Configurations in Android Testing: They Matter

Emily Kowalczyk
University of Maryland, College Park
College Park, MD, USA
emilyk@cs.umd.edu

Myra B. Cohen
University of Nebraska-Lincoln
Lincoln, NE, USA
myra@cse.unl.edu

Atif M. Memon
University of Maryland, College Park
College Park, MD, USA
atif@cs.umd.edu

ABSTRACT

Android has rocketed to the top of the mobile market thanks in large part to its open source model. Vendors use Android for their devices for free, and companies make customizations to suit their needs. This has resulted in a myriad of configurations that are extant in the user space today. In this paper, we show that differences in configurations, if ignored, can lead to differences in test outputs and code coverage. Consequently, researchers who develop new testing techniques and evaluate them on only one or two configurations are missing a necessary dimension in their experiments and developers who ignore this may release buggy software. In a large study on 18 apps across 88 configurations, we show that only one of the 18 apps studied showed no variation at all. The rest showed variation in either, or both, code coverage and test results. 15% of the 2,000 plus test cases across all of the apps vary, and some of the variation is subtle, i.e. not just a test crash. Our results suggest that configurations in Android testing do matter and that developers need to test using configuration-aware techniques.

CCS CONCEPTS

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

KEYWORDS

Android, Mobile Testing

ACM Reference Format:
Emily Kowalczyk, Myra B. Cohen, and Atif M. Memon. 2018. Configurations in Android Testing: They Matter. In *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis (A-Mobile '18), September 4, 2018, Montpellier, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3243218.3243219

1 INTRODUCTION

Mobile devices are becoming our primary means for performing day to day activities. They are used to conduct business, for personal well being and for entertainment. The amount of software written to run on these devices is exploding, and the mobile landscape is beginning to resemble that of highly-configurable software systems [6, 18], systems where different platforms and features can be combined in a variety of ways, e.g., apps automatically adapt to how they detect user movement based on the sensors (proximity, light, accelerometer) available. This leads to a potential combinatorial explosion in the number of possible platforms under which the software must be run and tested, potentially taxing the limit of developer resources [10].

There has been a large body of research on software testing for highly-configurable software [16, 18] where studies have shown that different configurations of the same system may fail or pass independently on the same test cases. This has led to a plethora of testing techniques to either sample the possible set of configurations, or order (prioritize) configurations to find faults as early as possible. However, most of this research has been performed in a traditional (desktop) domain. There is mounting evidence [4] that this is problematic in mobile environments as well, including both of the most popular systems, Android and iOS, where the term fragmentation has been introduced to represent a subset of the possible causes.

Modern mobile systems differ from traditional desktop systems in several ways. First, the software stack works tightly with the different hardware devices and sensors, meaning changes in hardware are likely to impact software. Second, most devices can rotate and have differing (small) screen sizes which can make rendering of information less portable. Third, there is a rapid evolution of the software development kits (SDKs). Software written today, will likely be run on newer versions of the SDK, and included libraries will have been written for earlier SDKs. Last, the need for a large number of hardware platforms or simulation environments is necessary and it is potentially expensive to test all these systems. The study that we performed in this paper cost approximately $1,600 using Google's Test Lab [7] and we would expect to incur similar costs on other testing platforms.

In recent years, multiple testing techniques and tools have been proposed [1, 3, 13–15], however, few have been configuration-aware in their techniques, implementations or evaluations [5, 8, 15, 17]. Others have pointed to a configurabilty issue, called *fragmentation*, which usually refers exclusively to the diversity in Android devices (OS customizations, display) and API versions. While developers and researchers have cited fragmentation as a challenge, its impact on test outputs remains unclear, as does the impact of other factors (locale, orientation) and their interactions.

In this paper we perform an empirical study across a range of applications, in varying environments– including different hardware, forms (emulators and physical devices), Android API levels, and orientations– to evaluate the impact of configurability on test outputs and code coverage. We run over 2,000 tests on a test matrix of 5 factors, and find that over 15% of test cases vary when run on differing configurations. The differences include code coverage, different functional outcomes and system crashes. We also find

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
A-Mobile '18, September 4, 2018, Montpellier, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5973-3/18/09…$15.00
https://doi.org/10.1145/3243218.3243219

configuration factors interact, and examine some of the causes of differences – seeing that changing configurations sometimes finds subtle test dependencies due to ordering changes across SDKs.

The contributions of this work are:

(1) A case study on tests suites run across 88 configurations spanning 5 factors including 6 physical devices, 3 virtual devices, 5 APIs, 2 locales and 2 orientations. To the best of our knowledge, this is the first study to look at how test outputs are impacted by variability across device configurations for multiple Android apps.
(2) Evidence indicating that configuration does matter (15% of all tests varying across configurations) and factors interact.

In the next section we present background and related work. We then present our Empirical Study (Section 3) and Results (Section 4). We end with conclusions and future work.

2 BACKGROUND & RELATED WORK

There has been a large body of work on testing Android applications [1–3, 5, 14, 15]. This includes random techniques such as Dynodroid which incorporates feedback [13], model-based techniques such as MobiGuitar [1] and SwiftHand [3], and search-based techniques such as EvoDroid [14] and Sapienz [15]. The notion of variability in Android is known [9, 11, 19, 20].

The term fragmentation has been used to describe variability due to vendor customizations (resulting in OS differences and UI themes), API versions, display (screen density and size), and/or hardware (CPU, RAM). Such variations have been shown to introduce testing [5] as well as performance and security challenges [12, 21] where results or vulnerabilities appear only on certain configurations. Other factors beyond those typically associated with fragmentation have been shown to impact testing as well. These include orientation of the device (i.e., landscape or portrait), localization (which may load different resources) and permissions [17].

While developers and researchers have cited fragmentation as a challenge, there is only one empirical study, which explores the features and causes of resulting compatibility issues. Recently, Wei et al. [20] evaluated 191 bug reports containing compatibility issues from 5 open source apps to find common types, causes, symptoms, and fixes. They found that issues could be divided into two types, those seen on a certain device model (i.e., device specific) and those that occur with a specific API level (i.e., non-device specific).

The existing research differs in that it studies bug reports and does not quantify the differences and impact on outcomes such as code coverage and test results. That is the focus of this this paper. Our work also differs in that it does not look at a single factor in isolation (permissions, locale, device model, API), but how their combined impact may create noisy artifacts.

3 EMPIRICAL STUDY

We focus our study on the following two research questions.

RQ1 (Variation in test outputs): How are test outputs such as test results and code coverage impacted by the variability seen across Android devices?

RQ2 (Causes of variation and interaction between factors): (a) What are the causes of variability in test results in our sample? (b) Is variation caused by one factor or do multiple factors interact?

We collect a sample of open source apps with publicly available test suites, and run each on a test matrix of over 80 configurations. We selected configurations that are representative of the diversity seen in Android devices. Our factors are device model, screen size, form, Android API level, locale and device orientation. Test results and code coverage are collected for each run, and each app's outputs are evaluated and compared across configurations for variation.

3.1 Apps Studied

We selected apps from F-Droid, an open source app repository. We placed two constraints on the sample. The apps had to have been updated in the last two years, to ensure they are likely to cover the API versions available on Test Lab's devices. Second, we required the test suite be of reasonable size, which we defined as containing 40 or more tests. This allows us to evaluate variation in suites as well as tests, while staying within a budgeted time on Test Lab.

Our final sample consists of the 18 apps listed in Table 1. We obtained this by first compiling a list of open source apps listed on the F-Droid repository on May 20, 2017. In addition, we searched online for 'Open Source Android app' using Google, and manually looked through the first 3 pages of results.

The initial search resulted in 2,135 apps, which we then filtered to include only those whose code had been updated in the last 2 years. For each of the remaining 1,170 apps, we downloaded their code and estimated the number of tests in each project by grepping the files in each app's test directories, and counting the number of test annotations and method names containing "test". All projects that were estimated to contain more than 30 tests (approx. 31) were then manually inspected and run. This cutoff was used to account for imprecision in estimating the test count. Apps that ran 40 or more tests when run locally were added to our final sample.

Table 1 presents detail for each of the apps including the date of its last update (at time of download), relevant build information, the maximum observed size of its test suite, and information related to its presence on the Play store. All tests are integration tests.

3.2 Device Configurations & Test Matrices

The independent variable in our study is the device configuration. We define a *device configuration* as a simplified description of the system that executes a test suite. Test Lab offers 4 configurable device parameters, which we use to define a device configuration. These include:

Device. This is the hardware or simulated hardware that is used to execute the tests (e.g. Nexus 5,Samsung Galaxy).

Android API level. Google releases a new Android API roughly every year. At the time of this paper, there are currently 26 API levels with the majority of devices (approximately 98.8%) spanning 11 levels (API 16-26).

Locale. Each Android device has a default locale set by the user. The locale indicates a geographical or political region, which is used by the system to localize the app and load locale-dependent resources based on the setting.

Device Orientation. The orientation of the device refers to the device's screen rotation (portrait or landscape).

Table 2 lists the values each of the variables can take in our study. The first column, device, lists the 9 device models along with their

Table 1: Sample apps. 'App' is the app's name, 'LOC' is the lines of code in the app's source code, '# of tests' is the max. number of tests run on any configuration, 'Updated' is the date of the app's last update (at the time of download) and 'Category' and 'Installs' are the app's category and # of installs if available on Google Play.

App	LOC			# of Tests	API		Updated	Commit	Category	Installs
	Java	C	XML		min	target				
Orgzly	18,532	0	4,838	357	14	25	2017-04-17	bd44aa2	Prod.	10-50k
AntennaPod	17,186	0	5,642	215	10	23	2016-12-17	e76e78c	Video	100-500k
Open Camera	23,581	0	16,741	173	15	24	2017-02-18	90f6470	Photo	10-50M
MyExpenses	40,653	0	31,010	156	9	25	2017-04-17	993bda0	Finance	500-1m
AndStatus	38,883	0	14,025	150	16	25	2017-03-15	dc410f9	Social	5-10k
AnyMemo	20,377	0	11,031	150	15	25	2017-02-08	b612944	Edu.	100-500k
ForkHub	21,423	0	9,576	143	19	25	2017-03-26	e50777e	Prod.	50-100k
KouChat	9,178	0	466	134	16	24	2016-09-05	5769417	Comm.	1-5k
PocketHub	19,839	0	9,915	133	15	25	2017-04-13	acdb0e6	Prod.	10-50k
Google I/O	26,179	0	10,981	123	16	23	2016-11-06	5b09d51	Ref.	500-1m
TopoSuite	21,693	0	11,724	109	15	25	2017-01-17	e9ea9dc	Prod.	1-5k
Loop Habit Tracker	20,892	0	7,953	105	15	25	2017-04-13	96b95ed	Prod.	500-1m
Kore	25,445	0	11,036	71	15	25	2017-02-03	1cb7787	Video	10-50m
Simple SMS remote	5,213	0	1,698	71	16	25	2017-01-02	8dfb65f	Tools	500-1k
KeePassDroid	15,415	6,238	6,408	68	3	12	2016-11-07	4115746	Tools	1-5m
Suntimes Widget	11,503	0	5,918	53	10	24	2017-04-17	35b2bbe	-	-
Poet Assistant	5,719	0	1,804	49	15	25	2017-04-23	692698d	Ref.	50-100k
Vespucci	49,035	0	32,391	47	9	24	2017-04-08	53c0291	Tools	10-50k

Table 2: Device configuration variables and values. Each column represents a dimension in our test matrix. The matrix contains all possible combinations of values for device, API, locale (loc.), and orientation (ort.) with the API value conditioned on the device chosen.

Device	API	Loc.	Ort.
Nexus 5 (VM, Ph., 1920 x 1080)	19,21,22,23		
Nexus 5 (Phys., Ph., 1920 x 1080)	19,21,22,23		
Nexus 7 (VM, Tab., 1280 x 800)	19,21,22		
Nexus 7 (Phys., Tab., 1920 x 1200)	19,21	EN/	Prt./
Nexus 9 (VM, Tab., 2048 x 1536)	21,22,23,24,25	FR	Lnd.
Nexus 9 (Phys., Tab., 2048 x 1536)	21		
LG G3 (Phys., Ph., 2560 x 1440)	19		
Galaxy Tab 3 (Phys., Tab., 600 x 1024)	19		
Galaxy S6 (Phys., Ph., 2560 x 1440)	22		

attributes such as form (virtual vs. physical), screen size (phone vs. tablet), and screen resolution. A screen is labeled a phone if it is less than 7" and a tablet if greater. The second column, API, lists the range of API versions the device runs on in Test Lab. The columns, locale and orientation, list the locales and orientations our configurations cover.

The variables and values in Table 2 are then used to build the test matrix each app is run over. The matrix contains 88 configurations obtained from all possible combinations of the values in Table 2's columns with the values for API conditioned on the device. For example, if we were to build a matrix containing only the Galaxy Tab 3 and Galaxy S6 the matrix would contain the following 8 configurations: (Galaxy Tab 3, 19, en_US, Port.), (Galaxy Tab 3, 19, en_US, Land.), (Galaxy Tab 3, 19, fr, Port.), (Galaxy Tab 3, 19, fr, Land.), (Galaxy S6, 22, en_US, Port.), (Galaxy S6, 22, en_US, Land), (Galaxy S6, 22, fr, Port.), (Galaxy S6, 22, fr, Land.).

3.3 Testing Platform

We use Google's Test Lab [7] as our platform to run tests since it provides a cloud based solution, with a full test harness for running tests on different platforms (both physical and virtual). Testers submit jobs containing an Android Package Kit (APK), a test APK containing the test suite, as well as device and configuration information that is used to form the test matrix. Test Lab then runs the tests on each configuration in the matrix, and produces test results, code coverage, and a video showing the application running during the test. For virtual devices, Test Lab creates new instances for each run of a test suite. For physical devices, all app data is deleted, and if the device allows it, a clean ROM is flashed each time [7].

3.4 Test Artifacts

The test outputs provide our dependent variables for this study. We measure several aspects of the test output. These include a test results file, code coverage report, and system log. We describe each in more detail below.

Test Results: Test results contain information about which tests ran and their success. Tests either PASS, FAIL, SKIP or are INCONCLUSIVE. INCONCLUSIVE and SKIP indicate the test failed to be executed. We do not discuss these in more detail since they are not evaluated. Tests that are executed either PASS or FAIL. A test returns FAIL if it encounters an exception before completing, and PASS if it completes with no such exception.

Code Coverage: Code coverage is collected for each run using Jacoco, the Android SDK's default code coverage library. To enable

coverage, we altered each app's build and manifest file to include required attributes and permissions, and extended the app's test runner to ensure the needed permission was granted before writing to the device. Each coverage executable is then used to generate an XML report that provides information about the total line and branch coverage of the suite as well as which lines in the code were covered during the run. We parse the report for overall statement coverage, and compare the distribution within each app.

Logcat: A logcat file (log of system messages) for the session is also collected. It provides detailed information such as when background services are run, and captures complete stack trace information for exceptions thrown. The logcat file was used for RQ2, to obtain information about failures and the state of the device.

3.5 Experimental Setup

Since it is possible for test outputs to vary due to flakiness, we ran each app twice (see Section 3.6). In order to run the tests, we were required to build the apps and their test apks ourselves. When building, we made as few changes as possible, altering the source code only to enable code coverage. To enable code coverage, we added the needed properties to the build file and permissions to the AndroidManifest.xml file. We then extended tests runners to grant the needed permissions at the end of testing, which was required for configurations running APIs greater than or equal to API 23.

All runs took place during non-business hours during August-October 2017. Combined over 290,486 instances of tests were run. The runs amounted to over 248 hours of testing on physical devices and 293 hours on virtual devices.

3.6 Calculation of Metrics

3.6.1 RQ1 (Variation in test outputs). For this RQ we evaluate two output files, their test results and code coverage, and look for differences between configurations within each app.

Each run of the test suite on a configuration outputs a test results file. The results for each configuration are then used to form a results matrix for that execution of the test matrix. To account for flakiness, we combine the results for both runs before measuring variation so we can identify tests that produced different results within configuration between runs. When combining the matrices, we resolve each cell value using the following rules: If the test passes on a configuration for both runs, the corresponding cell for the test and configuration is given the value PASS. If it fails on both runs, the cell is given the value FAIL. And if it passes during one run but fails on the other, the cell is given the value FLAKE. There were several tests that either timed out or terminated early on configurations, and had 1 result instead of 2. Due to budget restrictions, we used the result of the successful run as the test's result in the combined results matrix. Note the majority of tests (over 85%) had two results. After the results for both runs have been aggregated, we use the resulting matrix to determine the number of tests that varied between configurations, where a test is considered to vary if it passed on one configuration and failed on another.

Code coverage was also compared between configurations. We collected the overall statement coverage percentage for every run on every configuration and evaluated the distribution for each app.

3.6.2 RQ2 (Causes of variation and interaction of variables). In RQ2, we look to better understand the causes of variation. Since determing the cause of configuration-dependent failures can be difficult [20], we manually evaluate 2 apps, the app with the most variation as well as 1 randomly selected. We then look at the types of exceptions thrown in configuration-dependent failures across the entire sample.

We also constructed classification trees for each varying test and failure, where a failure is an exception type and line number thrown. For each unique failure within a test we constructed a decision tree using the output of all test runs that executed the line. All configuration variables were features, and a boolean indicating whether the exception was thrown at that line as it's target. The resulting tree provides an explanation of the variables and values that may lead the exception to surface at that line. The trees were created using Weka's J48 algorithm. The trees used 10-fold cross validation, had the minimum number of instances required to split set to 1, and were left unpruned.

In addition to causes, we also evaluated whether configuration-dependent failures occurred when multiple variables interacted. Variable interaction was first detected by the length of the path from the exception leaf to the root, and then evaluated manually.

3.7 Threats to Validity

We explore our study's primary threats to validity below.

Construct validity. Some of the variation we report may be caused by flakiness and not the configuration. We have attempted to reduce this by running the apps twice, and also accounting for signs of flakiness in our results.

External validity. Our sample contains only 18 apps. However, these are from a wide range of categories from a popular open source repository, F-Droid and we used objective criteria to filter out candidates from our sample.

Internal validity. We acknowledge that it is possible there were faults in the tools used to collect code coverage etc. However, we used a standard platform (Test Lab).

4 RESULTS

RQ1 (Variation in test outputs). Figure 1 presents the percentage of tests in each app's suite that varied. The dark blue bars show the percentage of varying tests in each app's suite, and light blue indicates the percentage of these tests that flaked on at least one configuration. The app, Vespucci, had the highest percentage of varying tests with nearly half (46.81%) of its tests varying. The app, KeePassDroid, had the least variation (0%) with all tests giving consistent results. Overall, nearly half of the apps in our sample (8 out of 18) had over 20% of their tests vary across configurations.

Variation in code coverage was less common, but did exist. Figure 2 shows the variation seen in each app's statement coverage percentages. Note two apps, Open Camera and KouChat, are not included in Figure 2. This is because a test in each app's suite killed the process, resulting in the suite not completing and code coverage not being collected. Only configurations where the suite ran to completion are included in Figure 2.

Simple SMS Remote had the largest amount of coverage variation with a range of 10.89%, while Vespucci (10.7%), Poet Assistant

(8.2%), Suntimes Widget (5.3%), and MyExpenses (3.9%) also showed variation. The apps on the left side of the figure show the most variation in code coverage. These are also the apps with the highest number of varying tests. The apps on the right side of the figure are those with the least variation in tests results and code coverage.

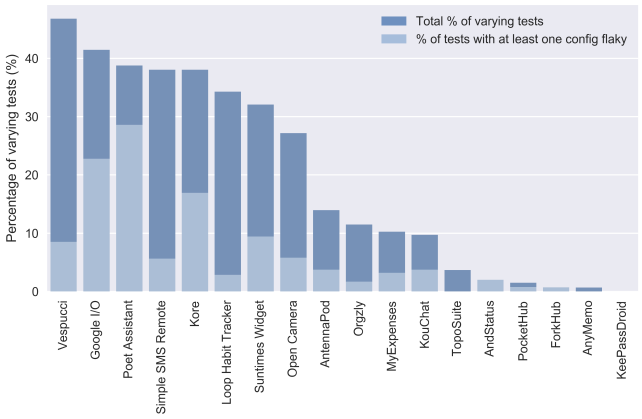


Figure 1: Variation in test results across configurations. The '% of varying tests' is the percentage of tests in the app's suite that pass on one configuration and fail on another.

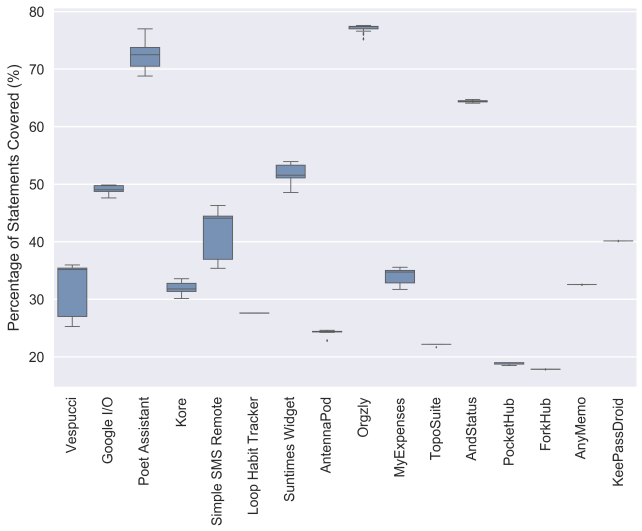


Figure 2: Variation in statement coverage across configurations.

RQ2 (Causes of variation and interaction among factors). We first look at a randomly selected app, TopoSuite, as well as the app with the most variation, Vespucci. We then look at the types of exceptions thrown by varying tests across all apps.

TopoSuite. The first app we examined was TopoSuite. TopoSuite is an application for land surveyors, and allows users to manage and perform calculations on geographical points. It had little variation in its outputs with only 4 of its 109 tests varying and code coverage variation of 3.6%. We manually inspected TopoSuite's outputs and found its varying tests were due to 3 configuration-dependent failures that surfaced in 4 tests. Table 3 shows the causes for each test. All failures were caused by a single factor with 2 tests failing due to device, 1 API, and 1 locale.

Table 3: Frequent causes of variation in TopoSuite and Vespucci. 'Cause' is the factor that caused the failure ('A' is for API, 'D' device, 'L' locale, 'O' orientation, 'F' form, 'SS' screen size). Number in each cell is the number of tests whose failure was caused by the factor.

App	Cause					
	A	D	L	O	F	SS
TopoSuite	1	2	1			
Vespucci	24	4	6	3	3	1

TopoSuite's variation occurred primarily in testing tools and libraries. The device specific failures were caused by the same exception, a RuntimeException that surfaced only on Galaxy Tab 3 devices. The error occurred in the test class's setup method, and was impacted by OS customizations that caused the View object to be created off the main UI thread – an action prohibited in Android. Another failure was caused by locale, and was due to an oracle that had not taken configuration into account. The test computes distances between location points. It fails on the FR locale, because values are converted to strings for comparison and the FR locale uses a comma for the decimal marker instead of an expected period. The last error in TopoSuite was API-dependent, and happened while asserting over floating point coordinates. The variation is caused by the ordering of tests in the suite, which differs across API versions. APIs greater than 19 ran the test after another test that changed the coordinate's precision to 20 decimal places, without reverting back to the default after testing. This resulted in all runs passing. API 19 ran the test before this test. As a result, it used the default value of 3 decimal places and failed. This example is interesting because of the cause's known relationship with flakiness. However, here runs on the same configuration return a consistent result, but vary between configurations.

Vespucci. The second app we evaluated was Vespucci. Vespucci is a map editor that allows users to customize and edit map data. It had the most variation with 22 of its 47 (46.8%) tests leading to different results and code coverage with a range of 7% (28-35%). Table 3 shows the causes. Its errors span all factors including API, device, locale, and orientation. The majority of failures were caused by API (21) alone, with other single factor causes including device (2) and locale (4). 6 failures were caused by interactions. None were due to orientation, form or screen size alone.

Much of Vespucci's variation also occurred in testing libraries and tools. For example, the majority of these tests (19/24) were VerifyErrors caused by a configuration dependent error in a test mocking library, which throws an API-dependent exception on API 19. Tests that made use of UI automation libraries were also impacted by the configuration. Several Espresso tests (4) failed because of changes in locale. These tests failed when trying to select a UI element through the content description attribute, which

changes value with locale. Another UI test failed due to 2 factors, device and API, which causes the the UiAutomator library to behave differently. The test is supposed to click on a new text field and insert text, however, on Nexus devices running API 19 the test fails to select the view and adds text to the previously selected field. The error is caused by branching behavior in the test library that fails to call a method to change focus to the newly selected view.

In general, Vespucci's test results were more difficult to analyze than TopoSuite's. One reason was that several (5) tests failed at different lines (with 1 test failing at 4 different lines on different configurations). This caused difficulties because results from other configurations were used to reason about failures, and resulted in trees for later failures having less information. Unlike Toposuite, Vespucci also had several tests where factors interacted (device and API, API and form, device and orientation, and API and locale). The app also had 2 failures which appear to require up to 3 (orientation, form, screen size) and 4 factors (device, orientation, form, and locale). However after tracing and recording tests on various configurations, both appear to be flaky.

Exceptions. A total of 8,249 exceptions were thrown spanning 30 exception types. NullPointerExceptions were the most frequently seen in our sample (1,143) as well as exceptions in UI tests (NoMatchingViewException, PerformException) (3,280). VerifyErrors, FileNotFoundExceptions, and RuntimeExceptions were also common (1,485). 4 SecurityExceptions also surfaced. 3 were on APIs greater than or equal to 23, and caused by not requesting permissions at runtime. The other required 2 factors (device and API) and was due to vendor customizations and differing file systems. The exception surfaced only on non-Google devices (LG G3 and Galazy Tab 3) running API 19 when the app's test attempted to access a file from external storage.

In summary, the test's in our sample varied due to all factors we considered (API, locale, orientation, form, screen size, and device) and spanned 30 exception types. Many of the failures occurred in testing libraries and tools, suggesting room for improvement in their implementations or that test writers may need to write tests more carefully to reduce the number of false positives. While the cause of many of these failures could be reduced to one configuration factor, single factor causes could not describe all failures we observed. Several required factors to interact with interaction of up to 2 factors verified.

5 CONCLUSIONS AND FUTURE WORK

In this paper we have examined the impact of variability on Android testing. We evaluate 18 apps with existing test suites and run these on 88 configurations by varying 5 factors– device, API, form, locale, and orientation. We show that 15% of the test cases in our study vary with respect to either (or both) code coverage and test results indicating that configuration does matter.

When we further examined the results we find some differences are due to single configuration options, but others more complex – requiring 2 or more factors to show a difference. The implications are broad suggesting both developers and researchers should consider multiple configurations when testing and writing tests. Given the high cost (time and money) to test across multiple configurations, it also suggests more efficient techniques are needed to help Android testers.

As future work we plan to explore the impact of noisy test artifacts on downstream learning and analysis. We also plan to explore different sampling techniques and extend our study to more apps.

ACKNOWLEDGEMENTS

This work is supported in part by NSF grant CCF-1745775.

REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2014. MobiGUITAR – A Tool for Automated Model-Based Testing of Mobile Apps. *IEEE Software* (2014).

[2] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Intl. Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 641–660.

[3] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Intl. Conf. on Obj. Orient. Prog. Syst. Langs. and Apps. (OOPSLA)*. 623–640.

[4] Tiago Coelho, Bruno Lima, and João Pascoal Faria. 2016. MT4A: A No-programming Test Automation Framework for Android Applications. In *Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST)*. 59–65.

[5] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso. 2017. Barista: A Technique for Recording, Encoding, and Running Platform Independent Android Tests. In *Intl. Conf. on Software Testing, Verification and Validation (ICST)*. 149–160.

[6] Nicolas Fußberger, Bo Zhang, and Martin Becker. 2017. A Deep Dive into Android's Variability Realizations. In *Intl. Systems and Software Product Line Conf. (SPLC)*. 69–78.

[7] Google. 2017. Firebase Test Lab for Android Overview. Retrieved Aug 16, 2017 from https://firebase.google.com/docs/test-lab/overview

[8] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. 2015. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *Intl. Symp on Perf. Anal. of Syst. and Soft. (ISPASS)*. 215–224.

[9] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. 2012. Understanding Android fragmentation with topic analysis of vendor-specific bugs. In *Reverse Engineering, Working Conf. on*. 83–92.

[10] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. 2017. Developer Mistakes in Writing Android Manifests: An Empirical Study of Configuration Errors. In *Intl. Conf. on Mining Software Repositories (MSR)*. 25–36.

[11] Hammad Khalid, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. 2014. Prioritizing the devices to test your app on: A case study of android game apps. In *Intl. Symp. on Foundations of Software Engineering*. 610–620.

[12] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Intl. Conf. on Software Engineering*. 1013–1024.

[13] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 224–234.

[14] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Intl. Symp. on Foundations of Software Engineering (FSE)*. 599–609.

[15] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Intl. Symp. on Software Testing and Analysis (ISSTA)*. 94–105.

[16] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. 2008. Configuration-aware Regression Testing: An Empirical Study of Sampling and Prioritization. In *Intl. Symp. on Software Testing and Analysis (ISSTA)*. 75–86.

[17] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2017. PATDroid: Permission-aware GUI Testing of Android. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 220–232.

[18] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems. In *Intl. Conf. on Automated Software Engineering*. 342–352.

[19] Sergiy Vilkomir, Katherine Marszalkowski, Chauncey Perry, and Swetha Mahendrakar. 2015. Effectiveness of multi-device testing mobile applications. In *Intl. Conf. on Mobile Software Engineering and Systems*. 44–47.

[20] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *Intl. Conf. on Automated Software Engineering*. 226–237.

[21] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014. The peril of fragmentation: Security hazards in android device driver customizations. In *Symp. on Security and Privacy (SP)*. 409–423.