Resource Interaction Failures in Mobile Applications: A Challenge for the Software Product Line Testing Community

Euler Marinho  
Federal University of Minas Gerais  
Belo Horizonte, Minas Gerais, 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 Fernandes  
University of Southern Denmark  
Odense, Denmark

ABSTRACT

Context: Many mobile applications run on multiple platforms with specific available resources. These resources are associated with communication capabilities, sensors, and user customization. Certain resource combinations imply interactions between resources that are likely to produce failures in mobile applications, thereby harming the user experience.

Challenge: There may be a large number of resource combinations for a single mobile application. Consequently, exhaustively testing resource interactions to spot failures can be very challenging. However, in order to address this challenge, having robust, well-documented, and publicly available datasets for mobile application testing is necessary.

Proposal: This paper proposes the Resource Interaction Challenge targeting mobile applications. We introduce a curated dataset of 20 mobile applications with varying sizes (up to 350K lines of code) and required resources (Bluetooth, Wi-Fi, etc.). Due to the shortage of sampling strategies for testing resource interactions in mobile applications, we opted for strategies commonly used for configurable systems in general. Our dataset includes failures detected and source code metrics computed for each mobile application.

Conclusion: We expect to engage both researchers and practitioners in reusing our dataset, especially to propose and evaluate novel testing strategies.

CCS CONCEPTS

• Software and its engineering → Software configuration management and version control systems; Software testing and debugging

KEYWORDS

Mobile Application Testing; Resource Interaction Failures;
the case of configurable systems [5, 12, 16] in which all tests must be executed in several configurations. An alternative for decreasing the testing effort is the use of sampling strategies involving the selection of a subset of input combinations. Sampling strategies are a well-known technique, such as in the domain of configurable systems [5]. Several sampling strategies have been proposed and investigated in the literature, such as \( t \)-wise [16], one-disabled [1], and most-enabled-disabled [27]. They have been shown to be effective in finding faults, even with the number of combinations tested much lower than the universe of all possible combinations [27, 40].

Resource interaction failures have been explored in mobile applications testing [24, 26, 42]. However, the investigation of these failures is a still little explored aspect of research. We lack work to evaluate resource interaction failures in real mobile applications, verify which resources are most related to failures, and investigate the faults behind these failures. Testers may neglect to properly test mobile applications considering the interaction of resources due to a lack of knowledge of such failures. Therefore, resource interaction failures may occur in the everyday use of the mobile application but they may be imperceptible in the testing phase.

In this context, we contribute to the research of mobile application testing with a dataset composed of 20 mobile applications with instrumented tests proposed in our prior work [26]. We propose the use of the number of failures traditional metric and a metric for verifying the efficiency of the proposed testing strategy. Through these metrics, it is possible to measure the fault-detection capability of the proposed testing strategies and the most efficient testing strategy.

**Challenge:** the participants must propose testing strategies for mobile applications taking resource interactions into account. The failure detection capability and the efficiency of the strategy must be higher than our baseline. In other words, the proposed testing strategies increase the number of unique detected failures and minimize the number of tested settings.

The dataset of mobile applications with instrumented test suites and reports of failures can be found at:

https://eulerhm.github.io/splc-challenge/

Therefore, we challenge the research community to use their testing strategies to find resource interaction failures in the target applications of our proposed dataset. We argue that the provided dataset is well suited as subjects for the challenge of finding resource failures due to the variety of mobile applications and because each application has an instrumented test suite. We expect participants to evaluate their solutions by measuring how strongly their testing strategies can be in finding resource interaction failures in the applications of our dataset. Each solution could be assessed concerning how efficient the solution is for testing mobile applications.

2 BACKGROUND

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 [5]. Our study includes 14 resources often used by Android applications and present in most devices. Some resources are directly manageable by mobile device users (for instance, Location and Mobile Data) whereas others are restricted only to more advanced users (e.g., Accelerometer and Gyroscope).

Among the target resources, three are used to manage networks and connections (Bluetooth, Mobile Data, and Wi-FI), six to control sensors (Accelerometer, Gyroscope, Light, Magnetometer, Orientation, and Proximity), one to control a device’s hardware element (Camera), one to control a privacy option (Location), and three to manage user-controlled options (Auto Rotate, Battery Saver, and Do Not Disturb).

Figure 1 presents a code excerpt from Wikimedia Commons Android app, showing a case of a resource interaction failure [42]. 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 [7]. 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.

```
1 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 1: Code Excerpt from Wikimedia Commons app.

In another example, we illustrate by means of an issue of Traccar Client2 how the combination of platform resources and application settings may lead to an unexpected failure of the application. Traccar Client is an open source application available for download at Google Play Store. In summary, it is a GPS Tracker, which communicates with its own application server. Traccar Client has a configuration called Accuracy, which can be set to three values: High, Medium, or Low. To achieve the Accuracy High, it is necessary that the GPS, Wi-Fi, Mobile Data, and Bluetooth sensors are enabled on the smartphone. According the the issue #390, opened at the Traccar issue manager at GitHub3, it can be seen that, even

---

1. https://commons-app.github.io/
2. https://www.traccar.org
if the four sensors are enabled, the application stops changing loca-
tion when its Accuracy is set do Medium. However, works again
for the other two possible values, i.e., High and Low.

2.2 Sampling Testing Strategies
Sampling strategies are commonly used to test configurable soft-
ware systems [1, 16, 27, 40]. Exhaustive testing of configurable
systems encompasses the exploration of a configuration space, i.e.,
the combination of all input options that can be used to configure a
system [40]. The validity of a configuration is often determined by
a feature model. As the exhaustive exploration of this space 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. The effort can be measured
considering the size of the sample set (related to the test execution
time), whereas the failure-detection capability can be associated to
the number of failures detected by the sampled configurations [27].

The use of sampling testing has been promising to find feature
interaction failures [1, 16, 27, 40] and resource interaction fail-
ures [26]. 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 re-
sources are enabled and other 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 one with all resources disabled [27]. The strategy Random
creates $n$ distinct settings with all resources randomly enabled
or disabled. We used the implementation of this strategy present in
FeatureIDE [44].

In a $t$-wise combinatorial interaction testing, each combination
of $t$ resources is required to appear in at least one setting of
the sample, i.e., only the subset of settings that covers a valid group
of $t$ resources being enabled and disabled actually matters [31].
Generating such configurations can be modeled as a covering array
problem instance. However, this optimization is NP-hard and sev-
eral heuristics have been proposed to perform $t$-wise sampling [2].

In this context, we encourage the community to use our dataset
as it provides instrumented test suites for each application. As an
advantage of our challenge, testing strategies can benefit from the
available test suites. Moreover, the dataset of mobile applications
and resource interaction failures is a unique opportunity for us to
characterize them. For instance, a deep understanding of resource
interaction failures in mobile applications may help practitioners
to identify the reasons for failures that occur in their applications.

3 DATASET OVERVIEW
In this section, we provide an overview of the proposed dataset.
Section 3.1 presents the metrics that characterize the proposed
dataset. We provide a summary of the dataset in Section 3.2. Section
3.3 shows an overview of the test suite instrumentation process
for mobile applications. Section 3.4 presents a motivating example
of using the dataset. In Section 3.5, we present the failures found
in the proposed dataset. Finally, Section 3.6 describes the dataset
artifacts.

3.1 Evaluation Metrics
For a better comprehension of the subject applications in our dataset,
we collected static metrics with two different tools. Metrics related
to the size of the mobile applications (e.g., lines of code and num-
ber of packages) were computed by CLOC tool and CodeMR.
CLOC [10] is an open-source tool to count lines of source code in
multiple programming languages. CodeMR [11] is a static analysis
tool for Java and Kotlin languages, installed as a plugin for Android
Studio IDE.

3.2 Mobile Application Dataset
Table 1 presents an overview of the 20 mobile applications that
compose our dataset. We provide additional information about the
proposed dataset in our supplementary website. These applica-
tions belong to different categories, named according to the Play
Store categories, with a large variation of size and test code size.
The columns of Table 1 represent the applications’ name, descrip-
tion, size, test suite metrics, and resources. We discuss each of these
metrics next.

Size metrics. We selected applications of different sizes. We
measure the number of lines of code (#LOC), packages (#Packages),
classes (#Classes), and methods (#Methods). The applications in
our dataset vary from 455 lines of code (MoonShot) to more than
347,000 lines of code (WordPress-Android). Similarly, while Nekome
has 29 classes and WordPress-Android has almost 4,180 classes.

Test suite metrics. We report the number of test cases and the
lines of code of each test suite. The variety of sizes and charac-
teristics of the applications of our dataset can be a challenge for
candidate solutions. In this way, we encourage participants to apply
their strategies to our dataset and report on which situations their
test strategies provide the best results.

3.3 Test Suite 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. The following resources are manageable by mobile
device users: Auto Rotate, Battery Saver, Bluetooth, Do Not Disturb,
Location, Mobile Data, and Wi-Fi. Therefore, we enable or disable
these resources interacting with Android Quick Settings. We control
the other resources using third-party applications. For instance,
Camera is controlled by Lens Cap and the sensors are managed by
Sensor Disabler. This application requires a rooted Android
device.

In this study, we named an input combination as a setting, i.e., a
set of resources whose states (enabled or disabled) are previously
defined. A setting is a 14-tuple of pairs (resource, state) where
state can be True or False depending on whether the resource is
enabled or disabled. The test instrumentation consists of the function
AdjustResourceStates presented in Algorithm 1. For the

1https://eulerhm.github.io/splc-challenge/
2https://developer.android.com/training/testing/ui-automator
3https://developer.android.com/training/testing/instrumented-tests
4https://github.com/percula/LensCap
5https://github.com/wardellbagby/Sensor-Disabler
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 clear [35]. One study suggests a maximum of five re-executions [22]. Based on previous essays, we set the number of re-executions to three. We empirically 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.

### Algorithm 2 Test_execution_manager

1. **Input**
   - AP application with extended tests
   - SL list of settings

2. **Output**
   - TR test reports

3. **maxExecutions** ← 3

4. for **exec** ← 1 to **maxExecutions** do

5. **SHUFFLE**(SL)

6. for all st ∈ SL do

7. **ADJUSTRESOURCESTATES**(st)

8. Execute the whole test suite of AP

9. end for

10. end for

### 3.4 Example of Use

In this section, we present an example of mobile application in our dataset. We show our framework for running the test suite. With the support of our framework, participants can use the test suite for each setting that their testing strategies produce. Vocable is a communication tool for individuals who are speech impaired. According to the application website [45], Vocable uses the ARCore SDK [37] to track the user’s head movements and understand where the user is looking on the screen.

We implemented Algorithm 1 for managing 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 clear [35]. One study suggests a maximum of five re-executions [22]. Based on previous essays, we set the number of re-executions to three. We empirically 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.

### Algorithm 1 Resource_setup

1. **Input**
   - $S$, list of < resource, state > pairs

2. **procedure** AdjustResourceStates($S$)

3. for all pair ∈ $S$ do

4. if pair.state == true then

5. ENABLE(pair.resource)

6. else

7. DISABLE(pair.resource)

8. end if

9. end for

10. end procedure

### Table 1: The Dataset

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Category</th>
<th>Commits</th>
<th>LOC</th>
<th>#Packages</th>
<th>#Classes</th>
<th>#Methods</th>
<th>Test Suite metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ground [18]</td>
<td>Productivity</td>
<td>Productivity</td>
<td>4,936</td>
<td>19,906</td>
<td>51</td>
<td>412</td>
<td>1,676</td>
<td>4</td>
</tr>
<tr>
<td>Lockwise [23]</td>
<td>Reference</td>
<td>Reference</td>
<td>503</td>
<td>14,535</td>
<td>12</td>
<td>350</td>
<td>1,124</td>
<td>38</td>
</tr>
<tr>
<td>Mixin-Messenger [28]</td>
<td>Finance</td>
<td>Finance</td>
<td>8,086</td>
<td>168,080</td>
<td>136</td>
<td>2,212</td>
<td>12,625</td>
<td>160</td>
</tr>
<tr>
<td>Moonahel [29]</td>
<td>Tools</td>
<td>Tools</td>
<td>351</td>
<td>455</td>
<td>7</td>
<td>48</td>
<td>196</td>
<td>28</td>
</tr>
<tr>
<td>Nekome [30]</td>
<td>Productivity</td>
<td>Productivity</td>
<td>2,742</td>
<td>1,084</td>
<td>4</td>
<td>29</td>
<td>62</td>
<td>64</td>
</tr>
<tr>
<td>NL-covalid9 [32]</td>
<td>Medical</td>
<td>Medical</td>
<td>1,293</td>
<td>65,839</td>
<td>26</td>
<td>311</td>
<td>634</td>
<td>20</td>
</tr>
<tr>
<td>OpenScale [33]</td>
<td>Health</td>
<td>Health</td>
<td>1,742</td>
<td>27,781</td>
<td>19</td>
<td>174</td>
<td>1,046</td>
<td>14</td>
</tr>
<tr>
<td>OwnTracks [34]</td>
<td>Fitness</td>
<td>Fitness</td>
<td>1,995</td>
<td>14,499</td>
<td>37</td>
<td>273</td>
<td>1,176</td>
<td>27</td>
</tr>
<tr>
<td>PocketHub [36]</td>
<td>Travel, Local</td>
<td>Travel, Local</td>
<td>3,512</td>
<td>29,001</td>
<td>40</td>
<td>323</td>
<td>1,332</td>
<td>107</td>
</tr>
<tr>
<td>Radio-Droid [37]</td>
<td>Productivity</td>
<td>Productivity</td>
<td>1,186</td>
<td>22,815</td>
<td>25</td>
<td>207</td>
<td>810</td>
<td>23</td>
</tr>
<tr>
<td>Scarlet-Notes [38]</td>
<td>Music</td>
<td>Music</td>
<td>636</td>
<td>4,260</td>
<td>5</td>
<td>51</td>
<td>228</td>
<td>52</td>
</tr>
<tr>
<td>Showly-1.0 [39]</td>
<td>Entertainment</td>
<td>Entertainment</td>
<td>3,255</td>
<td>2,547</td>
<td>8</td>
<td>29</td>
<td>173</td>
<td>55</td>
</tr>
<tr>
<td>SpaceX-Follower [41]</td>
<td>News</td>
<td>News</td>
<td>356</td>
<td>7,664</td>
<td>22</td>
<td>110</td>
<td>526</td>
<td>30</td>
</tr>
<tr>
<td>Threema [43]</td>
<td>Communication</td>
<td>Communication</td>
<td>238,045</td>
<td>238,045</td>
<td>123</td>
<td>1,718</td>
<td>9,763</td>
<td>54</td>
</tr>
<tr>
<td>Vocale [45]</td>
<td>Communication</td>
<td>Communication</td>
<td>863</td>
<td>13,188</td>
<td>14</td>
<td>106</td>
<td>534</td>
<td>14</td>
</tr>
<tr>
<td>Woo-Commerce [46]</td>
<td>Business</td>
<td>Business</td>
<td>26,527</td>
<td>156,962</td>
<td>120</td>
<td>2,283</td>
<td>8,334</td>
<td>27</td>
</tr>
</tbody>
</table>

---

11https://developer.android.com/reference/android/location/LocationManager
13https://developer.android.com/training/testing/junit-runner
The challenged participants are expected to use our framework to run the test suite with their testing solution. We emphasise that the proposed settings are not limited to validity rules as usually found in configurable systems, e.g. defined by a feature model. We present a small example to illustrate the use of our framework to call the test suite for each target system in our dataset. Figure 2 shows a setting for the mobile application Vocable. The settings must be in a CSV file with only the enabled resources listed. Figure 2 shows a setting with 6 resources enabled. Our instrumentation considers each line of the file as a setting. According with the line 8 of Algorithm 2, the list of settings is shuffled and the execution continues for each setting.

**Figure 2: Setting example**

To exemplify the output, we report a resource interaction failure for Vocable. The failure we found (e.g., by a test named “verifyDefaultTextAppears”) happens when Mobile data and Wi-Fi are both disabled. The failure generates an ARCore Fatal Exception.

### 3.5 Failure Report

Table 2 presents a summary of failure reports for each application in our dataset. We selected applications with failures manifested in three executions. As we can see, 10 applications present this kind of failure which represents 50% of the applications in our dataset. In addition, we can see that 409 failing settings (FS) were found. To illustrate this, we look at the data related to PocketHub for which we only found 4 failing settings.

We used five testing strategies [26]: Random, One-Enabled, One-Disabled, Most-Enabled-Disabled, and Pairwise as a baseline for reporting failures in mobile applications. A thorough test against all settings (214 = 16,384) is a costly and even impractical practice. Therefore, we have chosen the number of 68 settings in total for each application because it delivers the results in a feasible time. The number of settings generated by Random was limited to 30. One-Disabled and One-Enabled generated 14 settings each. Most-Enabled-Disabled generated 2 settings and Pairwise generated 8 settings. We emphasise that the list of settings for each application does not vary because we control the device resources, as explained in Section 2.1. The SE column of Table 2 presents a measurement of solution efficiency. We discuss this metric in Section 3.7. We make available on the dataset website the failure found, each setting that failed, and the test cases that observed the reported failures.

<table>
<thead>
<tr>
<th>Name</th>
<th>FS</th>
<th>SE</th>
<th>#Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>CovidNow</td>
<td>32</td>
<td>0.47</td>
<td>2</td>
</tr>
<tr>
<td>Lockwise</td>
<td>68</td>
<td>1.00</td>
<td>4</td>
</tr>
<tr>
<td>Mixin-Messenger</td>
<td>20</td>
<td>0.29</td>
<td>2</td>
</tr>
<tr>
<td>ni-covid19</td>
<td>55</td>
<td>0.81</td>
<td>6</td>
</tr>
<tr>
<td>OnceTracks</td>
<td>68</td>
<td>1.00</td>
<td>3</td>
</tr>
<tr>
<td>PocketHub</td>
<td>4</td>
<td>0.06</td>
<td>1</td>
</tr>
<tr>
<td>SpaceXBrowser</td>
<td>68</td>
<td>1.00</td>
<td>4</td>
</tr>
<tr>
<td>Threema</td>
<td>33</td>
<td>0.48</td>
<td>1</td>
</tr>
<tr>
<td>Vocable</td>
<td>24</td>
<td>0.35</td>
<td>7</td>
</tr>
<tr>
<td>WordPress-Android</td>
<td>37</td>
<td>0.54</td>
<td>11</td>
</tr>
</tbody>
</table>

FS stand for the number of failing settings. SE indicates the Solution Efficiency. #Failures stand for the unique detected failures. Total settings (68).

#### 3.6 Description of dataset artifacts

The challenge artifacts are available in the companion website of the dataset, organized into three items. We report the artifacts concerning the Vocable application. However, all other applications in our dataset follow the same structure.

1. **Source Code**: We provide the source code and test suite for each mobile application. These applications were implemented in Java and/or Kotlin.
2. **Found Failures**: We provide the found failures for the challenge applications. We present these failures in a CSV file that contains the identifier of the setting in which the failure occurred and the test cases that observed the failure.
3. **Analyzed Settings**: We provide the settings that we run with our baseline. The setting files follows the model described in Figure 2.

#### 3.7 Solution evaluation

This section presents the metric that we use to measure how efficient test strategies for mobile applications can be in observing failures. We assume that each failing test case corresponds to a unique failure despite the used test oracle.[8]. We provide a set of failures found in the systems in our dataset. However, it is possible to find other failures in unvisited settings. We calculate the efficiency according to Equation 1. Failingsettings is the amount of settings that cause at least one failure. Totalsettings represents the total of settings generated by all testing strategies.

\[
SE = \frac{\text{FailingSettings}}{\text{TotalSettings}} \tag{1}
\]

### 4 CONCLUSION

We proposed a dataset with 20 mobile applications with instrumented test suites as a challenge for testing strategies taking resource interactions into account. We provide two groups of metrics (size and test suite) to characterize the proposed dataset for the challenge. Moreover, we found and reported a total of 409 failing settings in 10 applications of the proposed dataset.

Several datasets for the mobile application have been used. However, this dataset is the first dataset for mobile applications with focused on resource interaction failures[26]. Furthermore, it is an excellent opportunity to share knowledge on testing strategies because we use the same test suite towards an unbiased comparison of the failure detection capability and the efficiency of testing strategies for mobile applications. We believe that our dataset can be a common point of comparison for testing strategies, and we encourage you to submit your solutions to the proposed challenge.

---

Acknowledgements. This research was partially supported by Brazilian funding agencies: CAPES, CNPq, and APMEP.

REFERENCES


