Evaluating testing strategies for resource related failures in mobile applications

Euler Horta Marinho[1,2*], Fischer Ferreira[1], João P. Diniz[1] and Eduardo Figueiredo[1*]

[1*]Computer Science Department, Federal University of Minas Gerais, Brazil.
[2]Computer and Systems Department, Federal University of Ouro Preto, Brazil.

*Corresponding author(s). E-mail(s): eulerhm@dcc.ufmg.br; figueiredo@dcc.ufmg.br;
Contributing authors: fischerjf@dcc.ufmg.br; jpaulo@dcc.ufmg.br;

Abstract

Mobile applications have been used for multiple purposes from entertainment to critical domains. 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. The diversity of mobile platforms is pervaded by several resources related to communication capabilities, sensors, and user-controlled options. As a result, applications can present unexpected behaviors and resource interactions can introduce failures that manifest themselves in specific resource combinations. These failures can compromise the mobile application quality and harm the user experience. We evaluate the failure-detection capability and effectiveness of five sampling testing strategies (`Random`, `One-Enabled`, `One-Disabled`, `Most-Enabled-Disabled`, and `Pairwise`) in the context of resource related failures in 15 mobile applications. We focus on 14 common resources of the Android platform and analyze the resource interactions more likely to cause failures. `Random` had great percentages of failing test cases, followed by `One-Enabled`, `Most-Enabled-Disabled`, and `Pairwise`. We observe that `One-Enabled` and `One-Disabled` are the most effective testing strategies for six and four applications, respectively. Surprisingly, resource pairs have more influence on failures than other resource combinations, varying widely among the applications.

Keywords: Mobile applications, Software testing, Sampling strategies, Resource interactions

1 Introduction

Mobile applications have been used for multiple purposes including entertainment, management of personal information, and control of devices, such as home security systems, health monitors, and cars (Amalfitano et al., 2017). Therefore, applications have been developed not only for entertainment purposes but also for targeting safety and critical domains. 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 (Júnior et al., 2022). The growing awareness of mobile applications quality has resulted in a broad spectrum of testing techniques (Júnior et al., 2022; Luo et al., 2020; Tramontana et al., 2019; Kong et al., 2018). However, despite the availability of testing methods, techniques, and tools, the field of mobile application testing is still under development (Escobar-Velásquez et al., 2020).

Mobile applications are often executed on a variety of platform configurations (Galindo et al., 2016) and each platform configuration has different platform resources. These resources may be related to communication capabilities (e.g., Wi-Fi, Bluetooth, Mobile Data, and GPS), sensors (e.g., Accelerometer, Gyroscope, and Magnetometer), and user-controlled options (e.g., Battery Saver and Do Not Disturb). Some resources can be directly managed by means of system-level settings, for example, the Android Quick Settings[1], allowing users to customize many system or application behaviors (Lu et al., 2019). However, applications can present unexpected behaviors because the resource interactions can introduce failures that manifest themselves in specific resource combinations (Marinho et al., 2021).

Resource interaction failures have been explored in mobile applications testing (Marinho et al., 2021; Sun et al., 2021; Lu et al., 2019). These failures occur when resources influence the behavior of other resources, similarly to the feature interaction problem in configurable software systems (Apel, Kolesnikov, et al., 2013) and telecommunication systems (Bowen et al., 1989). An example of resource interaction failure occurs for Commons App when a pair of resources are disabled (Sun et al., 2021). 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 and verify which resources are most related to 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 may be imperceptible in the testing phase.

[1] support.google.com/android/answer/9083864?hl=en

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 (Apel, Batory, et al., 2013) (Ferreira et al., 2021; Cohen et al., 2007) in which all tests must be executed in several configurations. In this work, we named a input combination as a *setting*, i.e. a set of resources whose states (enabled or disabled) are previously defined. The set we investigate in this study contains 14 common resources: `Auto Rotate`, `Battery Saver`, `Bluetooth`, `Camera`, `Do Not Disturb`, `Location`, `Mobile Data`, `Wi-Fi`, `Accelerometer`, `Gyroscope`, `Light`, `Magnetometer`, `Orientation`, and `Proximity`. The total of settings (2^{14}) makes unfeasible the use of brute force testing approaches in real development environments.

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 (Apel, Batory, et al., 2013). Several sampling strategies have been proposed and investigated in the literature, such as *t-wise* (Ferreira et al., 2021), *one-disabled* (Abal et al., 2014), and *most-enabled-disabled* (Medeiros et al., 2016). 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 (Souto et al., 2017; Medeiros et al., 2016)

In this work, we evaluate five sampling strategies in the context of resource related failures of mobile applications. Since we are not aware of specific sampling testing strategies for mobile applications domain, we choose strategies commonly used for configurable systems. To achieve our goal in this study, we follow five steps. First, we select applications from Github repositories. Second, we define the target resources and generate settings for the selected sampling testing strategies: Random, One-Enabled, One-Disabled, Most-Enabled-Disabled, and Pairwise. Third, we extend test suites with an instrumentation code aiming to control (enabling or disabling) the resources. Fourth, we execute the test suites for each setting. Finally, we analyse the generated test reports to compare the effectiveness and failure-detection capability of the sampling strategies. Moreover, we identify the resource interactions that most commonly cause failures using a frequent itemset mining analysis.

Our results indicate that Random had great percentages of failing test cases, followed by One-Enabled, Most-Enabled-Disabled, and Pairwise. Concerning the effectiveness, we found that One-Enabled and One-Disabled were the most effective strategies for six and four applications, respectively. Concerning the resource interactions more likely to cause failures, we found that some resource pairs have more influence on failures. For instance, most of the applications have failures related to pairs of disabled resources.

Overall, we delivery the following main contributions:

- We provide evidence of which testing strategies are more effective on finding resource interaction failures in real mobile applications.
- We exhibit the resources interactions more likely to cause failures.

- We make our data publicly available for further investigations on a GitHub repository [2].

The remainder of this paper is organized as follows. Section 2 presents background information on resource interaction failures and sampling testing strategies. Section 3 describes the study design. Section 4 discusses the results of our empirical study. Section 5 describes the threats to validity of this work. Section 6 presents some related work. Finally, Section 7 concludes this study and shows directions for future work.

2 Background

Some definitions are important for the understanding of our work. Therefore, this section presents an overview of resource interaction failures (Section 2.1) and the sampling testing strategies evaluated in this study (Section 2.2).

2.1 Resource Interaction Failures

We use "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 (Apel, Batory, et al., 2013).

Figure 1 presents a code excerpt of `Wikimedia Commons Android app`, showing a case of a resource interaction failure (Sun et al., 2021). The Android Platform supports the positioning via GPS or network [3]. An issue describes the situation involving the crash of the application when it is opened and both, GPS and network, are disabled [4]. 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  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
```

Fig. 1: Code Excerpt from the Wikimedia Commons Android app.

[2] https://eulerhm.github.io/samplingapptest/
[3] In this work, GPS and network are related to Location, and Wi-Fi/Mobile Data
[4] https://github.com/commons-app/apps-android-commons/issues/1735

Another example of resource interaction failure happens in `Traccar Client` [5], which is an open source application available for download at Google Play Store. In summary, this application is a GPS tracker, which communicates with its own application server. Traccar Client has an internal setting called Accuracy [6], which can be set to three values: High, Medium, or Low. To achieve the Accuracy High, it requires that the GPS, Wi-Fi, Mobile data, and Bluetooth are enabled on the smartphone.

According the the issue #390, opened at the Traccar issue tracker at GitHub [7], the application stops changing location when its Accuracy is set to Medium even if the four resources are enabled. However, it works again for the other two possible values, i.e., High and Low. It is worth to mention that the referred issue was registered in 2019 and remains "Open" in 2022.

2.2 Sampling Testing Strategies

Sampling testing strategies are commonly used to test configurable software systems (Ferreira et al., 2021; Souto et al., 2017; Medeiros et al., 2016; Abal et al., 2014). Testing 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 (Souto et al., 2017). 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 (Medeiros et al., 2016).

In this work, we used 14 Android resources, resulting in a total of $16,384$ (2^{14}) settings to be tested. Using `Vocable` as an example, we would spend a test execution time of about 1,292 hours (about 54 days) to fully test this application. This time constraint makes the exhaustive testing unfeasible in practice. In addition to time, we have other challenges to deal with, such as memory consumption and traceability of failed settings, which makes exhaustive testing in a real development environment unfeasible.

The use of sampling testing has been promising (Ferreira et al., 2021). Even with a small number of settings, it is possible to find feature interaction failures using sampling testing strategies (Ferreira et al., 2021; Souto et al., 2017; Medeiros et al., 2016; Abal et al., 2014). Below, we describe five of the most common sampling testing strategies that we selected to apply in this study. In addition, for a hypothetical system with three resources, named R_1, R_2, R_3, and no constraints among them, Table 1 shows examples of settings generated by each of the selected strategy. In this table, when an exclamation mark precedes the resource name, it indicates that the resource is disabled.

[5] https://www.traccar.org

[6] In this work, we do not deal with internal settings of applications (https://developer.android.com/guide/topics/ui/settings)

[7] https://github.com/traccar/traccar-client-android/issues/390

The strategy One-Disabled (Abal et al., 2014) 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 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 (Medeiros et al., 2016). The strategy Random creates n distinct settings with all resources randomly enabled or disabled. We used the implementation of this strategy present in FeatureIDE (Thüm et al., 2014).

In a t-wise combinatorial interaction testing (CIT), 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 (Nie & Leung, 2011). Generating such configurations can be modeled as a covering array problem instance. However, this optimization is NP-hard and several heuristics have been proposed to perform t-wise sampling (Al-Hajjaji et al., 2016). In this paper, we used the Pairwise strategy, where t=2. Specifically, the algorithm available in FeatureIDE.

Table 1: Examples of settings covering the resources R_1, R_2, R_3 for each strategy

Strategy	Settings
One-Disabled	$\{!R_1, R_2, R_3\}, \{R_1, !R_2, R_3\}, \{R_1, R_2, !R_3\}$
One-Enabled	$\{R_1, !R_2, !R_3\}, \{!R_1, R_2, !R_3\}, \{!R_1, !R_2, R_3\}$
Most-Enabled-Disabled	$\{R_1, R_2, R_3\}, \{!R_1, !R_2, !R_3\}$
Random	$\{!R_1, !R_2, R_3\}, \{!R_1, R_2, R_3\}, \{R_1, !R_2, !R_3\}, \{R_1, R_2, !R_3\}, \{R_1, R_2, R_3\}$
Pairwise	$\{R_1, R_2, !R_3\}, \{R_1, !R_2, R_3\}, \{!R_1, R_2, R_3\}, \{!R_1, !R_2, !R_3\}$

3 Study Design

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

3.1 Research Questions

We propose the following research questions:

RQ1: What is the number of failures detected by each sampling testing strategy?

RQ2: Which sampling testing strategies are the most effective on finding failures in mobile applications?

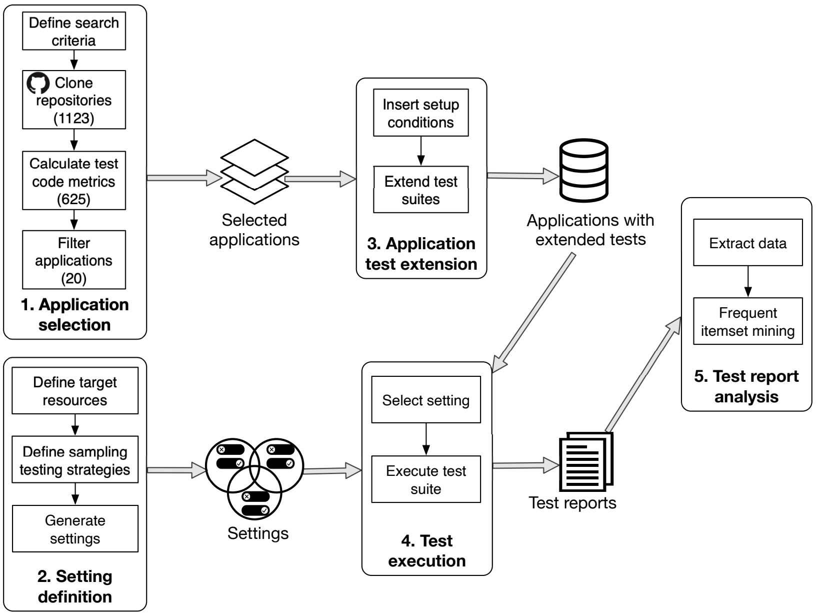


Fig. 2: The phases of the study.

RQ3: Which resource interactions are more likely to cause failures?

The first research question can be answered directly by measuring the number of failing test cases. We assume that each failing test case corresponds to a unique failure despite the used test oracle (Barr et al., 2015). For the second research question, we calculate the effectiveness according to Equation 1. *FailingSettings* is the amount of settings that cause at least one failure. *TotalSettings* represents the total of settings generated by the sampling testing strategy.

$$Effectiveness = \frac{FailingSettings}{TotalSettings} \tag{1}$$

For the third research question, we apply the frequent item set mining analysis aiming to identify the group of resources more likely to cause failures.

3.2 Study Phases

Figure 2 shows an overview of the phases of the study. First, we select the applications (Phase 1) and define the settings according to the sampling testing strategies (Phase 2). We extend the test suites to enable or disable the resources (Phase 3). We then execute the extended test suites (Phase 4) for each setting indicated by the testing strategy. Finally, we perform the analysis of the recorded test reports (Phase 5). In the following sections, we detail each phase.

Table 2: Resource related tags

Resource	Tag	Value
Bluetooth	uses-feature	android.hardware.bluetooth
Bluetooth	uses-permission	bluetooth
Location	uses-permission	access_fine_location
Location	uses-feature	android.hardware.location
Wi-Fi, Mobile data	uses-permission	internet
Camera	uses-feature	android.hardware.camera
Others	uses-feature	android.hardware.sensor.*

3.3 Application Selection

We searched for Android applications from public GitHub repositories. We defined as the search criteria repositories with at least 100 stars, with Java or Kotlin programming languages, and with the last commit on or after January 1, 2017. Java and Kotlin are the main languages for developing Android applications (Mateus & Martinez, 2019). The number of stars is a metric related to the popularity of the repository (Coelho et al., 2020) often used by researchers to select GitHub projects for empirical studies in software engineering (Borges & Valente, 2018). The data of the last commit was used to avoid unmaintained applications. The number of repositories and applications is inside the steps of the Phase 1. We conducted an initial search in order to validated the chosen thresholds. Initially, we selected a small set with 30 repositories and we perceived that in this set there were projects of interest for our study.

We cloned the initial 1,000 repositories. However, one of these repositories contains a curated list of Kotlin applications, resulting in 123 additional repositories. From the initial set, we get 625 projects with Android instrumented tests. For these projects, we used the `cloc` tool[8] to calculate the test code size. In order to manage the complexity of test executions and to not deal with very simple test suites, we constrained the projects with test code size between 400 and 4,000 LOC. This initial filtering resulted in 83 applications. We applied a last filtering to find applications with at least one of the tags described in Table 2. These tags are declared in the Manifest file and indicate some resources used by the application. A `uses-permission` tag [9] is used to request permission of the user aiming its correct operation. A `uses-feature` tag [10] is used by online stores to filter the application from devices that do not meet the hardware requirements. Hence, developers are able to control the devices in which the application may be installed. We highlight that the applications can use other resources besides those presented, since other resources could be identified from the static analysis of the application code.

Finally, the last application set contains 74 applications. From this set, we randomly selected 20 applications that did not present building or test

[8]https://github.com/AlDanial/cloc
[9]https://developer.android.com/guide/topics/manifest/uses-permission-element
[10]https://developer.android.com/guide/topics/manifest/uses-feature-element

execution issues. Table 3 presents an overview of the applications used in our study. We selected applications from different categories named according to the Play Store categories[11], with a large variation of size and test code size. For instance, applications vary from 455 lines of code (MoonShot) to more than 347,000 lines of code (WordPress-Android). Similarly, test code size vary between 464 LOC (MoonShot) and 3,732 LOC (Mixin-Messenger). Moreover, the number of test cases vary from 4 (Ground) to 164 (AnkiDroid). The number of commits, a possible metric associated to the level of maintenance activity of the repository (Coelho et al., 2020), vary between 21 (Threema) to 68,148 (WordPress-Android).

3.4 Setting Definition

For defining the settings, we established the target resources. For the selection of resources, we were oriented by two factors. First, we had to select resources often used by Android applications, such as Wi-Fi, Mobile Data, and Location and present in most of the devices. For example, the step counter motion sensor and some ambient sensors, e.g., humidity and pressure are not so common. Second, we deal with resources that can be controlled by our instrumentation. Therefore, our work includes the 14 resources described in Table 4. Some resources are directly manageable by mobile device users (for instance, Location and Mobile Data) whereas others are restricted only to more advanced users (for example, sensors, such as Accelerometer and Gyroscope), as we describe in Section 3.5. 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).

We can note that some target resources are not present in the column "Resources" of Table 3. This column presents resources declared in the Manifest file that we 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` and `Gyroscope`). In this case, other approaches for code analysis could be used for identifying additional resources.

We define setting as a 14-tuple of pairs `(resource, state)` where `state` can be `True` or `False` depending on whether the `resource` is enabled or disabled. Figure 3 presents an example of a setting where an exclamation mark preceding the resource name indicates it is disabled. For instance, only Wi-Fi and Location are disabled in this setting.

We use five sampling test strategies: Random, One-Disabled, One-Enabled, Most-Enabled-Disabled, and Pairwise. The number of settings generated by Random was limited to 30 due to the experimental time constraints. One-Disabled

[11] https://support.google.com/googleplay/android-developer/answer/9859673?hl=en

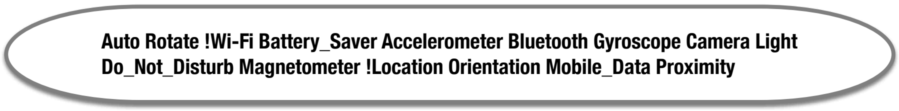
Table 3: Characteristics of the selected applications

Application	Category	LOC	Test LOC	Test cases	Resources	Commits	Exec Time
AnkiDroid[i]	Education	158,607	2,770	164	Camera, Mobile Data, Wi-Fi	13,643	1h21m39s
CovidNow[ii]	Medical	2,269	540	21	Mobile Data, Wi-Fi	85	12m11s
Ground[iii]	Productivity	19,906	525	4	Camera, Location, Mobile Data, Wi-Fi	4,936	4m31s
Iosched[iv]	Books, Reference	27,824	473	9	Location, Mobile Data, Wi-Fi	3,101	7m4s
Lockwise[v]	Productivity	14,535	1,184	38	Mobile Data, Wi-Fi	503	19m13s
Mixin-Messenger[vi]	Finance	168,080	3,732	160	Bluetooth, Camera, Location, Mobile Data, Wi-Fi	8,086	1h31m1s
Moonshot[vii]	Tools	455	464	28	Mobile Data, Wi-Fi	351	11m40s
Nekome[viii]	Productivity	1,084	2,097	64	Mobile Data, Wi-Fi	2,742	38m50s
Nl-covid19[ix]	Medical	65,839	1,114	20	Bluetooth, Mobile Data, Wi-Fi	1,293	14m34s
OpenScale[x]	Health, Fitness	27,781	1,451	14	Bluetooth, Location	2,027	8m36s
OwnTracks[xi]	Travel, Local	14,499	889	27	Location, Mobile Data, Wi-Fi	1,995	25m36s
PocketHub[xii]	Productivity	29,001	1,663	107	Mobile Data, Wi-Fi	3,512	43m50s
Radio-Droid[xiii]	Music, Audio	22,815	1,735	23	Bluetooth, Mobile Data, Wi-Fi	1,186	30m10s
Scarlet-Notes[xiv]	Productivity	4,260	770	52	Mobile Data, Wi-Fi	656	10m38s
Showly-2.0[xv]	Entertainment	2,547	952	55	Mobile Data, Wi-Fi	3,251	25m39s
SpaceX-Follower[xvi]	News, Magazines	7,664	940	30	Mobile Data, Wi-Fi	356	15m45s
Threema[xvii]	Communication	238,045	1,931	54	Bluetooth, Camera, Location, Mobile Data, Wi-Fi	21	27m49s
Vocable[xvii]	Communication	13,188	499	14	Camera	863	4m44s
Woo-Commerce[xix]	Business	156,962	1,367	27	Mobile Data, Wi-Fi	26,527	13m58s
WordPress-Android[xx]	Productivity	347,897	3,674	115	Camera, Mobile Data, Wi-Fi	68,148	1h6m5s

[i] github.com/ankidroid/Anki-Android
[ii] github.com/OMIsie11/CovidNow
[iii] github.com/google/ground-android
[iv] github.com/google/iosched
[v] github.com/mozilla-lockwise/lockwise-android
[vi] github.com/MixinNetwork/android-app
[vii] github.com/haroldadmin/MoonShot
[viii] github.com/Chesire/Nekome
[ix] github.com/minvws/nl-covid19-notification-app-android
[x] github.com/oliexdev/openScale
[xi] github.com/owntracks/android
[xii] github.com/pockethub/PocketHub
[xiii] github.com/segler-alex/RadioDroid
[xiv] github.com/BijoySingh/Scarlet-Notes
[xv] github.com/michaldrabik/showly-2.0
[xvi] github.com/OMIsie11/SpaceXFollower
[xvii] github.com/threema-ch/threema-android
[xviii] github.com/willowtreeapps/vocable-android
[xix] github.com/woocommerce/woocommerce-android
[xx] github.com/wordpress-mobile/WordPress-Android

Table 4: Descriptions of the resources

Resource	Description
Auto Rotate	An option used to allow the screen automatically rotates
Battery Saver	An option used to select power management profiles
Bluetooth	An option used to turn the access to Bluetooth connections on/off
Camera	An option used to turn the camera on/off
Do Not Disturb	An option used to control notifications, alarms, and vibration
Location	An option used to turn the location on/off
Mobile Data	An option used to turn the access to data in mobile networks on/off
Wi-Fi	An option used to turn the access to wireless networks on/off
Accelerometer	A sensor used to measure the acceleration force along the device axis
Gyroscope	A sensor used to measure the rate of the rotation around the device axis
Light	A sensor used to determine the illuminance
Magnetometer	A sensor used to determine the geomagnetic field strength along the device axis
Orientation	A sensor used to measure the device's orientation
Proximity	A sensor used to measure the distance from objects



Auto Rotate !Wi-Fi Battery_Saver Accelerometer Bluetooth Gyroscope Camera Light Do_Not_Disturb Magnetometer !Location Orientation Mobile_Data Proximity

Fig. 3: An example of setting.

and One-Enabled generated 14 settings each. Most-Enabled-Disabled generated 2 settings and Pairwise generated 8 settings. A file with the configuration set is used as input for the test execution.

3.5 Application Test Extension

We implemented a test instrumentation based on the UI Automator framework[12] to control the resources. 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[13] and the sensors are managed by Sensor Disabler[14]. This application requires a rooted Android device [15].

Figure 4 presents the icons used to control some resources used in this study. The Location icon allows the user to control the device's location. The next icons allow the user to manage Wi-Fi (e.g, ParcTE_Ext is the name of the connected network), Mobile data, and Bluetooth connections. The Battery

[12]https://developer.android.com/training/testing/ui-automator
[13]https://github.com/percula/LensCap
[14]https://github.com/wardellbagby/Sensor-Disabler
[15]https://en.wikipedia.org/wiki/Rooting_(Android)

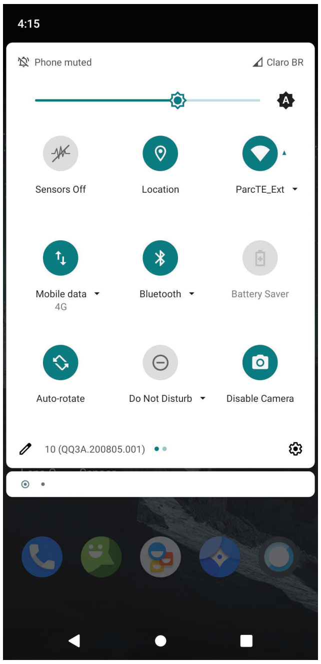


Fig. 4: Android Quick Settings.

Saver icon allows the user to select power profiles related to battery consumption. The Auto-rotate icon allows the user to turn the screen rotation on or off. The Do Not Disturb icon allows the user to limit interruptions caused by sound, vibration, and notifications. The Disable/Enable Camera icon, implemented by Lens Cap, allows the control of the device camera.

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 [16]. 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 [17] for the `Location` and TelephonyManager [18] 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

[16] https://junit.org/junit4/javadoc/4.12/org/junit/BeforeClass.html
[17] https://developer.android.com/reference/android/location/LocationManager
[18] https://developer.android.com/reference/android/telephony/TelephonyManager

Orchestrator [19], a tool that helps minimize possible shared states, a known factor associated to flaky tests (Parry et al., 2021) and isolate the crashes.

Algorithm 1 Resource_setup

```
1:  Input
2:      S    list of $< resource, state >$ pairs
3:  procedure AdjustResourceStates($S$)
4:      for all $pair \in S$ 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
```

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:  $maxExecutions \leftarrow 3$
7:  for $exec \leftarrow 1$ to $maxExecutions$ do
8:      SHUFFLE($SL$)
9:      for all $st \in SL$ do
10:         AdjustResourceStates($st$)
11:         Execute the whole test suite of $AP$
12:     end for
13: end for
```

3.6 Test Execution

We implemented Algorithm 2 for managing the test executions. 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. As noted by Parry et al. (2021), there does not seem to be a clear, optimal number of re-executions to identify flaky tests. The study of Lam et al. (2020) suggests a maximum of five re-executions.

[19] https://developer.android.com/training/testing/junit-runner

Based on previous essays, we set the number of re-executions to three. We took to account our time constraints for the experiment 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. We used the following devices: Samsung Galaxy M30 with 4 GB RAM, Samsung Galaxy S10 with 8 GB RAM, and Xiaomi Pocophone F1 with 6 GB RAM. All devices run Android 10. For each device, we allocated a group of applications.

For illustrating the experimental effort, we synthesize the average execution time of the test suites of each application in "Average Exec Time" column of Table 3. The average CPU time was calculated considering three executions of all testing strategies. We can see that the execution time varies between 4m 31s (Ground) and 1h 31m 1s (Mixin-Messenger). Considering the number of settings of all strategies (68) and the total number of executions (3) the total execution time vary between 15h 30m 24s and 12d 21h 27m 24s. From our observations, we verify an expressive time overhead due to the use of the Android Test Orchestrator mentioned in Section 3.5.

3.7 Test Report Analysis

We analysed test reports for identifying the failed test cases and registering the related settings. For the same setting, multiple test cases can fail. For each Android application, we use an implementation of the Apriori algorithm (Agrawal & Srikant, 1994) to perform a frequent itemset mining analysis on the occurrences of all settings that led a test case to fail. A frequent itemset is defined as a set of items that occur together in at least a `Support` threshold value of all transactions available. `Support` of an itemset is defined as the proportion of transactions in the data set which contain the itemset (Hornik et al., 2005). In this work, each item is a resource state i.e., enabled or disabled. We empirically determine the `Support` as 0.1 (10%) (Marinho et al., 2021). Values of the `Support` less than 10% caused the data resulting from the analysis of the whole set of applications to increase from around 17 thousand records to hundreds of thousands records [20]. Moreover, the lower value for the `Support` allowed us to do a more in-depth analysis, as can be seen in Section 4.3.

4 Results and Discussion

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

4.1 Failures Detected by Testing Strategies (RQ1)

A failed test case corresponds to a test case which originally terminated with success and which, subjected to a change in the resource setting, fails. Therefore, each failing test case corresponds to a failure despite the type (e.g., crash,

[20]A record include a set of resource states and the value of the `Support` achieved

assertion violation, etc) and the number of failures reported. Table 5 presents the number of failing test cases for each testing strategy and the percentage of failing test cases related to the total number of test cases (#TC). We used the symbol "-" to indicate that the strategy was not able to detect failures. Note that we executed the test suites three times for each application setting. We report failures manifested in all three executions (see Section 3.6). Some applications had only failures in one or two executions, such as Iosched and RadioDroid. For each application, we highlight the strategies that achieved the best results with respect to the percentage of failing test cases.

Table 5: Failing test cases for each testing strategy

Application	#TC	Random	One-Dis	One-Enab	Most-Enab-Dis	Pairwise
CovidNow	21	2(9%)	1(5%)	1(5%)	2(9%)	-
Lockwise	38	4(10%)	4(10%)	4(10%)	4(10%)	4(10%)
Mixin-Messenger	160	2(1%)	-	2(1%)	2(1%)	2(1%)
Nl-covid19	20	6(30%)	6(30%)	6(30%)	6(30%)	6(30%)
OwnTracks	27	3(11%)	3(11%)	3(11%)	3(11%)	3(11%)
PocketHub	107	3(3%)	1(1%)	-	-	-
SpaceXFollower	30	4(13%)	2(6%)	4(13%)	1(3%)	4(13%)
Threema	54	1(2%)	1(2%)	1(2%)	1(2%)	1(2%)
Vocable	14	7(50%)	7(50%)	7(50%)	7(50%)	7(50%)
WordPress-Android	115	11(9%)	2(1%)	6(5%)	10(8%)	11(9%)

The percentage of failing test cases varied between 1% (e.g., One-Disabled with PocketHub) and 50% (e.g., all strategies with Vocable). Note that some applications have a low number of test cases, such as Vocable (14) and Nl-covid19 (20). Random had high percentages of failing test cases (10 applications), followed by Pairwise (8 applications). Overall, we can see that these failures are not common since the percentages for most of the applications (with the exception of Nl-covid19 and Vocable) are lower than 13%. A remarkable case occurred to PocketHub, for which only Random and One-Disabled were able to detect failures with percentages of failing test cases of 3% and 1%, respectively.

4.2 The Most Effective Testing Strategies (RQ2)

Table 6 presents the number of settings that cause failures (FS) and Effectiveness (E) calculated using Equation 1 of Section 3.1. Similarly to RQ1, we consider only settings that cause failures in three executions. We used the symbol "-" to indicate that none of the strategy settings were able to detect failures. We highlight in Table 6 the most effective strategies for each application.

The most effective strategies were One-Enabled (8 applications) and One-Disabled (4 applications). The effectiveness of One-Disabled and One-Enabled varied widely from 0.07 to 1.00. We can also observe that the effectiveness of Most-Enabled-Disabled and Pairwise oscillated between 0.25 and 1.00 despite

Table 6: Effectiveness of testing strategies

	Random		One-Disabled		One-Enabled		Most-Enab-Dis		Pairwise	
Application	FS	E	FS	E	FS	E	FS	E	FS	E
CovidNow	17	0.57	1	0.07	13	0.93	1	0.50	-	-
Lockwise	30	1.00	14	1.00	14	1.00	2	1.00	8	1.00
Mixin-Messenger	5	0.17	-	-	12	0.86	1	0.50	2	0.25
Nl-covid19	26	0.87	8	0.57	14	1.00	1	0.50	6	0.75
OwnTracks	14	0.47	14	1.00	14	1.00	2	1.00	8	1.00
PocketHub	3	0.10	1	0.07	-	-	-	-	-	-
SpaceXFollower	30	1.00	14	1.00	14	1.00	2	1.00	8	1.00
Threema	14	0.47	13	0.93	1	0.07	1	0.50	4	0.50
Vocable	5	0.17	1	0.07	13	0.93	1	0.50	4	0.50
WordPress-Android	18	0.60	2	0.14	12	0.86	1	0.50	4	0.50

the low number of settings of each strategy (2 and 8, respectively). These two strategies can be used in a test environment in which a trade-off between execution time and effectiveness is required.

4.3 Resource Interactions Most Likely to Cause Failures (RQ3)

Aiming to understand the resource interactions most likely to cause failures, we perform a frequent item set mining analysis (Agrawal et al., 1993). We consider the settings that cause the test cases to fail. From this analysis, we can observe which resource pairs have more impact on failures, since the support of pairs is greater compared to other resource combinations. Table 7 shows the three most common pairs and the support for each application, considering the whole set of all testing strategies. We use an exclamation mark to indicate when the resource is disabled. The pairs are often distinct among the applications. Only the pair $\langle !Mobile\_Data, !Wi\text{-}Fi\rangle$ is the most common for four applications (CovidNow, Mixin-Messenger, SpaceXFollower, and WordPress-Android). In general, the pairs include at least one of the resources identified from the application Manifest file as presented in Section 3.3.

Figures 5, 6, and 7 show heat maps with the supports of resource pairs for each application. Figure 5 presents the pairs for the whole set of applications exhibiting the resource names along the axes. We can observe in Figure 5 that failures are frequently caused to disabled resources, such as `!Mobile_Data`, and `!Wi-Fi`. Figures 6 and 7 present an overview of the pairs for the remaining applications. Eight applications (CovidNow, Lockwise, Mixin-Messenger, Nl-covid19, OwnTracks, SpaceXFollower, Vocable, and WordPress-Android) have most failures related to pairs of disabled resources. Four applications (Lockwise, OwnTracks, PocketHub, and Threema) have most failures related to pairs of enabled resources. For PocketHub and Threema, `Battery_Saver` and `Do_Not_-Disturb` are the most common pairs of resources causing failures since these resources are restrictive when they are enabled. For instance, `Battery Saver`,

Table 7: Most common resource pairs per application

Application	Resource Pair	Support
CovidNow	$\langle !Mobile\_Data, !Wi\text{-}Fi \rangle$	0.86
CovidNow	$\langle !Light, !Wi\text{-}Fi, \rangle$	0.80
CovidNow	$\langle !Light, !Mobile\_Data \rangle$	0.77
Lockwise	$\langle Accelerometer, Bluetooth \rangle$	0.41
Lockwise	$\langle Accelerometer, Light \rangle$	0.41
Lockwise	$\langle Accelerometer, Proximity \rangle$	0.41
Mixin-Messenger	$\langle !Mobile\_Data, !Wi\text{-}Fi \rangle$	1.00
Mixin-Messenger	$\langle !Gyroscope, !Wi\text{-}Fi \rangle$	0.85
Mixin-Messenger	$\langle !Gyroscope, !Mobile\_Data \rangle$	0.85
Nl-covid19	$\langle !Accelerometer, !Bluetooth \rangle$	0.66
Nl-covid19	$\langle !Accelerometer, !Mobile\_Data \rangle$	0.61
Nl-covid19	$\langle !Accelerometer, !Wi\text{-}Fi \rangle$	0.60
OwnTracks	$\langle !Light, !Location \rangle$	0.47
OwnTracks	$\langle !Battery\_Saver, !Location \rangle$	0.47
OwnTracks	$\langle !Camera, !Location \rangle$	0.47
PocketHub	$\langle Gyroscope, Wi\text{-}Fi \rangle$	0.75
PocketHub	$\langle Light, Wi\text{-}Fi \rangle$	0.75
PocketHub	$\langle Battery\_Saver, Mobile\_Data \rangle$	0.75
SpaceXFollower	$\langle !Accelerometer, !Wi\text{-}Fi \rangle$	0.58
SpaceXFollower	$\langle !Mobile\_Data, !Wi\text{-}Fi \rangle$	0.53
SpaceXFollower	$\langle !Orientation, !Wi\text{-}Fi \rangle$	0.53
Threema	$\langle Do\_Not\_Disturb, Magnetic\_Field \rangle$	0.78
Threema	$\langle Bluetooth, Do\_Not\_Disturb \rangle$	0.73
Threema	$\langle Auto\_Rotate, Do\_Not\_Disturb \rangle$	0.73
Vocable	$\langle !Camera, !Magnetic\_Field \rangle$	0.83
Vocable	$\langle !Camera, !Proximity \rangle$	0.80
Vocable	$\langle !Battery\_Saver, !Camera \rangle$	0.75
WordPress-Android	$\langle !Mobile\_Data, !Wi\text{-}Fi \rangle$	0.92
WordPress-Android	$\langle !Bluetooth, !Wi\text{-}Fi \rangle$	0.81
WordPress-Android	$\langle !Gyroscope, !Wi\text{-}Fi \rangle$	0.81

when enabled, limits the executions of background functionalities [21], such as location updates.

4.4 Implications

The results of our study provided practical implications for testers, practitioners, researchers, and tool builders.

Implications for Testers. Our results indicate that testers must be aware of the resource interaction failures when implementing test suites. Moreover, we suggest that the adoption of the testing strategies be guided by considerations of the execution times and the effectiveness of the generated settings. In this way, the strategies with a low number of settings, such as Most-Enabled-Disabled and Pairwise, can be used in test environments where there are tight execution time constraints, despite they are not the most effective strategies.

[21] https://developer.android.com/about/versions/pie/power

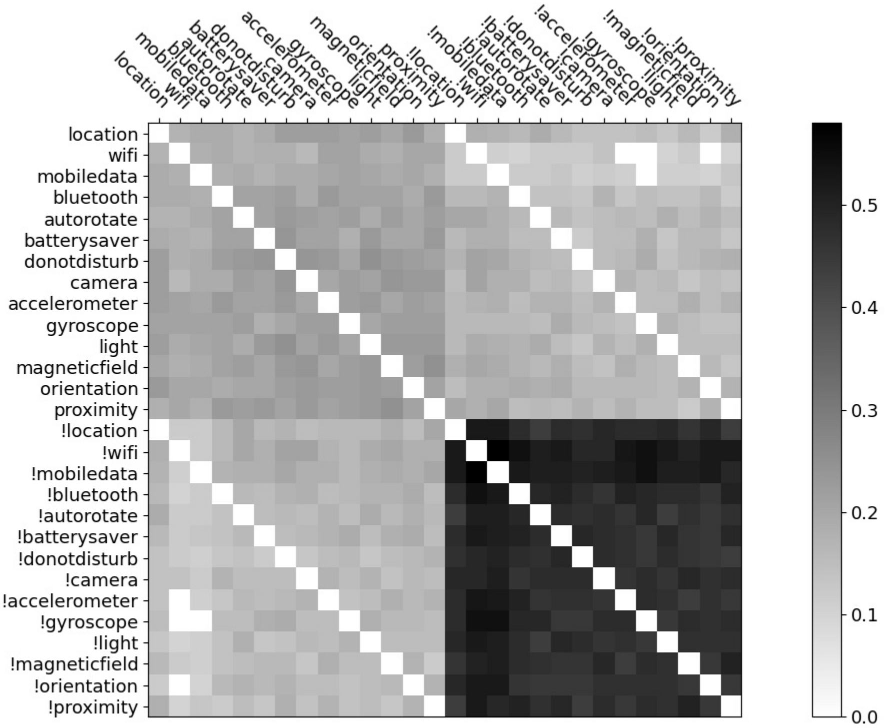
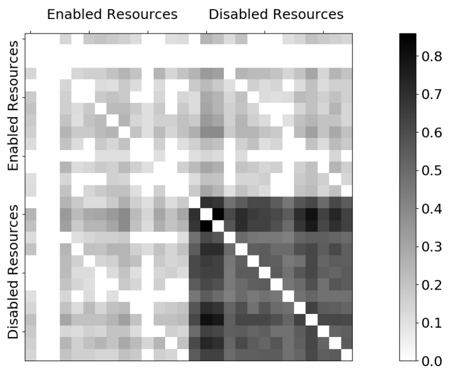


Fig. 5: Two-resources interactions - All applications.

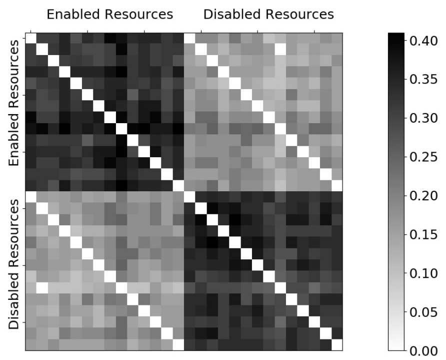
Implications for Developers. Our results suggest that developers can benefit from our findings, particularly, when designing and implementing their applications. They can look for resource interaction pairs that have more impact on failures, starting from those declared in the Manifest file. These pairs can also be evaluated in testing activities closer to the development, such as unit and integration testing. Therefore, we can see the direction to decrease the overhead of our instrumentation code, for example, managing the resources only using APIs [22] to avoid the need for GUI interactions often not common during unit and integration testing.

Implications for Researchers. Researchers have the opportunity to explore other dimensions of resource interaction failures. First, they can explore what code elements are more prone to this kind of failure, for example, using object-oriented metrics and looking for correlations between classes and the faults behind the failures (Ferreira et al., 2021). Other opportunity is the exploration of testing strategies focused on resources explicitly used by the application. We see the potential use of code analysis techniques aiming to identify resources directly used by the application. However, there must be a carefully approach to deal with resources used in a indirect way (for

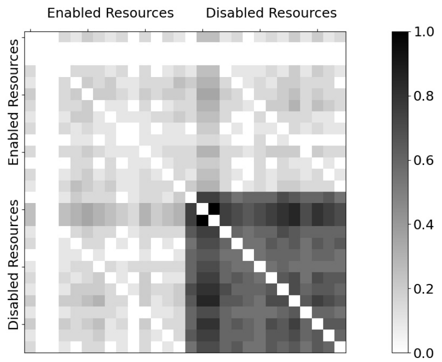
[22] Two Android APIs are mentioned in Section 3.5



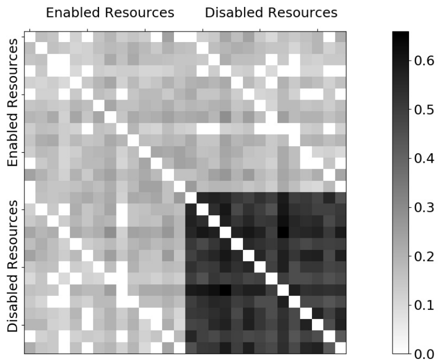
(a) CovidNow



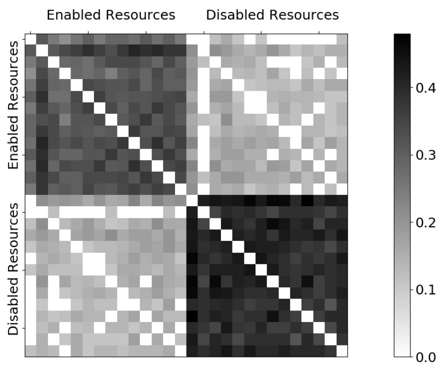
(b) Lockwise



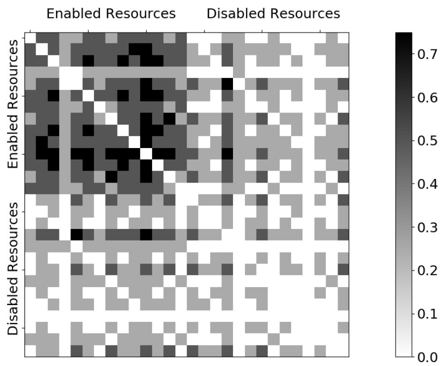
(c) Mixin-Messenger



(d) Nl-covid19

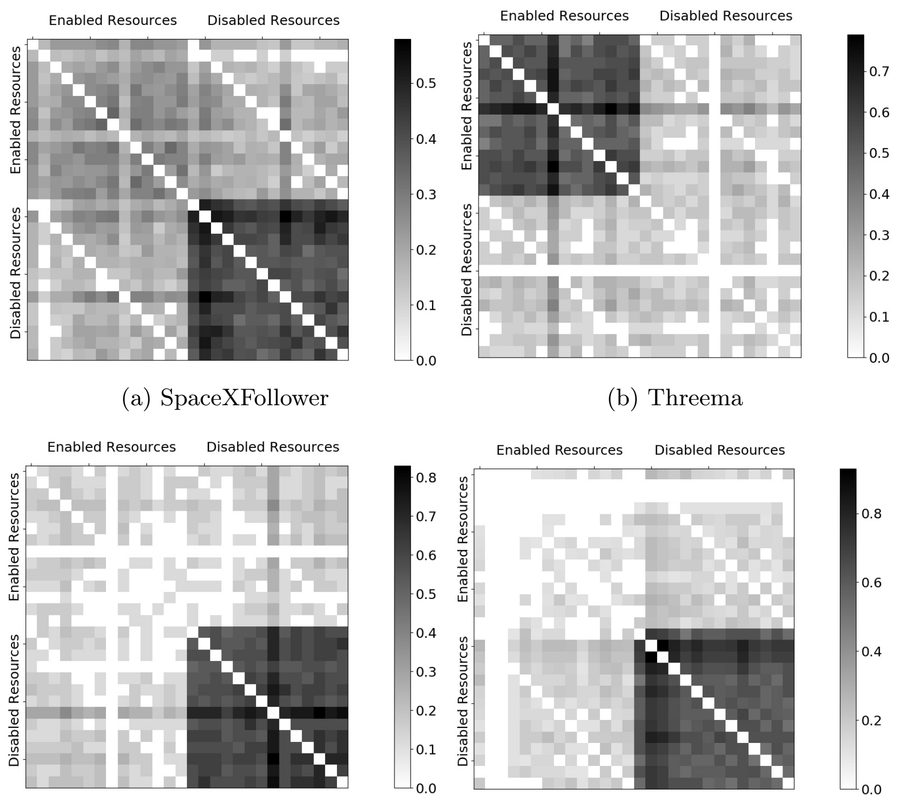


(e) OwnTracks



(f) Pockethub

Fig. 6: Two-resources interactions - Part 1.



(a) SpaceXFollower (b) Threema

(c) Vocable (d) WordPress-Android

Fig. 7: Two-resources interactions - Part 2.

instance, an application using the Camera can also require the Location for image tagging).

Implications for Tool Builders. Tool builders can automate improved techniques so that they can be used by practitioners and testers. As some studies show (Pecorelli et al., 2022; Luo et al., 2020; Silva et al., 2018), there are limitations in IDEs, emulators, and frameworks to simulate events that are different from those of the GUI. These events are sometimes referred to as system events (Cai & Ryder, 2020) or context events (Luo et al., 2020). The design principles of our instrumentation can be used to support handling of contextual events in mobile application testing and extending existing test suites.

5 Threats to Validity

One of the main concerns when performing case studies like ours is the validity of results and their applicability to other contexts. Although we carefully designed and conducted our study, some threats to validity may have harmed our study results and discussions. We discuss some threats to the study validity based on four well-known categories of validity threats (Wohlin et al., 2012): construct, internal, external, and conclusion.

Construct Validity. The construct validity reflects what extent the operational measures that are studied represent what the researchers have in mind and what is investigated according to the research questions (Wohlin et al., 2012). In our study, the first threat to construct validity is related to how we extend test suites and control the resource states (enabled/disabled). To mitigate this threat, all co-authors of this study carefully inspected the implemented code. Another threat to construct validity concerns the choice of the analyzed applications and metrics. We opt to select twenty different projects to favor external validity as discussed below. However, these metrics may not properly capture how the testing strategies perform in mobile applications. To mitigate this threat, we rely on well-known metrics to quantify failures and effectiveness of the testing strategies. We also discussed among the authors aiming to select the best metrics available according to our research questions.

Internal Validity. The internal validity is related to uncontrolled factors that may affect the study results (Wohlin et al., 2012). The execution of the study steps is a threat to the internal validity, since a poor execution may result in the collection of incorrect data. For instance, we discarded from our study small Android applications with less than 400 lines of test code. Some uncontrolled factors could also emerge from our instrumentation of test cases. To minimize threats due to our instrumentation, we set the resource status manually and ran the tests that identified faults in each faulty configuration. Finally, we restricted our study to run each test suite three times for each application. This time constraint may affect our findings since flaky tests can occur in other executions. Lam et al. (2020) suggests that five reruns are able to detect 88% of flaky tests (three reruns detect more than 75%). According to our evaluations, we observed three separate runs were sufficient to find the flaky tests which oscillated between 20% (Mixin-Messenger) and 75% (PocketHub) of total failed tests observed for each application.

Our solution may not be very robust to different Android versions, since we use rooted devices in order to manage the individual sensors by means of the Sensor Disabler application that we were able to set up in devices running Android 10. One of the reasons of our option for physical devices is the constraints present in emulators, such as the lack of native support for bluetooth connections, control (enable/disable) of individual sensors, and the test execution performance. Sensor Disabler requires the installation of a module of the Xposed framework [23] that only executes in rooted devices. This module was tested and confirmed to work on devices with Android 5 to Android 10 [24]. We believe that improvements of the testing support of emulators would help us to scale our experimental methodology.

[23]https://github.com/ElderDrivers/EdXposed
[24]https://github.com/wardellbagby/sensor-disabler

External Validity. The external validity concerns the ability to generalize the results to other environments, such as to industry applications and systems (Wohlin et al., 2012). Regarding this validity threat, we have performed our study with twenty Android applications and five sampling strategies. We also restricted our analysis to mobile applications developed in the Java and Kotlin programming languages. Therefore, we cannot generalize to other programming languages and frameworks, such as Flutter and React Native. In fact, the selected applications may not represent the characteristics of all Android applications. However, we selected popular applications with more than 100 stars from various domains, longevity, and sizes. With respect to the testing strategies, we focus on some of the most investigated strategies based on several research papers (Ferreira et al., 2021; Souto et al., 2017; Medeiros et al., 2016).

Conclusion Validity. The conclusion validity concerns with issues that affect the ability to draw the correct conclusions from the study (Wohlin et al., 2012). These results reflect our perceptions and interpretations of the metrics collected from the applications after running the testing strategies. All authors participated in the data analysis process and discussions on the main findings, to mitigate the bias of relying on the interpretations of a single person. Nonetheless, there may be other important aspects in the data collected, not yet discovered or reported by us. For instance, for answering the research questions, we assumed that the identified failures are related to interactions of the 14 managed resources. Nevertheless, the failures can be caused by other resources not considered in this study that may be interacting with some of the managed resources. However, we believe that similar conclusions would also be achieved using different metrics and tools that quantify similar attributes for the same mobile applications.

6 Related Work

Combinatorial Testing Approaches for Highly-Configurable Systems. Several works have investigated different strategies to deal with the combinatorial explosion problem in the context of highly-configurable systems. For instance, Ferreira et al. (2021) compared Random, 1-, 2-, 3-, and 4-wise sampling testing strategies, generated by four distinct algorithms, in terms of efficiency and effectiveness. In our study, we used Random and Pairwise. Medeiros et al. (2016) compared the selected sample sizes and the fault-detection capabilities of 10 current state-of-the-art sampling algorithms. We evaluate a subset of these algorithms in the context of mobile applications, namely Pairwise, Random, One-Enabled, One-Disabled, and Most-Enabled-Disabled. Souto et al. (2017) used One-Enabled, One-Disabled, Most-Enabled-Disabled, and Pairwise sampling techniques to combine them with a previous sound technique (SPLat) in order to validate their proposed soundness algorithm, S-SPLat. However, they did not provide a comparison among sampling techniques as we do in this work.

Secondary Studies on Mobile Applications Testing. We perceived mobile applications testing has been an active research field, as evidenced by several secondary studies (Júnior et al., 2022; Villanes, Endo, & Dias-Neto, 2022; Luo et al., 2020; Tramontana et al., 2019). Júnior et al. (2022) present a systematic mapping of testing techniques and tools for non-functional requirements, identifying quality characteristics addressed by the primary studies. Among these, the compatibility and portability are related to the proper operation of applications considering the huge diversity of devices and platforms (Villanes et al., 2022). In this study, we also evaluate the variety of resource settings that is to some degree related to compatibility testing.

Luo et al. (2020) give an overview of context simulation methods for testing mobile context-aware applications and highlight the challenges for tackling the generation of dynamic input data commonly originating from sensors. In this study, we also deal with the complexity of input data by enabling/disabling resources.

Tramontana et al. (2019) use a systematic mapping to analyze studies involving the automation of functional testing of mobile applications. They claim that there is some scarcity of studies considering the context-aware aspects for functional testing, such as the sources of contextual events (e.g., sensors, Internet connections, etc). Our study targets the expansion of test suites of instrumented tests, a kind of functional testing, with specific contextual information of the resource states.

Combinatorial Testing Approaches for Mobile Applications. In the context of testing Android applications, we found works that study the combinatorial explosion of testing with distinct targets, like combining application's preferences, system settings, device and operating system characteristics. For instance, Wei et al. (2018) focus on understanding the fragmentation-induced compatibility (FIC) issues in Android applications. Such issues occur due to fast-evolving Android platforms (many versions and API levels) and numerous device models. Different from our study and the others mentioned above, they inspect source code and FIC-related problems and fixes and propose a tool (FicFinder). However, they do not perform any kind of combinatorial testing.

Lu et al. (2019) work was motivated by the fact that each application can provide its specific set of preferences to the end user. For instance, the application's preferences screen may present distinct types of widgets (checkboxes, lists, etc.) to set the option values that change the app's behaviour. The preference set sizes of the 30 subject systems in their study vary between 5 and 96 (27 in average). Therefore, testing all combinations of preference options is also impracticable. To tackle this problem, they proposed Prefest, an automatic preference-wise testing approach that uses static and dynamic combined analysis to locate the preferences that may affect the test cases and execute them only under necessary option combinations. Instead of proposing a technique, we brought to the context of Android testing five well established sampling strategies that suggest, statically, combinations of settings to test.

Vilkomir (2018) also tackled a combinatorial testing problem. Their study focused on device-specific faults, the multi-device testing coverage problem, since it is impracticable to test all combinations of devices, resolutions, resolution types, Android OS versions, screen sizes and RAM capacity. The goal was determining how many devices must be tested and which methods for device selection are best for revealing device-specific faults. For instance, *OS coverage* and *each-choice (1-wise)* were recommended as reasonable and practical approaches that are highly effective for fault detection with testing on a relatively small number of mobile devices. In this study, although using three distinct smartphone devices, we focused on the sampling combination strategies of the 14 resources available in each smartphone running Android 10. One of them, pair-wise, generates sets that embrace each-choice sets.

Sun et al. (2021) addressed a problem similar to the one we investigate, named *setting-related defects* (or "system defects"). They focuses on user configurable system settings with more than 50 options, which we call *resources*. However, their work is limited to defects caused by settings changes (enabling or disabling) during the usage of applications and argued that "generic state-of-the-art generic application testing techniques are limited to detecting crash failures due to the lack of strong test oracles, while many setting defects are logical ones that lead to application freezing, functionality failures, or GUI display failures". While they mined defects by using bug reports based on the option keywords and investigated them, we used sampling and combinatorial strategies to find failures and, mainly, to compare such strategies. They proposed *setting-wise metamorphic fuzzing*, the first automated testing approach to effectively detect setting defects without explicit oracles while we investigate the resource interaction more likely to cause failures (Section 4.3). They noted that only about 2% of setting defects are caused by the mutual influence between two settings, what we call *resource interaction failures* in Section 2.1. However, we found a more considerable amount of this kind of failures, as can be seen in Section 4.1.

Table 8 provides a brief comparison between our study and 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. As we can see, in our study we use established sampling testing strategies to deal with combinatorial aspects of the settings being tested. We use a dash '-' to indicate information not available in the respective study.

Previous Study. In our previous study (Marinho et al., 2021), we search for sensor-interaction failures by testing all combinations of 8 resources in Android applications. The resources Accelerometer, Barometer, Camera, Gyroscope, Magnetometer, Microphone, and Proximity were handled as a unique sensor (we named "usual sensors") in Android Quick Settings. In this follow-up study, we expanded our analysis to 14 resources by (i) handling five out of the six "usual sensors" as separated resources (Barometer is out); (ii) using all other seven sensors the same way as in the previous study; and (iii) including two new resources: Light and Orientation. Due to increased

Table 8: Comparison Summary

	This study	Wei et al. (2018)	Lu et al. (2019)	Vilkomir (2018)	Sun et al. (2021)
Operating System	Android	Android	Android	Android	Android
Applications	20	53	30	15	31
Strategies-Techniques	Sampling	FicFinder	Pairwise, PrefTest	Each-Choice	Setting-wise metamorphic fuzzing
Comparison Elements (Resources, hardware options)	14	-	4	5	> 50
Applications with failures	10	-	5	15	29

number of resources, it was impractical to run all possible settings (2^{14}) and, therefore, we choose 5 sampling testing strategies to suggest sets of settings with which we were able to look for resource-interaction failures.

7 Conclusion and Future Work

In this work, we evaluated sampling testing strategies in the context of resource related failures of mobile applications. Our study involves 14 typical mobile resources. The dataset includes 20 applications with different characteristics (category, Test LOC, Test cases etc). We identified the number of failures from the amount of failing test cases detected by each strategy, the most effective strategies, and the resource interactions more likely to cause failures. Our approach relied on multiple executions for minimizing the effects of flaky tests. Therefore, from the initial set of application, our analysis was focused on eight applications that had failures in three executions.

As a result, Random had great percentages of failing test cases, followed by Pairwise. We found that resource related failures are not so common, such as for pocketHub. For this application, Random and One-Disabled had 3% and 1% of failing test cases, respectively. Concerning the effectiveness, we found that One-Enabled and One-Disabled were the most effective strategies for eight and four applications, respectively. On the other hand, Most-Enabled-Disabled and Pairwise can be used in development contexts where there must be a balance between test execution time and effectiveness. Concerning the particularities of resource interactions more likely to cause failures, we found that resource pairs have more influence on failures. Most of the applications have failures related to pairs of disabled resources. The exceptions are related mainly to `Battery_Saver` and `Do_Not_Disturb` that are restrictive when they are enabled. Surprisingly, the pairs varied widely among the applications and include resources identified from the Manifest file.

Our results can be used by developers and testers as a decision making support for considering resource interactions when implementing their test suites. Moreover, they can benefit in use sampling strategies to improve or propose testing approaches. Researchers can benefit from our analysis and replicate our study for other mobile applications technologies besides Android.

As future work, we suggest further studies to include other resources and the investigation of cloud testing (Bertolino et al., 2019) aiming to decrease the test execution effort. Another direction is the exploration of other sampling strategies such as statement-coverage and other t-wise strategies. Moreover, the faults behind the resource interaction failures can be used to determine the classes more prone to this kind of failure. Therefore, we can investigate whether these classes differ from other classes, e.g., by determining traditional source code metrics and CK metrics.

Author Contributions

Euler Horta Marinho, Fischer Ferreira, and João P. Diniz were responsible for the data collection. All authors were involved with data analysis as well as writing and reviewing the manuscript.

Funding Declaration

This work was partially supported by Brazilian funding agencies: CAPES and CNPq.

Conflict of Interest Statement

The authors declare that there is no conflict of interest.

Data Availability Statement

The data used in this study is available in a public repository at https://eulerhm.github.io/samplingapptest

References

Abal, I., Brabrand, C., & Wasowski, A. (2014). 42 variability bugs in the Linux Kernel: A qualitative analysis. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)* (p. 421–432).

Agrawal, R., Imieliński, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (p. 207–216). New York, NY, USA: Association for Computing Machinery.

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. In *Proceedings of the International Conference on Very Large Databases (VLDB)* (Vol. 1215, pp. 487–499).

Al-Hajjaji, M., Krieter, S., Thüm, T., Lochau, M., & Saake, G. (2016). Incling: efficient product-line testing using incremental pairwise sampling. *ACM SIGPLAN Notices*, *52*(3), 144–155.

Amalfitano, D., Amatucci, N., Memon, A. M., Tramontana, P., & Fasolino,

A. R. (2017). A general framework for comparing automatic testing techniques of android mobile apps. *Journal of Systems and Software*, *125*, 322-343.

Apel, S., Batory, D., Kastner, C., & Saake, G. (2013). *Feature-oriented software product lines.* Springer Berlin / Heidelberg.

Apel, S., Kolesnikov, S., Siegmund, N., Kästner, C., & Garvin, B. (2013). Exploring feature interactions in the wild: The new feature-interaction challenge. In *proceedings of the 5th international workshop on feature-oriented software development (fosd).*

Barr, E., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). The oracle problem in software testing: a survey. *IEEE Transactions on Software Engineering*, *41*, 507–525.

Bertolino, A., Angelis, G. D., Gallego, M., García, B., Gortázar, F., Lonetti, F., & Marchetti, E. (2019). A systematic review on cloud testing. *ACM Computing Surveys*, *52*(5).

Borges, H., & Valente, M. T. (2018). What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, *146*, 112-129.

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

Cai, H., & Ryder, B. (2020). A longitudinal study of application structure and behaviors in Android. *IEEE Transactions on Software Engineering*, *47*(12), 2934–2955.

Coelho, J., Valente, M. T., Milen, L., & Silva, L. L. (2020). Is this github project maintained? Measuring the level of maintenance activity of open-source projects. *Information and Software Technology*, *122*, 106274.

Cohen, M. B., Dwyer, M. B., & Shi, J. (2007). Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (p. 129–139).

Escobar-Velásquez, C., Linares-Vásquez, M., Bavota, G., Tufano, M., Moran, K., Di Penta, M., ... Poshyvanyk, D. (2020). Enabling mutant generation for open- and closed-source Android apps. *IEEE Transactions on Software Engineering (TSE)*, *48*(1), 186-208.

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

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

Hornik, K., Grün, B., & Hahsler, M. (2005). arules-a computational environment for mining association rules and frequent item sets. *Journal of*

Statistical Software, *14*(15), 1–25.

Júnior, M. C., Amalfitano, D., Garcés, L., Fasolino, A. R., Andrade, S. A., & Delamaro, M. (2022). Dynamic testing techniques of non-functional requirements in mobile apps: A systematic mapping study. *ACM Computing Surveys (CSUR)*, *54*(10s), 1–38.

Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F., & Klein, J. (2018). Automated testing of Android apps: A systematic literature review. *IEEE Transactions on Reliability*, *68*(1), 45–66.

Lam, W., Winter, S., Astorga, A., Stodden, V., & Marinov, D. (2020). Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *ISSRE 2020: 31st IEEE International Conference on Software Reliability Engineering* (pp. 403–413).

Lu, Y., Pan, M., Zhai, J., Zhang, T., & Li, X. (2019). Preference-wise testing for Android applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (pp. 268–278).

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

Marinho, E. H., Diniz, J. P., Ferreira, F., & Figueiredo, E. (2021). Evaluating sensor interaction failures in mobile applications. In *International Conference on the Quality of Information and Communications Technology (QUATIC)* (pp. 49–63).

Mateus, B. G., & Martinez, M. (2019). An empirical study on quality of Android applications written in Kotlin language. *Empirical Software Engineering*, *24*, 3356-3393.

Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., & Apel, S. (2016). A comparison of 10 sampling algorithms for configurable systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (pp. 643–654).

Nie, C., & Leung, H. (2011). A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, *43*(2), 1–29.

Parry, O., Kapfhammer, G. M., Hilton, M., & McMinn, P. (2021). A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *31*(1).

Pecorelli, F., Catolino, G., Ferrucci, F., De Lucia, A., & Palomba, F. (2022). Software testing and Android applications: a large-scale empirical study. *Empirical Software Engineering*, *27*(2).

Silva, D. B., Eler, M. M., Durelli, V. H., & Endo, A. T. (2018). Characterizing mobile apps from a source and test code viewpoint. *Information and Software Technnology*, *101*, 32–50.

Souto, S., d'Amorim, M., & Gheyi, R. (2017). Balancing soundness and efficiency for practical testing of configurable systems. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (pp.

632–642).

Sun, J., Su, T., Li, J., Dong, Z., Pu, G., Xie, T., & Su, Z. (2021). Understanding and finding system setting-related defects in Android apps. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (pp. 204–215).

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., & Leich, T. (2014). FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, *79*, 70-85.

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

Vilkomir, S. (2018). Multi-device coverage testing of mobile applications. *Software Quality Journal (SQJ)*, *26*(2), 197–215.

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

Wei, L., Liu, Y., Cheung, S.-C., Huang, H., Lu, X., & Liu, X. (2018). Understanding and detecting fragmentation-induced compatibility issues for Android apps. *IEEE Transactions on Software Engineering (TSE)*, *46*(11), 1176–1199.

Wohlin, C., Runeson, P., Host, M., Ohlsson, M. C., Regnell, B., & Wesslen, A. (2012). *Experimentation in Software Engineering.* Springer Berlin / Heidelberg.