	a >b a == b a <b	a >= b a != b
LCR	Logical Connector Replacement	a || b	a && b	

Table 3: Mutants Generated for each Application.

Application	Resource-Related Classes	General Classes
AnkiDroid	10	10
Ground	5	15
OpenScale	10	10
OwnTracks	10	10
PocketHub	10	10
Radio-Droid	10	10
Threema	0	20
WordPress	10	10

the answer to RQ1, Section 4.2 presents the answer to RQ2, and Section 4.3 provides the answer to RQ3.

4.1 Use of SBFL for Mobile Applications (RQ1)

Table 4 presents an overview of the results of the executions with the total rank in descending order. The column "DM" (Dead Mutants) refers to the amount of mutants detected by the test suite. The column "MS" (Mutation Score) exhibits the mutation score considering the total of mutants generated (Table 3). We calculate the Ochiai coefficients and the ranking position according to the procedures

Table 4: Overview of the Results.

Application	DM	MS	Ranking of Mutants		
			Rank <= 10	Rank > 10	Total
Threema	18	0.90	18(100%)	0(0%)	18(100%)
PocketHub	9	0.45	9(100%)	0(0%)	9(100%)
OpenScale	7	0.35	7(100%)	0(0%)	7(100%)
Ground	1	0.05	1(100%)	0(0%)	1(100%)
Radio-Droid	4	0.20	2(50%)	1(25%)	3(75%)
AnkiDroid	20	1.00	6(30%)	4(20%)	10(50%)
WordPress	12	0.60	4(34%)	1(8%)	5(42%)
OwnTracks	8	0.40	3(37%)	0(0%)	3(37%)

presented in Section 2.1. The columns "Rank <= 10" and "Rank > 10" show the amount of faulty methods ranked by Ochiai within top-10 positions and positions greater than 10, respectively. The column "Total" indicates the total of dead mutants that were ranked by the suspiciousness score (Ochiai). That is, this last column is the sum of the "Rank <= 10" and "Rank > 10" columns.

The amount of dead mutants and the mutation score are related to the quality of the test suites, generally assessed by some code coverage criteria [16]. We can note that AnkiDroid, Threema, and WordPress have mutation scores greater than 0.60 while the number of test cases oscillated between 54 and 164 (Table 1). However, their code coverage is between 2% and 19%. According to the ranked dead mutants, the SBFL is able to rank more than 75% of the dead mutants for 6 applications. A faulty method could not be ranked if there is no failed test case for it.

4.2 Coefficients in Resource-Related Classes and in General Classes (RQ2)

We analyse the Ochiai coefficients for resource-related classes (Group 1) and general classes (Group 2). Figure 5 shows the box plots of the coefficients. The horizontal axis presents the groups,

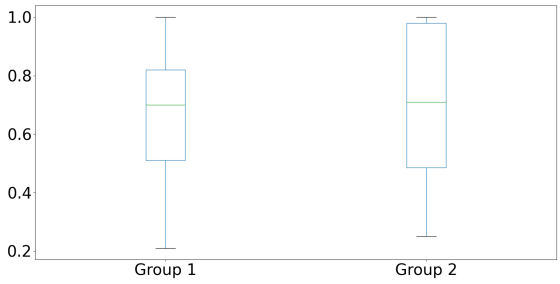


Figure 5: Ochiai coefficients for the groups of classes.

while the vertical axis presents the Ochiai coefficients. We can see that the variance of Group 2 is larger than the variance of Group 1.

To answer RQ2, we check whether the coefficients of Group 1 and Group 2 follow a normal distribution. First, we create a Quantile-Quantile (QQ) plot of the data of each group. Figure 6 depicts the QQ plots for Group 1 and Group 2.

As we can see, the points in both groups do not fall approximately on the diagonal straight line, indicating that our data do not follow a normal distribution. Therefore, we perform a nonparametric test using the Mann-Whitney U test (also known as the Mann-Whitney Wilcoxon test or the Wilcoxon Rank Sum test). For this test, we defined the following null and alternative hypotheses.

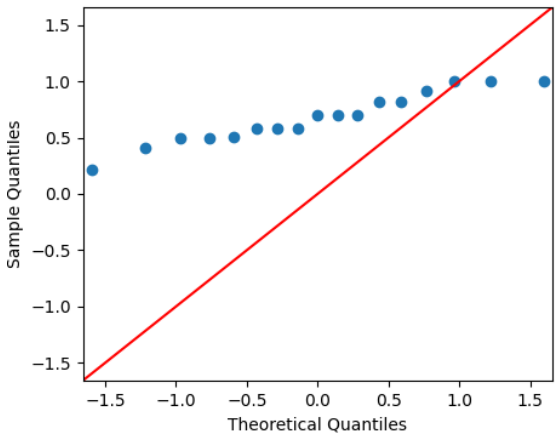
H0: Groups 1 and 2 are from the same population.
H1: Groups 1 and 2 are not from the same population.

We perform the test with a 5% confidence level (i.e., $\alpha = 0.05$) and obtain a p-value = 0.99, which does not allow the rejection of the null hypothesis. In addition, we calculate the 95% confidence intervals for the means of the two groups. Figure 7 shows the overlap of the confidence intervals. Accordingly, we have no evidence of a difference between the groups.

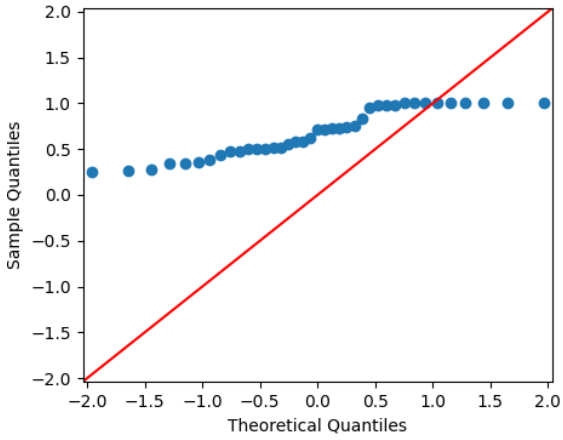
4.3 Variations in Resource Settings (RQ3)

We select three instrumented applications to compare the suspiciousness score in the context of the variation of resource settings. The applications were selected considering the fact that they had failures manifested in three executions, an experimental procedure for dealing with flaky tests [35]. We randomly select settings for each application from 2^{14} possibilities that are able to cause failures in three executions to avoid flaky tests. For instance, S_A for application Owntracks is ⟨Wi-Fi, !MobileData, !Location, Bluetooth, !Camera, !AutoRotate, !BatterySaver, !DoNotDisturb, !Accelerometer, !Gyroscope, Light, Magnetometer, Orientation, Proximity⟩, in which the exclamation mark indicates the disabled resource. We analyze the pairs of ranks and determine the number of methods ranked differently and calculate the percentage of difference in relation to the total number of ranked methods.

Table 5 depicts the applications, the settings id, and the difference of the rank between pairs of settings. The percentage of difference of the rank varied between 0% and 98% suggesting an influence of the resource settings on the suspiciousness score. This can point out to extensions of the SBFL techniques to rank resource related classes.



(a) QQPlot of Group 1



(b) QQPlot of Group 2

Figure 6: QQ Plots - Groups of classes

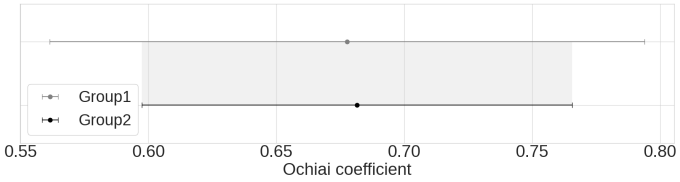


Figure 7: Overlap in the 95% confidence intervals.

In this table, the settings id were labelled to make the presentation more clear.

5 DISCUSSION

We considered 8 applications with test suites created by the developers. In this way, the ranking of faults depends on the quality of these test suites. Since we investigate fault localization in the

Table 5: Difference of the rank

Application	Settings id	Difference of the rank
OwnTracks	S_A, S_B, S_C	S_A-S_C (70%), S_B-S_C (70%), S_A-S_B (0%)
PocketHub	S_A, S_B, S_C	S_A-S_B(0%), S_A-S_C(0%), S_B-S_C(0%)
Threema	S_A, S_B, S_C	S_A-S_B (98%), S_A-S_C (28%), S_B-S_C (28%)

context of resource-related failures by looking at RQ2 and RQ3 (Section 3.1), our analysis relies on instrumented tests, a type of functional tests (Section 3.5). Therefore, further efforts need to be made to extend the detection of these failures to other levels, e.g., unit and integration tests.

As we answered RQ2, there is no evidence of a difference between the two groups of classes. Therefore, we believe that the ranking of SBFL needs to be improved by differentiating the contribution of the tests to produce a broader program spectra [56]. We can use coverage metrics of failed tests with respect to the methods of resource-related classes. For example, following the same logic as in the study of Zhang et al. [56], a failed test that covers fewer methods would be more helpful for locating faults than a failed test that covers more methods.

As we answered RQ3, we found an influence of resource settings on the suspiciousness score. Using the example of Figure 3 presented in Section 2.3, we argue that resource-interactions can be caused by multiple faulty elements as can be seen by the provided correction of the issue [10]. Therefore, similar to the study of Zou et al. [57], we assume that if a failure is caused by multiple faulty elements, a fault will be localized by SBLF if one faulty element is localized. That is, if SBFL indicates one of the faulty elements, the developer can infer the other faulty elements.

We observed that the SBFL is thus a fault localization technique that can be applied in the context of mobile application testing (RQ1). We see two directions to improve the SBFL. The first is to adapt the ranking metrics to leverage the characteristics of mobile applications, for instance, exploring coverage metrics that estimate the use of the resources (e.g., to point out the methods that effectively use the resources). The second is to get feedback from the ranking process to evolve the test suites. For example, creating more tests to improve the coverage of the methods of resource-related classes.

The interplay of software testing and debugging is a well-known demand for software quality assurance [8]. Wang et al. [48] emphasized the rule of test artifacts as a support mechanism for debugging. In addition, the adoption of the automation of mobile applications testing by developers is influenced by the generation of test cases that improve debugging and traceability between test cases and features [25].

As we discussed in Section 2.3, resource-interaction failures have only recently been explored in mobile applications testing. The characterization of the faults behind these failures is an identified demand [29]. In this way, an improved SBFL approach focusing on these faults could be integrated into a toolset to promote quality-related development activities.

[10]https://github.com/commons-app/apps-android-commons/pull/1791

6 THREATS TO VALIDITY

We carefully designed and conducted our study. Nevertheless, some threats to validity may have harmed our study results and discussions. We discuss below some major threats and their respective treatments. We divided into construct, internal, external, and conclusion, well-known categories of validity threats [50]

Construct validity. In our study, the first threat to construct validity is related to the choice of the subject applications and metrics. We opt to select 8 applications to favor external validity as we discuss bellow. The use of Ochiai coefficient for measuring the suspiciousness score may not properly capture how SBFL perform in mobile applications. However, we believe that the Ochiai have promise results since it is among the metrics with the best performance [36, 57].

Another threat concerns the use of artificial faults from a mutation tool. Although some studies have shown limitations in using artificial defects in experiments comparing SBFL techniques [36], the use of datasets with real defects also has limitations [56]. Our study has a different scope because we do not intend to compare the performance of the SBFL techniques, but rather to look for evidence of the feasibility of using SBFL in the context of mobile applications.

Internal validity. We use only three mutation operators. Despite the existence of recent studies discussing specialized mutation operators for mobile applications [13, 41], we believe that the mutation operators used in this study are a representative subset of the traditional five-operator set [12]. Moreover, traditional operators are associated with a significant proportion of real-world application failures as evidenced by the study of Escobar-Velasquez and others [13]

External validity. We conducted our study using 8 open-source Android applications. We believe that these applications are representative of the target population for the experimental study, as they were randomly sampled from GitHub public repositories. To mitigate the impact of the representativeness of the Android applications selected for our study, we choose applications from different categories, sizes, and test suite size (ranging from 525 LOC to 3,674 LOC). Another threat to our study is the quality of the test suite for the selected applications. Our entire analysis depends on the test suite's ability to detect failures. For instance, the study of Heiden et al. [19] suggests that the accuracy of SBFL is affected by the number of failed test cases. To mitigate the effect of an incomplete test suite, we limited our analysis to applications with a test suite of at least 500 LOC. Furthermore, we limited our analysis to mobile applications developed in Java or mixing Java and Kotlin. Therefore, we cannot generalize to other programming languages and frameworks such as Flutter and React Native.

Conclusion validity. These results reflect our perceptions and interpretations of the metrics collected from the applications after execution of the testing strategies. All authors participated in data analysis and discussions of key findings to reduce bias from any one person's interpretation. Nevertheless, we believe that one would get similar results using other metrics and tools that quantify similar attributes for the same mobile applications.

7 RELATED WORK

Several studies investigate failures in mobile applications on different aspects, such as exception tracking [42], automatic debugging [49], and setting-related defects [29, 43]. However, we investigated mobile application failures through the SBFL to better understand resource-related failures in mobile applications.

Su et al. [42] extensively studied to track failures through unique exceptions from 2,486 open-source and 3,230 commercial Android applications. Also, they conducted an online survey of 135 professional application developers to understand how developers handle exceptions. While they manually investigated failures reported by unique exceptions, we used SBFL to identify suspicious faulty program elements related to a software failure. Through their survey, they observed that developers use tools for fault detection. However, they still have many limitations, such as insufficient bug detection. Our results indicate that SBFL can rank more than 75% of the faulty code in 6 out of 8 applications. In addition, they demonstrate that manually tracking failures is a very costly activity. However, with our exploratory study, it was possible to identify that using SBFL for mobile applications is a viable and effective strategy.

Win et al. [49] describe that debugging and testing Android applications is more challenging than traditional Java programs. In order to reduce costs with testing and debugging Android applications, their work proposes an automatic debugging technique for Android applications called "Event-aware Precise Dynamic Slicing" (EPDS). This technique is based on "dynamic slicing," which is a technique that reduces the scope of program execution to a relevant subset of instructions. Their experiment evaluated the performance of EPDS compared to other techniques for automatically debugging Android applications. Their results showed that EPDS could significantly reduce program scope compared to the other techniques, making it easier for software developers to identify and correct application errors. While they are concerned with reducing the program's scope, we focus on observing the interactions of resources and using SBFL. We seek to indicate the part of the code that is more prone to failure. Furthermore, this indicates that parts are more prone to failures and thus indicates parts of the code where the tests should be concentrated. Our results indicate that the SBFL can perceive resource interaction failures and artificially inserted failures.

Sun et al. [43] proposes an approach to identify and correct defects in Android applications related to system settings, they addressed a problem similar to the one we investigated. In their work, system settings can include screen orientation, screen brightness, and system language, among other settings that can be defined by the user (enabling or disabling) or during the usage of applications. While in our work, we also use resources such as Wi-Fi, Bluetooth, and Location. Their work describes an experiment to evaluate the proposed approach's effectiveness in identifying defects related to system settings in Android applications. Their results showed that the proposed approach could identify several defects not found by other testing techniques, making it a valuable tool for Android application developers. In our work, we insert artificial faults and use instrumentation to activate and deactivate resources through informed configuration.

Marinho et al. [29] evaluated five sampling strategies in the context of resource-related failures of mobile applications. They generated and analyzed settings for the selected sampling testing strategies: Random, One-Enabled, One-Disabled, Most-Enabled-Disabled, and Pairwise. They observed that the Random strategy found better results concerning settings with failures. We also did a study to understand failures in mobile applications considering the 14 resources used in their work. In addition, we focused on analyzing SBFL to better understand resource-related failures in mobile applications. Finally, they commented on the challenges and difficulties in testing mobile applications concerning verifying failures arising from resource interactions. For example, they observed some challenges: the need for tooling support, the instrumentation of the test suite, and the time needed to test different settings. In order to overcome these challenges, we use SBFL as a technique that can support and reduce testing costs for mobile applications while considering the resources related to failures.

8 CONCLUSION

In this paper, we evaluated the use of SBFL aiming to locate faults in 8 Android applications and evaluate the sensitivity to resource interactions. We used faults seeded from a subset of mutation operators aiming to conduct the experimental study. Moreover, we used the Ochiai coefficient for calculating the suspiciousness score.

As a result, SBFL is able to rank more than 75% of the faulty code in 6 out of 8 applications. We found a major influence of resource settings on the suspiciousness score. That is, for the same failure (i.e., mutant), the ranking of suspicious methods varies depending on the combination of enabled resources (e.g., Wi-Fi and GPS). Therefore, we believe that SBFL is a promising technique that should be used in further studies to characterize the faults behind resource interaction failures.

As future work, we suggest the expansion of the experimental study to include applications implemented in other languages and frameworks, such as Kotlin, Flutter and React Native. Moreover, we can investigate specific mutation operators of mobile applications [13]. Another direction is the use of fault localization families [57] and empirical studies of the SBFL using real faults [36].

AVAILABILITY OF ARTIFACTS

We make our data publicly available for further investigations on a GitHub repository [11].

ACKNOWLEDGMENTS

This research was partially supported by CNPq (Grant 312920/2021-0) and FAPEMIG (Grant PPM-00651-17).

REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J.C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION)*. 89–98.

[2] Rui Abreu, P. Zoeteweij, and A. J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*. 39–46.

[3] AnkiDroid. Accessed 6-May-2023. Anki flashcards on Android. https://github.com/ankidroid/Anki-Android

[11] https://github.com/sbflappres/sbflappres_int

[4] S. Apel, D. Batory, C. Kästner, and G. Saake. 2013. *Feature-oriented software product Lines*. Springer Berlin / Heidelberg.

[5] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. 2013. Exploring Feature Interactions in the Wild: The New Feature-Interaction Challenge. In *International Workshop on Feature-Oriented Software Development (FOSD)* (Indianapolis, Indiana, USA) *(FOSD '13)*. 1–8.

[6] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2022. Evaluating and Improving Unified Debugging. *IEEE Transactions on Software Engineering* 48, 11 (2022), 4692–4716.

[7] T. F. Bowen, FS Dworack, C. Chow, N. Griffeth, G. E Herman, and Y-J Lin. 1989. The feature interaction problem in telecommunications systems. In *International Conference on Software Engineering for Telecommunication Switching Systems (SETSS)*. 59–62.

[8] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella. 2015. Do Automatically Generated Test Cases Make Debugging Easier? An Experimental Assessment of Debugging Effectiveness and Efficiency. *ACM Transactions on Software Engineering and Methodology* 25, 1, Article 5 (2015), 38 pages.

[9] M. B. Cohen, M. B. Dwyer, and J. Shi. 2007. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 129–139.

[10] J. Dąbrowski, E. Letier, A. Perini, and A. Susi. 2022. Analysing app reviews for Software Engineering: a systematic literature review. *Empirical Software Engineering* 27, 43 (2022), 1–63.

[11] H. A. de Souza, D. Mutti, M. L. Chaim, and F. Kon. 2018. Contextualizing spectrum-based fault localization. *Information and Software Technology* 94 (2018), 245–261.

[12] J. P. Diniz, C. -P. Wong, C. Kästner, and E. Figueiredo. 2021. Dissecting Strongly Subsuming Second-Order Mutants. In *IEEE International Conference on Software Testing, Verification, and Validation (ICST)*. 171–181.

[13] C. Escobar-Velásquez, M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk. 2020. Enabling Mutant Generation for Open- and Closed-Source Android Apps. *IEEE Transactions on Software Engineering (TSE)* 48, 1 (2020), 186–208.

[14] F. Ferreira, G. Vale, J. P. Diniz, and E. Figueiredo. 2021. Evaluating T-wise testing strategies in a community-wide dataset of configurable software systems. *Journal of Systems and Software (JSS)* (2021), 110990.

[15] J. A. Galindo, H. Turner, D. Benavides, and J. White. 2016. Testing variability-intensive systems using automated analysis: an application to Android. *Software Quality Journal (SQJ)* 24 (2016), 365–405.

[16] R. Gopinath, C. Jensen, and A. Groce. 2014. Code Coverage for Suite Evaluation by Developers. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 72–82.

[17] Ground. Accessed 6-May-2023. Ground mobile data collection app for Android. https://github.com/google/ground-android

[18] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. 1998. An Empirical Investigation of Program Spectra. *SIGPLAN Notices* 33, 7 (1998), 83–90.

[19] S. Heiden, L. Grunske, T. Kehrer, F. Keller, A. van Hoorn, A. Filieri, and D. Lo. 2019. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Software: Practice and Experience* 49, 8 (2019), 1197–1224.

[20] K. Holl and F. Elberzhager. 2019. Chapter One - Mobile Application Quality Assurance. Advances in Computers, Vol. 112. Elsevier, 1–77.

[21] J. A. Jones, M. Jean Harrold, and J. Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 467–477.

[22] M. C. Júnior, D. Amalfitano, L. Garcés, A. R. Fasolino, Stevão A. Andrade, and M. Delamaro. 2022. Dynamic Testing Techniques of Non-functional Requirements in Mobile Apps: A Systematic Mapping Study. *ACM Computing Surveys (CSUR)* 54, 10s (2022), 1–38.

[23] P. Kong, L. Li, J. Gao, T. Riom, Y. Zhao, T. F. Bissyandé, and J. Klein. 2021. ANCHOR: locating Android framework-specific crashing faults. *Automated Software Engineering* 28, 2 (2021), 10.

[24] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. Le Le Traon, and A. Ventresque. 2017. Assessing and Improving the Mutation Testing Practice of PIT. In *IEEE International Conference on Software Testing, Verification, and Validation (ICST)*. 430–435.

[25] M. Linares-Vásquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk. 2017. How do Developers Test Android Applications?. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 613–622.

[26] Y. Lu, M. Pan, J. Zhai, T. Zhang, and X. Li. 2019. Preference-wise testing for Android applications. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 268–278.

[27] C. Luo, J. Goncalves, E. Velloso, and V. Kostakos. 2020. A survey of context simulation for testing mobile context-aware applications. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–39.

[28] P. Machado, J. Campos, and R. Abreu. 2013. MZoltar: Automatic Debugging of Android Applications. In *International Workshop on Software Development Lifecycle for Mobile (DeMobile)*. 9–16.

[29] E. H. Marinho, F. Ferreira, J. P. Diniz, and E. Figueiredo. 2023. Evaluating testing strategies for resource related failures in mobile applications. *Software Quality Journal* (2023), 1–27.

[30] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology* 5, 2 (1996), 99–118.

[31] J. Oliveira, M. Souza, M. Flauzino, R. Durelli, and E. Figueiredo. 2022. Can Source Code Analysis Indicate Programming Skills? A Survey with Developers. In *International Conference on the Quality of Information and Communications Technology (QUATIC)*. 156–171.

[32] Openscale. Accessed 6-May-2023. Open-source weight and body metrics tracker, with support for Bluetooth scales. https://github.com/oliexdev/openScale

[33] Owntracks. Accessed 6-May-2023. Open-source weight and body metrics tracker, with support for Bluetooth scales. https://github.com/owntracks/android

[34] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman. 2019. Mutation Testing Advances: An Analysis and Survey. Advances in Computers, Vol. 112. Elsevier, 275–378.

[35] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2021. A Survey of Flaky Tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1, Article 17 (2021), 74 pages.

[36] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. 2017. Evaluating and Improving Fault Localization. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 609–620.

[37] PocketHub. Accessed 6-May-2023. PocketHub Android App. https://github.com/pockethub/PocketHub

[38] M. Polo-Usaola and I. Rodríguez-Trujillo. 2021. Analysing the combination of cost reduction techniques in Android mutation testing. *Software Testing, Verification and Reliability* 31, 7 (2021), e1769.

[39] RadioDroid. Accessed 6-May-2023. Radio Browser App. https://github.com/segler-alex/RadioDroid

[40] Q. I. Sarhan and Á. Beszédes. 2022. A Survey of Challenges in Spectrum-Based Software Fault Localization. *IEEE Access* 10 (2022), 10618–10639.

[41] H. N. Silva, J. Prado Lima, S. R. Vergilio, and A. T. Endo. 2022. A mapping study on mutation testing for mobile applications. *Software Testing, Verification and Reliability* 32, 8 (2022), e1801.

[42] T. Su, L. Fan, S. Chen, Y. Liu, L. Xu, G. Pu, and Z. Su. 2020. Why my app crashes? Understanding and benchmarking framework-specific exceptions of Android apps. *IEEE Transactions on Software Engineering (TSE)* (2020), 1115–1137.

[43] J. Sun, T. Su, J. Li, Z. Dong, G. Pu, T. Xie, and Z. Su. 2021. Understanding and Finding System Setting-Related Defects in Android Apps. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 204–215.

[44] Threema. Accessed 6-May-2023. Threema App for Android. https://github.com/threema-ch/threema-android

[45] P. Tramontana, D. Amalfitano, N. Amatucci, and A. R. Fasolino. 2019. Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal (SQJ)* 27, 1 (2019), 149–201.

[46] R. H. Untch, A. J. Offutt, and M. J. Harrold. 1993. Mutation Analysis Using Mutant Schemata. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 139–148.

[47] I. K. Villanes, A. T Endo, and A. C. Dias-Neto. 2022. A multivocal literature mapping on mobile compatibility testing. *International Journal of Computer Applications in Technology* 69, 2 (2022), 173–192.

[48] Y. Wang, M. V. Mäntylä, Z. Liu, J. Markkula, and P. Raulamo-jurvanen. 2022. Improving test automation maturity: A multivocal literature review. *Software Testing, Verification and Reliability* 32, 3 (2022), e1804.

[49] H. M. Win, S. H. Tan, and Y. Sui. 2023. Event-aware precise dynamic slicing for automatic debugging of Android applications. *Journal of Systems and Software (JSS)* (2023), 111606.

[50] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. 2012. *Experimentation in Software Engineering*. Springer Berlin / Heidelberg.

[51] W. E. Wong, V. Debroy, R. Gao, and Y. Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Transactions on Reliability* 63, 1 (2014), 290–308.

[52] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[53] WordPress. Accessed 6-May-2023. WordPress for Android. https://github.com/wordpress-mobile/WordPress-Android

[54] A. Zakari, S. P. Lee, K. A. Alam, and R. Ahmad. 2019. Software fault localisation: a systematic mapping study. *IET Software* 13, 1 (2019), 60–74.

[55] C. Zamfir and G. Candea. 2010. Execution Synthesis: A Technique for Automated Software Debugging (EuroSys). In *European Conference on Computer Systems*. 321–334.

[56] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid. 2021. An Empirical Study of Boosting Spectrum-Based Fault Localization via PageRank. *IEEE Transactions on Software Engineering* 47, 6 (2021), 1089–1113.

[57] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. 2021. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering* 47, 2 (2021), 332–347.