RIFDiscoverer: A Tool for Finding Resource Interaction Failures

Euler Marinho
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

Izaia Machado Pessoa Neto
Federal University of Ceará
Sobral, Ceará, Brazil

Fischer Ferreira
Federal University of Itajubá
Itabira, Minas Gerais, Brazil

João Paulo Diniz
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

Eduardo Figueiredo
Federal University of Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

ABSTRACT

Mobile application quality has become a crucial aspect. In spite of the numerous studies concerning testing methods, tools, and techniques, the field of mobile application testing is still under development. In mobile applications, resource interaction failures occur when resources influence the behavior of other resources. They can compromise the application and harm the user experience. This paper presents RIFDiscoverer a tool for assisting developers and testers to deal with resource interaction failures in Android applications. Our preliminary results indicate a potential of the tool to find resource interaction failures. The source code and a demo video are available on GitHub at https://github.com/bytestaking/RIFDiscoverer.

CCS CONCEPTS
- Software and its engineering → Software testing and debugging.

KEYWORDS
Software Quality, Software Testing, Resource Interaction Failures

ACM Reference Format:

1 INTRODUCTION

Mobile applications have been developed not only for entertainment purposes but also for targeting safety and critical domains [3]. As a consequence, the quality of mobile applications has become a crucial aspect, for instance, by promoting the use of testing as a quality assurance practice [15, 16]. However, despite the availability of testing methods, techniques, and tools, the field of mobile application testing is still under development [9]. These applications are often executed on a variety of platform configurations [11] and each platform configuration has different platform resources.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC’24, September 2-6, 2024, Luxembourg
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

2 BACKGROUND AND RELATED WORK

This section discusses resource interaction failures (Section 2.1) and presents an overview of sampling testing strategies (Section 2.2).

2.1 Resource Interaction Failures

We define "resource interaction failures" as failures that occur when resources influence the behavior of other resources. This definition is inspired by the feature interaction problem in configurable systems [4, 5]. They have been shown to be effective in involving the selection of a subset of input combinations. Sampling strategies are a well-known technique, such as in the domain of configurable systems [4]. However, these strategies are often used in research efforts aiming to deal with scalability factors, for example, when interacting with physical or virtual device farms [18]. Our preliminary results indicate a potential of the tool to find resource interaction failures.

This paper is organized as follows. In Section 2, we discuss background information and some related work. In Section 3, we present the architecture, some design and implementation aspects of RIFDiscoverer. Section 4 presents a preliminary evaluation of the tool. Section 5 presents some concluding remarks.
systems [4]. Our study includes 12 resources often used by Android applications and present in most devices: Location, Wi-Fi, Mobile Data, Bluetooth, Auto Rotate, Battery Saver, Do Not Disturb, Accelerometer, Gyroscope, Magnetometer, Proximity, and Camera.

Figure 1 presents a code excerpt of Wikimedia Commons Android app, showing a case of a resource interaction failure [24]. This open source application allows users to upload pictures from the device to Wikimedia Commons, the image repository for Wikipedia1. The Android Platform supports the positioning via GPS or network (Wi-Fi/Mobile Data). The issue describes a situation involving the crash of the application when it is opened and both GPS and network are disabled [6]. The failure is caused by the call of getLastKnownLocation to get the current location via network (line 3). However, this call returns a null value which is later used in the construction of an object to store the location-related values (line 5). As a result, the application crashes because of a NullPointerException.

```java
1  locationManager.getLastKnownLocation(
2     LocationManager.GPS_PROVIDER);  
3  if (lastKL == null) {    
4     lastKL = locationManager.getLastKnownLocation(  
5           LocationManager.NETWORK_PROVIDER);  
6  }  
7  return LatLng.from(lastKL); //An object is constructed from the latitude and longitude coordinates
```

Figure 1: Code Excerpt from Wikimedia Commons app.

### 2.2 Sampling Testing Strategies

As the exhaustive exploration of the input space in configurable software systems is often very expensive or even impractical (for instance, by brute-force), an alternative to balance the effort and the failure-detection capability is to use sampling testing strategies [1, 10, 21, 23]. The use of sampling testing has been promising to find feature interaction failures [1, 10, 21, 23] and resource interaction failures [20]. For instance, the strategy One-Disabled [1] selects settings with only one resource disabled and all other resources enabled. The strategy One-Enabled selects settings with only one resource enabled and the other resources disabled. The strategy Most-Enabled-Disabled combines two sets of samples: one set in which most of the resources are enabled and another set in which most of the resources are disabled. In the case when constraints between resources do not exist, it establishes two settings: one with all resources enabled and the other with all resources disabled [21]. The strategy Random creates n distinct settings with all resources randomly enabled or disabled. We used the implementation of this strategy present in FeatureIDE [25].

### 2.3 Related Work

Table 1 provides a brief comparison between our tool and tools of similar previous studies. In each case, the columns refer to the individual studies. The rows show the characteristics of the studies, such as which strategy or technique was used. We use a dash ‘-’ to indicate information not available in the respective study. To our knowledge, RIFDiscoverer is the only one of the tools presented that explicitly works with a wide range of device sensors.

### 3 THE RIFDISCOVERER TOOL

This section provides a comprehensive overview of the RIFDiscoverer architecture (Section 3.1), design and implementations concerns taken into consideration (Section 3.2) and the instrumentation used for testing (Section 3.3).

#### 3.1 Architecture

Figure 2 presents the general architecture of RIFDiscoverer. The application requires the user to choose one available testing strategy (One-Disabled, One-Enabled, Most-Enabled-Disabled, Pairwise, Random, Custom, IncLing [2], CASA [12], Chvatal [14], ICPL [13]) and specify the path where the instrumented Android project is located. Once the testing strategy and execution parameters are defined, the application communicates with the WebSocket Server to start the execution of the tests. Before test execution, the WebSocket Server starts Genymotion Android emulator. Test Execution Manager receives all settings and the number of executions, and then it executes them one by one. Note that for each execution, the order in which the settings are processed is random. The setting processing script is responsible for executing the Gradle task that runs the test suite, with only the enabled resources for the current setting on the emulated device. After finishing the tests for a setting, the WebSocket Server sends a message to the Application to update a progress bar and the same is done for the following setting.

![Architecture of RIFDiscoverer](image)

#### 3.2 Design and Implementation

The two key components of RIFDiscoverer are the application front-end through which the user interacts with the tool and the WebSocket Server that rapidly runs scripts and communicates with the other application components.

For the front-end, we aim for an easy to use interface and a way for the interface to communicate directly to Python running on the user machine. For this reason, we choose Python EEL2.

1. https://commons-app.github.io/
2. https://github.com/python-eel/Eel
Table 1: Comparison of Related Tools

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Strategies-Techniques</td>
<td>Android Sampling</td>
<td>Android Static Analysis</td>
<td>Android Pairwise</td>
<td>Android Each-Choice</td>
<td>Android Setting-wise metamorphic fuzzing</td>
</tr>
<tr>
<td>Comparison Elements (Resources, hardware options)</td>
<td>14</td>
<td>-</td>
<td>4</td>
<td>5</td>
<td>&gt; 50</td>
</tr>
</tbody>
</table>

This framework allows for the development of a web-based front-end while maintaining a Python back-end. It facilitates seamless interaction between the user interface and the underlying Python code, enabling the front-end to directly call Python implemented functions.

WebSocket Server is a separate component that uses Python WebSockets library and runs alongside EEL, but in a separate thread. This design was implemented to handle the execution of scripts that takes a long time to complete, ensuring the application remains responsive throughout the execution of testing strategies. By running in a separate thread, WebSocket Server can manage lengthy operations without blocking the application thread. This approach ensures that users receive updates and can interact with the interface even during extensive testing processes.

3.3 Test Instrumentation

We implemented a test instrumentation based on the UI Automator framework to control the resources. The instrumentation is based on Android instrumented tests, i.e., a type of functional test. They execute on devices or emulators and can interact with Android framework APIs. We are to manage 7 resources interacting with Android Quick Settings: Auto Rotate, Battery Saver, Bluetooth, Do Not Disturb, Location, Mobile Data, and Wi-Fi. We control the other resources using third-party applications. For instance, Camera is controlled by Lens Cap and the sensors (Accelerometer, Gyroscope, Magnetometer, Proximity) are managed by Sensor Disabler. The test instrumentation consists of the function AdjustResourceStates presented in Algorithm 1. For the required setting S, we enable (line 6) or disable (line 8) each resource state (line 4) according to the state specified in the pair.

We implemented Resource_setup as a class with a static method annotated with BeforeClass. We extended each class of the test suites with the implemented class. Therefore, the execution of tests of a certain class is preceded by the execution of the setup method. In the current implementation, we perform the verification of resource state (line 5) via Android APIs, such as LocationManager for the Location and TelephonyManager for the Mobile Data. In other cases, we use the UI Automator features to find some screen widgets related to the resource state. For example, we inspect the sensors states by processing screens of Sensor Disabler. It is important to emphasize that in our implementation the resources are only adjusted (lines 6 and 8) if necessary. Besides, we modified the build scripts in order to use the Android Test Orchestrator, a tool that helps minimize possible shared states, a known factor associated to flaky tests and isolate the crashes.

Algorithm 1 Resource_setup

```java
1: Input
2: S list of <resource, state> pairs
3: procedure AdjustResourceStates(S)
4: for all pair ∈ S do
5: if pair.state == true then
6: ENABLE(pair.resource)
7: else
8: DISABLE(pair.resource)
9: end if
10: end for
11: end procedure
```

We implemented Algorithm 2 for managing the executions of the instrumented test suites. We used three executions (line 6) to deal with flaky tests, and shuffled the settings to minimize order dependencies between tests (line 8). Multiple execution is a common strategy for detecting flaky tests. However, the optimal number of re-executions to identify flaky tests is not defined [22]. One study suggests a maximum of five re-executions [17]. Based on previous essays, we set the default number of re-executions to three. We took into account our time constraints for the experiments and made the observation that this number is sufficient to detect flaky tests. We call the function AdjustResourceStates (line 10) defined in Algorithm 1 to adjust the states of all resources.

4 PRELIMINARY EVALUATION

To evaluate the usefulness of RIFDiscover, we applied it to Threema, a messenger focused on privacy and security. Table 2 presents the characteristics of the study. The declared resources are identified from the static analysis of the Android Manifest file. The tool takes as input settings, i.e., lists of pairs (resource, state) where state can be True or False depending on whether the resource is enabled or disabled. Our preliminary found a one resource interaction failure in Threema. The failure we found (by a test named "testNotificationWithoutAction") happens when Do Not Disturb is enabled.

11https://developer.android.com/training/testing/unit-runner
12https://github.com/threema-ch/threema-android

Published: September 2-6, 2024, Luxembourg
Algorithm 2 Test_execution_manager

1: Input
2: AP application with extended tests
3: SL list of settings
4: Output
5: TR test reports
6: maxExecutions ← 3
7: for exec ← 1 to maxExecutions do
8: SHUFFLE(SL)
9: for all st ∈ SL do
10: ADJUSTRESOURCESTATES(st)
11: Execute the whole test suite of AP
12: end for
13: end for

### Table 2: Evaluation of Threema Case Study.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>238,045</td>
</tr>
<tr>
<td>Test LOC</td>
<td>1,931</td>
</tr>
<tr>
<td>Test Cases</td>
<td>50</td>
</tr>
<tr>
<td>Settings</td>
<td>Random(30), One-Disabled(12), One-Enabled(12), Pairwise(8), Most-Enabled-Disabled(2)</td>
</tr>
<tr>
<td>Declared Resources</td>
<td>Bluetooth, Camera, Location, Mobile data, Wi-Fi</td>
</tr>
<tr>
<td>Failed Test Cases</td>
<td>1</td>
</tr>
</tbody>
</table>

This failure could let the user to loose an error notification, harming its experience.

5 CONCLUSION

This paper presented RIFDiscoverer, a tool that helps developers and testers to deal with resource interaction failures. We have chosen a simple but responsive user interface for RIFDiscoverer. We described its architecture, design and implementation aspects, and performed a preliminary evaluation. By using an extensible architecture, it can be used in research efforts aiming to deal with scalability factors, for example, when interacting with physical or virtual device farms [18]. We plan further improvements on RIFDiscoverer implementation. Currently, we are planning an adaptation of the tool as a Web Application to be hosted in cloud computing platforms. Future studies can involve empirical studies with application developers, deal with other kind of resources, and the portability of the tool to other mobile platforms.

REFERENCES


A SCREENS

Figure 3 presents the first screen of RIFDiscoverer. This screen shows fields to the selection of testing strategies (A) and a “T” parameter (B) depending on the selected strategy. A part of the screen (C) presents a table of settings with markers for enabled and disabled resources. Some strategies (Random and Custom) allow direct manipulation of the settings.

Figure 3: Choice of Testing Strategies.

Figure 4 presents the second screen of RIFDiscoverer. This screen allows changing some execution parameters: the folders of application project and generated reports of Gradle verification task, the number of executions for dealing with flaky tests, and the Gradle task name related to the execution of the instrumented test suite.

Figure 4: Changing Execution Parameters.

Figure 5 presents the third screen of RIFDiscoverer. This screen allows managing the execution of the instrumented test suite. A part of the screen (A) is a terminal with output logs from test suite executions. A button (B) loads the Genymotion emulator preconfigured to the execution of instrumented code. Another button (C) starts the execution of the test suite.

Figure 5: Running the Test Suite.

Figure 6 presents the third screen of RIFDiscoverer side by side with the emulator. The screen presents a progress bar (A) and buttons to abort the execution (B) and visualize the folder of the test reports (C).

Figure 6: Running the Test Suite with Emulator Loaded.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009