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

Eduardo Fernandes
University of Southern Denmark
Odense, Denmark

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

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

ABSTRACT

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;

SPLC '24, September 2–6, 2024, Luxembourg City, Luxembourg

ACM ISBN 978-1-4503-7569-6/20/10…$15.00
https://doi.org/10.1145/3382025.3414725

ACM Reference Format:
Euler Marinho, Fischer Ferreira, Eduardo Fernandes, João Paulo Diniz, and Eduardo Figueiredo. 2024. Resource Interaction Failures in Mobile Applications: A Challenge for the Software Product Line Testing Community . In *28th ACM International Systems and Software Product Line Conference (SPLC '24), September 2–6, 2024, Luxembourg City, Luxembourg*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3382025.3414725

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 [3]. Therefore, mobile 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 [20]. Nevertheless, the particularities of mobile applications set them apart from other kinds of applications, such as Desktop or Web Systems, where traditional testing approaches can be applied [14]. The growing awareness for mobile application quality has resulted in a broad spectrum of testing techniques [20, 21]. However, despite the availability of testing methods, techniques, and tools, the field of mobile application testing is still under development [15].

Mobile applications are often executed on a variety of platform configurations [17] 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, allowing users to customize many system or application behaviors [24]. However, applications can present unexpected behaviors since the resource interactions can introduce failures that manifest themselves in specific resource combinations [25].

Resource interaction occurs when one or more resources influence the behavior of other resources, similarly to the feature interaction problem in configurable software systems [5, 6] and telecommunication systems [9]. An example of resource interaction failure occurs for Commons App when a pair of resources are disabled [42]. 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. For instance, it is 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 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 of `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 Wikipedia[1]. 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 Client[2] 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 GitHub[3], it can be seen that, even

[1] https://commons-app.github.io/
[2] https://www.traccar.org
[3] https://github.com/traccar/traccar-client-android/issues/390

if the four sensors are enabled, the application stops changing location 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 software 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 failures [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 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 [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 several 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 number 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 [4]. These applications 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, description, 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 characteristics 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[5] to control the resources. The instrumentation is based on Android instrumented tests, i.e., a type of functional test[6]. 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[7] and the sensors are managed by Sensor Disabler[8]. This application requires a rooted Android device[9].

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

[4] https://eulerhm.github.io/splc-challenge/.
[5] https://developer.android.com/training/testing/ui-automator
[6] https://developer.android.com/training/testing/instrumented-tests
[7] https://github.com/percula/LensCap
[8] https://github.com/wardellbagby/Sensor-Disabler
[9] https://en.wikipedia.org/wiki/Rooting_(Android)

Table 1: The Dataset

Name	Description		Size metrics				Test Suite metrics	
	Category	Commits	#LOC	#Packages	#Classes	#Methods	#Test	#LOCTest
AnkiDroid [4]	Education	13,643	158,607	52	695	3,575	164	2,770
CovidNow [13]	Medical	85	2,269	42	811	252	21	540
Ground [18]	Productivity	4,936	19,906	51	412	1,676	4	525
Iosched [19]	Books, Reference	3,101	27,824	29	307	1,333	9	473
Lockwise [23]	Productivity	503	14,535	12	350	1,124	38	1,184
Mixin-Messenger [28]	Finance	8,086	168,080	136	2,212	12,025	160	3,732
Moonshot [29]	Tools	351	455	7	48	196	28	464
Nekome [30]	Productivity	2,742	1,084	4	29	62	64	2,097
Nl-covid19 [32]	Medical	1,293	65,839	26	311	634	20	1,114
OpenScale [33]	Health, Fitness	2,742	27,781	19	174	1,046	14	1,451
OwnTracks [34]	Travel, Local	1,995	14,499	37	273	1,176	27	889
PocketHub [36]	Productivity	3,512	29,001	40	323	1,332	107	1,663
Radio-Droid [37]	Music	1,186	22,815	25	207	810	23	1,735
Scarlet-Notes [38]	Productivity	656	4,260	5	51	228	52	770
Showly-2.0 [39]	Entertainment	3,251	2,547	8	29	173	55	952
SpaceX-Follower [41]	News, Magazines	356	7,664	22	110	526	30	940
Threema [43]	Communication	21	238,045	123	1,718	9,763	54	1,931
Vocable [45]	Communication	863	13,188	14	106	534	14	499
Woo-Commerce [46]	Business	26,527	156,962	120	2,283	8,334	27	1,367
WordPress-Android [47]	Productivity	68,148	347,897	240	4,175	15,292	115	3,674

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[10]. 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[11] for the `Location` and TelephonyManager[12] for the `Mobile Data`. In other cases, we use the UI Automator features to find some screen widgets related to the resource state. For example, we inspect the sensors states by processing screens of Sensor Disabler. It is important to emphasize that in our implementation the resources are only adjusted (lines 6 and 8) if necessary. Besides, we modified the build scripts in order to use the Android Test Orchestrator[13], a tool that helps minimize possible shared states, a known factor associated to flaky tests [35] 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**

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**
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**

3.4 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[14], Vocable uses the ARCore SDK[15] to track the user's head movements and understand where the user is looking on the screen.

[10] https://junit.org/junit4/javadoc/4.12/org/junit/BeforeClass.html
[11] https://developer.android.com/reference/android/location/LocationManager
[12] https://developer.android.com/reference/android/telephony/TelephonyManager
[13] https://developer.android.com/training/testing/junit-runner
[14] https://github.com/willowtreeapps/vocable-android
[15] https://developers.google.com/ar

Table 2: Failure Report

Name	FS	SE	#Failures
CovidNow	32	0.47	2
Lockwise	68	1.00	4
Mixin-Messenger	20	0.29	2
Nl-covid19	55	0.81	6
OwnTracks	68	1.00	3
PocketHub	4	0.06	1
SpaceXFollower	68	1.00	4
Threema	33	0.48	1
Vocable	24	0.35	7
WordPress-Android	37	0.54	11

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

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.

Location, Bluetooth, Battery_Saver, Do_Not_Disturb, Accelerometer, Light

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[16].

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 (2^{14} = 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.

[16]https://developers.google.com/ar/reference/java/com/google/ar/core/exceptions/FatalException

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 [17]. These applications were implemented in Java and/or Kotlin.
(2) **Found Failures**: We provide the found failures for the challenge applications [18]. 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 [19]. 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.

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

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.

[17]https://github.com/eulerhm/splc-challenge/tree/master/Dataset/Vocable
[18]https://github.com/eulerhm/splc-challenge/tree/master/Found_Failures
[19]https://github.com/eulerhm/splc-challenge/tree/master/Settings

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

REFERENCES

[1] I. Abal, C. Brabrand, and A. Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 421–432.

[2] M. Krieter S. Thüm T. Lochau M. Saake G Al-Hajjaji. 2016. IncLing: efficient product-line testing using incremental pairwise sampling. In *15th Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*. 144–155.

[3] Domenico Amalfitano, Nicola Amatucci, Atif M. Memon, Porfirio Tramontana, and Anna Rita Fasolino. 2017. A general framework for comparing automatic testing techniques of Android mobile apps. *Journal of Systems and Software* 125 (2017), 322–343.

[4] AnkiDroid. [n.d.]. AnkiDroid. https://github.com/ankidroid/Anki-Android, Accessed 6-mar-2024.

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

[6] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. 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)*. 1––8.

[7] Commons App. [n.d.]. Commons Issue 1735. https://github.com/commons-app/apps-android-commons/issues/1735, Accessed 6-mar-2024.

[8] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. 2015. The oracle problem in software testing: a survey. *IEEE Transactions on Software Engineering* 41 (2015), 507–525. Issue 5.

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

[10] cloc. [n.d.]. cloc. https://github.com/mauricioaniche/ck, Accessed 6-mar-2024.

[11] codeMR. [n.d.]. codeMR. https://plugins.jetbrains.com/plugin/10811-codemr, Accessed 6-mar-2024.

[12] M. B. Cohen, M. B. Dwyer, and J. Shi. 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)*. 129–139.

[13] CovidNow. [n.d.]. CovidNow. https://github.com/OMIsie11/CovidNow, Accessed 6-mar-2024.

[14] L. Cruz, R. Abreu, and D. Lo. 2019. To the attention of mobile software developers: guess what, test your app! *Empirical Software Engineering (EMSE)* 24 (2019), 2438–2468.

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

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

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

[18] Ground. [n.d.]. Ground. https://github.com/google/ground-android, Accessed 6-mar-2024.

[19] Iosched. [n.d.]. Iosched. https://github.com/google/iosched, Accessed 6-mar-2024.

[20] Misael C Júnior, Domenico Amalfitano, Lina Garcés, Anna Rita Fasolino, Stevão A Andrade, and Márcio Delamaro. 2022. Dynamic Testing Techniques of Non-functional Requirements in Mobile Apps: A Systematic Mapping Study. *ACM Computing Surveys (CSUR)* 54, 10s (2022), 1–38.

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

[22] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov. 2020. Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 403–413.

[23] Lockwise. [n.d.]. Lockwise. https://github.com/mozilla-lockwise/lockwise-android, Accessed 6-mar-2024.

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

[25] E. H. Marinho, J. P. Diniz, F. Ferreira, and E. Figueiredo. 2021. Evaluating Sensor Interaction Failures in Mobile Applications. In *International Conference on the Quality of Information and Communications Technology (QUATIC)*. Springer, 49–63.

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

[27] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 643–654.

[28] Mixin-Messenger. [n.d.]. Mixin-Messenger. https://github.com/MixinNetwork/android-app, Accessed 6-mar-2024.

[29] Moonshot. [n.d.]. Moonshot. https://github.com/haroldadmin/MoonShot, Accessed 6-mar-2024.

[30] Nekome. [n.d.]. Nekome. https://github.com/Chesire/Nekome, Accessed 6-mar-2024.

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

[32] Nl-covid19. [n.d.]. Nl-covid19. https://github.com/minvws/nl-covid19-notification-app-android, Accessed 6-mar-2024.

[33] OpenScale. [n.d.]. OpenScale. https://github.com/oliexdev/openScale, Accessed 6-mar-2024.

[34] OwnTracks. [n.d.]. OwnTracks. https://github.com/owntracks/android, Accessed 6-mar-2024.

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

[36] PocketHub. [n.d.]. PocketHub. https://github.com/pockethub/PocketHub, Accessed 6-mar-2024.

[37] Radio-Droid. [n.d.]. Radio-Droid. https://github.com/segler-alex/RadioDroid, Accessed 6-mar-2024.

[38] Scarlet-Notes. [n.d.]. Scarlet-Notes. https://github.com/BijoySingh/Scarlet-Notes, Accessed 6-mar-2024.

[39] Showly-2.0. [n.d.]. Showly-2.0. https://github.com/michaldrabik/showly-2.0, Accessed 6-mar-2024.

[40] S. Souto, M. d'Amorim, and R. Gheyi. 2017. Balancing soundness and efficiency for practical testing of configurable systems. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 632–642.

[41] SpaceXFollower. [n.d.]. SpaceXFollower. https://github.com/OMIsie11/SpaceXFollower, Accessed 6-mar-2024.

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

[43] Threema. [n.d.]. Threema. https://github.com/threema-ch/threema-android, Accessed 6-mar-2024.

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

[45] Vocable. [n.d.]. Vocable. https://github.com/willowtreeapps/vocable-android, Accessed 6-mar-2024.

[46] Woo-Commerce. [n.d.]. Woo-Commerce. https://github.com/woocommerce/woocommerce-android, Accessed 6-mar-2024.

[47] WordPress-Android. [n.d.]. WordPress-Android. https://github.com/wordpress-mobile/WordPress-Android, Accessed 6-mar-2024.