Enabling Mutation Testing for Android Apps

Mario Linares-Vásquez
Universidad de los Andes
Colombia

Gabriele Bavota
Università della Svizzera italiana
Switzerland

Michele Tufano
College of William and Mary
United States

Kevin Moran
College of William and Mary
United States

Massimiliano Di Penta
University of Sannio
Italy

Christopher Vendome
College of William and Mary
United States

Carlos Bernal-Cárdenas
College of William and Mary
United States

Denys Poshyvanyk
College of William and Mary
United States

ABSTRACT

Mutation testing has been widely used to assess the fault-detection effectiveness of a test suite, as well as to guide test case generation or prioritization. Empirical studies have shown that, while mutants are generally representative of real faults, an effective application of mutation testing requires "traditional" operators designed for programming languages to be augmented with operators specific to an application domain and/or technology. This paper proposes MDroid+, a framework for effective mutation testing of Android apps. First, we systematically devise a taxonomy of 262 types of Android faults grouped in 14 categories by manually analyzing 2,023 software artifacts from different sources (*e.g.,* bug reports, commits). Then, we identified a set of 38 mutation operators, and implemented an infrastructure to automatically seed mutations in Android apps with 35 of the identified operators. The taxonomy and the proposed operators have been evaluated in terms of stillborn/trivial mutants generated and their capacity to represent real faults in Android apps, as compared to other well know mutation tools.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**;

KEYWORDS

Mutation testing, Fault taxonomy, Operators, Android

ACM Reference Format:
Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling Mutation Testing for Android Apps. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 12 pages.

https://doi.org/10.1145/3106237.3106275

1 INTRODUCTION

In the last few years mobile apps have become indispensable in our daily lives. With millions of mobile apps available for download on Google Play [24] and the Apple App Store [9], mobile users have access to an unprecedentedly large set of apps that are not only intended to provide entertainment but also to support critical activities such as banking and health monitoring. Therefore, given the increasing relevance and demand for high quality apps, industrial practitioners and academic researchers have been devoting significant effort to improving methods for measuring and assuring the quality of mobile apps. Manifestations of interest in this topic include the broad portfolio of mobile testing methods ranging from tools for assisting record and replay testing [22, 30], to automated approaches that generate and execute test cases [46, 49, 54, 68, 74], and cloud-based services for large-scale multi-device testing [2].

Despite the availability of these tools/approaches, the field of mobile app testing is still very much under development; as evidenced by limitations related to test data generation [43, 68], and concerns regarding effective assessment of the quality of mobile apps' test suites. One way to evaluate test suites is to seed small faults, called mutants, into source code and asses the ability of a suite to detect these faults [17, 27]. Such mutants have been defined in the literature to reflect the typical errors developers make when writing source code [40, 44, 51, 56, 60, 66, 78].

However, existing literature lacks a thorough characterization of bugs exhibited by mobile apps. Therefore, it is unclear whether such apps exhibit a distribution of faults similar to other systems, or if there are types of faults that require special attention. As a consequence, it is unclear whether the use of traditional mutant taxonomies [40, 44] is enough to asses test quality and drive test case generation/selection of mobile apps.

In this paper, we explore this topic focusing on apps developed for Android, the most popular mobile operating system. Android apps are characterized by GUI-centric design/interaction, event-driven programming, Inter Processes Communication (IPC), and interaction with backend and local services. In addition, there are specific characteristics of Android apps—such as permission mechanisms, Software Development Kit (SDK) version compatibility, or features of target devices—that can lead to a failure. While this

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.

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106275

set of characteristics would demand a specialized set of mutation operators that can support mutation analysis and testing, there is no available tool to date that supports mutation analysis/testing of Android apps, and relatively few (eight) mutation operators have been proposed by the research community [18]. At the same time, mutation tools for Java apps, such as Pit [15] and Major [34, 38] lack any Android-specific mutation operators, and present challenges for their use in this context, resulting in common problems such as trivial mutants that always crash at runtime or difficulties automating mutant compilation into Android PacKages (APKs).

Paper contributions. This paper aims to deal with the lack of (i) an extensive empirical evidence of the distribution of Android faults, (ii) a thorough catalog of Android-specific mutants, and (iii) an analysis of the applicability of state-of-the-art mutation tools on Android apps. We then propose a framework, MDroid+, that relies on a catalog of mutation operators inspired by a taxonomy of bugs/crashes specific for Android apps, and a profile of potential failure points automatically extracted from APKs.

As a first step, we produced a taxonomy of Android faults by analyzing a statistically significant sample of 2,023 candidate faults documented in (i) bug reports from open source apps, (ii) bug-fixing commits of open source apps; (iii) Stack Overflow discussions, (iv) the Android exception hierarchy and APIs potentially triggering such exceptions; and (v) crashes/bugs described in previous studies on Android [4, 41, 46, 53, 54, 65, 68, 77, 79]. As a result, we produced a taxonomy of 262 types of faults grouped in 14 categories, four of which relate to Android-specific faults, five to Java-related faults, and five mixed categories (Figure 1). Then, leveraging this fault taxonomy and focusing on Android-specific faults, we devised a set of 38 Android mutation operators and implemented a platform to automatically seed 35 of them. Finally, we conducted a study comparing MDroid+ with other Java and Android-specific mutation tools. The study results indicate that MDroid+, as compared to existing competitive tools, (i) is able to cover a larger number of bug types/instances present in Android app, (ii) is highly complementary to the existing tools in terms of covered bug types, and (iii) generates fewer trivial and stillborn mutants.

2 RELATED WORK

This section describes related literature and publicly available, state-of-the-art tools on mutation testing. We do not discuss the literature on testing Android apps [5, 28, 43, 46, 48, 49, 54, 68, 74], since proposing a novel approach for testing Android apps is not the main goal of this work. For further details about the concepts, recent research, and future work in the field of mutation testing, one can refer to the survey by Jia and Harman [32].

Mutation Operators. Since the introduction of mutation testing in the 70s [17, 27], researchers have tried not only to define new mutation operators for different programming languages and paradigms (*e.g.,* mutation operators have been defined for Java [44] and Python [19]) but also for specific types of software like Web applications [64] and data-intensive applications [8, 80] either to exercise their GUIs [59] or to alter complex, model-defined input data [20]. The aim of our research, which we share with prior work, is to define customized mutation operators suitable for Android applications, by relying on a solid empirical foundation.

To the best of our knowledge, the closest work to ours is that of Deng *et al.*, [18], which defined eight mutant operators aimed at introducing faults in the essential programming elements of Android apps, *i.e.,* intents, event handlers, activity lifecycle, and XML files (*e.g.,* GUI or permission files). While we share with Deng *et al.* the need for defining specific operators for the key Android programming elements, our work builds upon it by (i) empirically analyzing the distribution of faults in Android apps by manually tagging 2,023 documents, (ii) based on this distribution, defining a mutant taxonomy—complementing Java mutants—which includes a total of 38 operators tailored for the Android platform.

Mutation Testing Effectiveness and Efficiency. Several researchers have proposed approaches to measure the effectiveness and efficiency of mutation testing [6, 25, 37, 57] to devise strategies for reducing the effort required to generate effective mutant sets [3, 26, 35], and to define theoretical frameworks [32, 71]. Such strategies can complement our work, since in this paper we aim at defining new mutant operators for Android, on which effectiveness/efficiency measures or minimization strategies can be applied.

Mutation Testing Tools. Most of the available mutation testing tools are in the form of research prototypes. Concerning Java, representative tools are μJava [45], Jester [52], Major [34], Jumble [72], PIT [15], and javaLanche [69]. Some of these tools operate on the Java source code, while others inject mutants in the bytecode. For instance, μJava, Jester, and Major generate the mutants by modifying the source code, while Jumble, PIT, and javaLanche perform the mutations in the bytecode. When it comes to Android apps, there is only one available tool, namely muDroid [76], which performs the mutations at byte code level by generating one APK (*i.e.,* one version of the mobile app) for each mutant. The tools for mutation testing can be also categorized according to the tool's capabilities (*e.g.,* the availability of automatic tests selection). A thorough comparison of these tools is out of the scope of this paper. The interested reader can find more details on PIT's website [14] and in the paper by Madeysky and Radyk [47].

Empirical Studies on Mutation Testing. Daran and Thévenod-Fosse [16] were the first to empirically compare mutants and real faults, finding that the set of errors and failures they produced with a given test suite were similar. Andrews *et al.* [6, 7] studied whether mutant-generated faults and faults seeded by humans can be representative of real faults. The study showed that carefully-selected mutants are not easier to detect than real faults, and can provide a good indication of test suite adequacy, whereas human-seeded faults can likely produce underestimates. Just *et al.* [36] correlated mutant detection and real fault detection using automatically and manually generated test suites. They found that these two variables exhibit a statistically significant correlation. At the same time, their study pointed out that traditional Java mutants need to be complemented by further operators, as they found that around 17% of faults were not related to mutants.

3 A TAXONOMY OF CRASHES/BUGS IN ANDROID APPS

To the best of our knowledge there is currently no (i) large-scale study describing a taxonomy of bugs in Android apps, or (ii) comprehensive mutation framework including operators derived from such a taxonomy and targeting mobile-specific faults (the only framework available is the one with eight mutation operators proposed by Deng *et al.* [18]). In this section, we describe a taxonomy of bugs in Android apps derived from a large manual analysis of (un)structured sources. Our work is the first large-scale data driven effort to design such a taxonomy. Our purpose is to extend/complement previous studies analyzing bugs/crashes in Android apps and to provide a large taxonomy of bugs that can be used to design mutation operators. In all the cases reported below the manually analyzed sets of sources—randomly extracted—represent a 95% statistically significant sample with a 5% confidence interval.

3.1 Design

To derive such a taxonomy we manually analyzed six different sources of information described below:

(1) *Bug reports of Android open source apps.* Bug reports are the most obvious source to mine in order to identify typical bugs affecting Android apps. We mined the issue trackers of 16,331 open source Android apps hosted on GitHub. Such apps have been identified by locally cloning all Java projects (381,161) identified through GitHub's API and searching for projects with an *AndroidManifest.xml* file (a requirement for Android apps) in the top-level directory. We then removed forked projects to avoid duplicated apps and filtered projects that did not have a single star or watcher to avoid abandoned apps. We utilized a web crawler to mine the GitHub issue trackers. To be able to analyze the bug cause, we only selected closed issues (*i.e.,* those having a fix that can be inspected) having "Bug" as type. Overall, we collected 2,234 issues from which we randomly sampled 328 for manual inspection.

(2) *Bug-fixing commits of Android open source apps.* Android apps are often developed by very small teams [33, 55]. Thus, it is possible that some bugs are not documented in issue trackers but quickly discussed by the developers and then directly fixed. This might be particularly true for bugs having a straightforward solution. Thus, we also mined the versioning system of the same 16,331 Android apps considered for the bug reports by looking for bug-fixing commits not related to any of the bugs considered in the previous point (*i.e.,* the ones documented in the issue tracker). With the cloned repositories, we utilized the *git* command line utility to extract the commit notes and matched the ones containing lexical patterns indicating bug fixing activities, *e.g.,"fix issue"*, *"fixed bug"*, similarly to the approach proposed by Fischer *et al.* [21]. By exploiting this procedure we collected 26,826 commits, from which we randomly selected a statistically significant sample of 376 commits for manual inspection.

(3) *Android-related Stack Overflow (SO) discussions.* It is not unusual for developers to ask help on SO for bugs they are experiencing and having difficulty fixing [10, 39, 42, 67]. Thus, mining SO discussions could provide additional hints on the types of bugs experienced by Android developers. To this aim, we collected all 51,829 discussions tagged "Android" from SO. Then, we randomly extracted a statistically significant sample of 377 of them for the manual analysis.

(4) *The exception hierarchy of the Android APIs.* Uncaught exceptions and statements throwing exceptions are a major source of faults in Android apps [13, 79]. We automatically crawled the official Android developer JavaDoc guide to extract the exception hierarchy and API methods throwing exceptions. We collected 5,414 items from which we sampled 360 of them for manual analysis.

(5) *Crashes/bugs described in previous studies on Android apps.* 43 papers related to Android testing[1] were analyzed by looking for crashes/bugs reported in the papers. For each identified bug, we kept track of the following information: app, version, bug id, bug description, bug URL. When we were not able to identify some of this information, we contacted the paper's authors. In the 43 papers, a total of 365 bugs were mentioned/reported; however, we were able (in some cases with the authors' help) to identify the app and the bug descriptions for only 182 bugs/issues (from nine papers [4, 41, 46, 53, 54, 65, 68, 77, 79]). Given the limited number, in this case we considered all of them in our manual analysis.

(6) *Reviews posted by users of Android apps on the Google Play store.* App store reviews have been identified as a prominent source of bugs and crashes in mobile apps [31, 39, 61–63, 73]. However, only a reduced set of reviews are in fact informative and useful for developers [12, 62]. Therefore, to automatically detect informative reviews reporting bugs and crashes, we leverage CLAP, the tool developed by Villarroel *et al.* [75], to automatically identify the bug-reporting reviews. Such a tool has been shown to have a precision of 88% in identifying this specific type of review. We ran CLAP on the Android user reviews dataset made available by Chen *et al.* [11]. This dataset reports user reviews for multiple releases of ~21K apps, in which CLAP identified 718,132 reviews as bug-reporting. Our statistically significant sample included 384 reviews that we analyzed.

The data collected from the six sources listed above was manually analyzed by the eight authors following a procedure inspired by open coding [50]. In particular, the 2,007 documents (*e.g.,* bug reports, user reviews, *etc.*) to manually validate were equally and randomly distributed among the authors making sure that each document was classified by two authors. The goal of the process was to identify the exact reason behind the bug and to define a tag (*e.g., null GPS position*) describing such a reason. Thus, when inspecting a bug report, we did not limit our analysis to the reading of the bug description, but we analyzed (i) the whole discussion performed by the developers, (ii) the commit message related to the bug fixing, and (iii) the patch used to fix the bug (*i.e.,* source code diff). The tagging process was supported by a Web application that we developed to classify the documents (*i.e.,* to describe the reason behind the bug) and to solve conflicts between the authors. Each author independently tagged the documents assigned to him by defining a tag describing the cause behind a bug. Every time the authors had to tag a document, the Web application also shows the list of tags created so far, allowing the tagger to select one of the already defined tags. Although, in principle, this is against the notion of open coding, in a context like the one encountered in this work, where the number of possible tags (*i.e.,* cause behind the bug) is extremely high, such a choice helps using consistent naming and does not introduce a substantial bias.

[1] The complete list of papers is provided with our online appendix [1].

In cases for which there was no agreement between the two evaluators (∼43% of the classified documents), the document was automatically assigned to an additional evaluator. The process was iterated until all the documents were classified by the absolute majority of the evaluators with the same tag. When there was no agreement after all eight authors tagged the same document (*e.g.,* four of them used the tag t_1 and the other four the tag t_2), two of the authors manually analyzed these cases in order to solve the conflict and define the most appropriate tag to assign (this happened for ∼22% of the classified documents). It is important to note that the Web application did not consider documents tagged as *false positive* (*e.g.,* a bug report that does not report an actual bug in an Android app) in the count of the documents manually analyzed. This means that, for example, to reach the 328 bug reports to manually analyze and tag, we had to analyze 400 bug reports (since 72 were tagged as false positives).

It is important to point out that, during the tagging, we discovered that for user reviews, except for very few cases, it was impossible (without internal knowledge of an app's source code) to infer the likely cause of the failure (fault) by only relying on what was described in the user review. For this reason, we decided to discard user reviews from our analysis, and this left us with 2,007-384=1,623 documents to manually analyze.

After having manually tagged all the documents (overall, 2,023 = 1,623 + 400 additional documents, since 400 false positives were encountered in the tagging process), all the authors met online to refine the identified tags by merging similar ones and splitting generic ones when needed. Also, in order to build the fault taxonomy, the identified tags were clustered in cohesive groups at two different levels of abstraction, *i.e.,* categories and subcategories. Again, the grouping was performed over multiple iterations, in which tags were moved across categories, and categories merged/split.

Finally, the output of this step was (i) a taxonomy of representative bugs for Android apps, and (ii) the assignment of the analyzed documents to a specific tag describing the reason behind the bug reported in the document.

3.2 The Defined Taxonomy

Figure 1 depicts the taxonomy that we obtained through the manual coding. The black rectangle in the bottom-right part of Figure 1 reports the number of documents tagged as *false positive* or as *unclear.* The other rectangles—marked with the Android and/or with the Java logo—represent the 14 high-level categories that we identified. Categories marked with the Android logo (*e.g.,* Activities and Intents) group together Android-specific bugs while those marked with the Java logo (*e.g.,* Collections and Strings) group bugs that could affect any Java application. Both symbols together indicate categories featuring both Android-specific and Java-related bugs (see *e.g.,* I/O). The number reported in square brackets indicates the bug instances (from the manually classified sample) belonging to each category. Inner rectangles, when present, represent sub-categories, *e.g., Responsiveness/Battery Drain* in *Non-functional Requirements.* Finally, the most fine-grained levels, represented as lighter text, describe the specific type of faults as labeled using our manually-defined tags, *e.g.,* the *Invalid resource ID* tag under the sub-category *Resources,* in turn part of the *Android programming* category. The analysis of Figure 1 allows to note that:

(1) *We were able to classify the faults reported in 1,230 documents (e.g., bug reports, commits, etc.).* This number is obtained by subtracting from the 2,023 tagged documents the 400 tagged as *false positives* and the 393 tagged as *unclear.*
(2) *Of these 1,230, 26% (324) are grouped in categories only reporting Android-related bugs.* This means that more than one fourth of the bugs present in Android apps are specific of this architecture, and not shared with other types of Java systems. Also, this percentage clearly represents an underestimation. Indeed, Android-specific bugs are also present in the previously mentioned "mixed" categories (*e.g.,* in *Non-functional requirements* 25 out of the 26 instances present in the *Responsiveness/Battery Drain* subcategory are Android-specific—all but *Performance (unnecessary computation)*). From a more detailed count, after including also the Android-specific bugs in the "mixed" categories, we estimated that 35% (430) of the identified bugs are Android-specific.
(3) *As expected, several bugs are related to simple Java programming.* This holds for 800 of the identified bugs (65%).

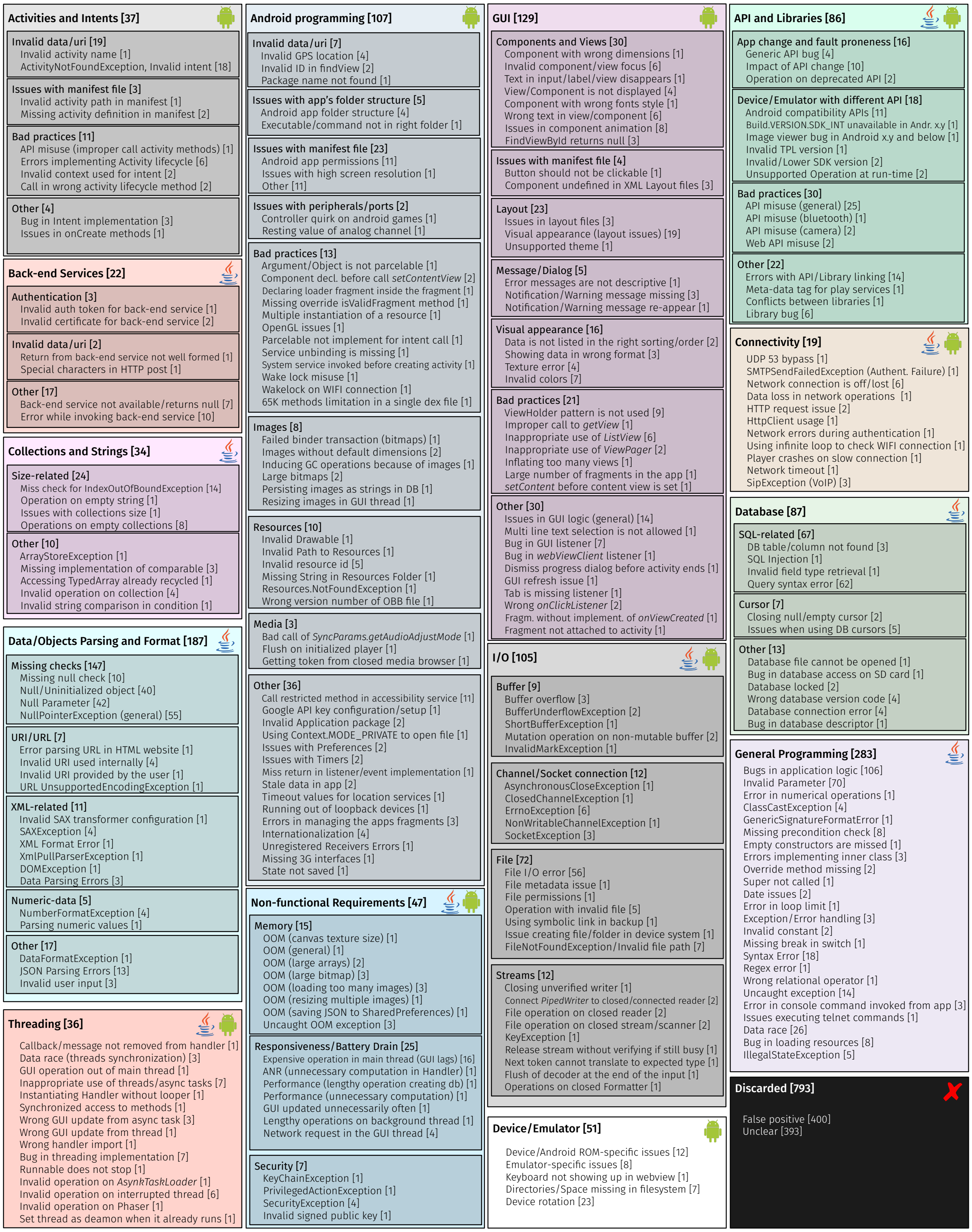
Take-away. Over one third (35%) of the bugs we identified with manual inspection are Android-specific. This highlights the importance of having testing instruments, such as mutation operators, tailored for such a specific type of software. At the same time, 65% of the bugs that are typical of any Java application confirm the importance of also considering standard testing tools developed for Java, including mutation operators, when performing verification and validation activities of Android apps.

4 MUTATION OPERATORS FOR ANDROID

Given the taxonomy of faults in Android apps and the set of available operators widely used for Java applications, a catalog of Android-specific mutation operators should (i) complement the classic Java operators, (ii) be representative of the faults exhibited by Android apps, (iii) reduce the rate of still-born and trivial mutants, and (iv) consider faults that can be simulated by modifying statements/elements in the app source code and resources (*e.g.,* the `strings.xml` file). The last condition is based on the fact that some faults cannot be simulated by changing the source code, like in the case of device specific bugs, or bugs related to the API and third-party libraries.

Following the aforementioned conditions, we defined a set of 38 operators, trying to cover as many fault categories as possible (10 out of the 14 categories in Figure 1), and complementing the available Java mutation operators. The reasons for not including operators from the other four categories are:

(1) API/Libraries: bugs in this category are related to API/Library issues and API misuses. The former will require applying operators to the APIs; the latter requires a deeper analysis of the specific API usage patterns inducing the bugs;
(2) Collections/Strings: most of the bugs in this category can be induced with classic Java mutation operators;
(3) Device/Emulator: because this type of bug is Device/Emulator specific, their implementation is out of the scope of source code mutations;
(4) Multi-threading: the detection of the places for applying the corresponding mutations is not trivial; therefore, this category will be considered in future work.



Activities and Intents [37]

- Invalid data/uri [19]
  - Invalid activity name [1]
  - ActivityNotFoundException, Invalid intent [18]
- Issues with manifest file [3]
  - Invalid activity path in manifest [1]
  - Missing activity definition in manifest [2]
- Bad practices [11]
  - API misuse (improper call activity methods) [1]
  - Errors implementing Activity lifecycle [6]
  - Invalid context used for intent [2]
  - Call in wrong activity lifecycle method [2]
- Other [4]
  - Bug in Intent implementation [3]
  - Issues in onCreate methods [1]

Back-end Services [22]

- Authentication [3]
  - Invalid auth token for back-end service [1]
  - Invalid certificate for back-end service [2]
- Invalid data/uri [2]
  - Return from back-end service not well formed [1]
  - Special characters in HTTP post [1]
- Other [17]
  - Back-end service not available/returns null [7]
  - Error while invoking back-end service [10]

Collections and Strings [34]

- Size-related [24]
  - Miss check for IndexOutOfBoundException [14]
  - Operation on empty string [1]
  - Issues with collections size [1]
  - Operations on empty collections [8]
- Other [10]
  - ArrayStoreException [1]
  - Missing implementation of comparable [3]
  - Accessing TypedArray already recycled [1]
  - Invalid operation on collection [4]
  - Invalid string comparison in condition [1]

Data/Objects Parsing and Format [187]

- Missing checks [147]
  - Missing null check [10]
  - Null/Uninitialized object [40]
  - Null Parameter [42]
  - NullPointerException (general) [55]
- URI/URL [7]
  - Error parsing URL in HTML website [1]
  - Invalid URI used internally [4]
  - Invalid URI provided by the user [1]
  - URL UnsupportedEncodingException [1]
- XML-related [11]
  - Invalid SAX transformer configuration [1]
  - SAXException [4]
  - XML Format Error [1]
  - XmlPullParserException [1]
  - DOMException [1]
  - Data Parsing Errors [3]
- Numeric-data [5]
  - NumberFormatException [4]
  - Parsing numeric values [1]
- Other [17]
  - DataFormatException [1]
  - JSON Parsing Errors [13]
  - Invalid user input [3]

Threading [36]

- Callback/message not removed from handler [1]
- Data race (threads synchronization) [3]
- GUI operation out of main thread [1]
- Inappropriate use of threads/async tasks [7]
- Instantiating Handler without looper [1]
- Synchronized access to methods [1]
- Wrong GUI update from async task [3]
- Wrong GUI update from thread [1]
- Wrong handler import [1]
- Bug in threading implementation [7]
- Runnable does not stop [1]
- Invalid operation on *AsynkTaskLoader* [1]
- Invalid operation on interrupted thread [6]
- Invalid operation on Phaser [1]
- Set thread as deamon when it already runs [1]

Android programming [107]

- Invalid data/uri [7]
  - Invalid GPS location [4]
  - Invalid ID in findView [2]
  - Package name not found [1]
- Issues with app's folder structure [5]
  - Android app folder structure [4]
  - Executable/command not in right folder [1]
- Issues with manifest file [23]
  - Android app permissions [11]
  - Issues with high screen resolution [1]
  - Other [11]
- Issues with peripherals/ports [2]
  - Controller quirk on android games [1]
  - Resting value of analog channel [1]
- Bad practices [13]
  - Argument/Object is not parcelable [1]
  - Component decl. before call *setContentView* [2]
  - Declaring loader fragment inside the fragment [1]
  - Missing override isValidFragment method [1]
  - Multiple instantiation of a resource [1]
  - OpenGL issues [1]
  - Parcelable not implement for intent call [1]
  - Service unbinding is missing [1]
  - System service invoked before creating activity [1]
  - Wake lock misuse [1]
  - Wakelock on WIFI connection [1]
  - 65K methods limitation in a single dex file [1]
- Images [8]
  - Failed binder transaction (bitmaps) [1]
  - Images without default dimensions [2]
  - Inducing GC operations because of images [1]
  - Large bitmaps [2]
  - Persisting images as strings in DB [1]
  - Resizing images in GUI thread [1]
- Resources [10]
  - Invalid Drawable [1]
  - Invalid Path to Resources [1]
  - Invalid resource id [5]
  - Missing String in Resources Folder [1]
  - Resources.NotFoundException [1]
  - Wrong version number of OBB file [1]
- Media [3]
  - Bad call of *SyncParams.getAudioAdjustMode* [1]
  - Flush on initialized player [1]
  - Getting token from closed media browser [1]
- Other [36]
  - Call restricted method in accessibility service [11]
  - Google API key configuration/setup [1]
  - Invalid Application package [2]
  - Using Context.MODE_PRIVATE to open file [1]
  - Issues with Preferences [2]
  - Issues with Timers [2]
  - Miss return in listener/event implementation [1]
  - Stale data in app [2]
  - Timeout values for location services [1]
  - Running out of loopback devices [1]
  - Errors in managing the apps fragments [3]
  - Internationalization [4]
  - Unregistered Receivers Errors [1]
  - Missing 3G interfaces [1]
  - State not saved [1]

Non-functional Requirements [47]

- Memory [15]
  - OOM (canvas texture size) [1]
  - OOM (general) [1]
  - OOM (large arrays) [2]
  - OOM (large bitmap) [3]
  - OOM (loading too many images) [3]
  - OOM (resizing multiple images) [1]
  - OOM (saving JSON to SharedPreferences) [1]
  - Uncaught OOM exception [3]
- Responsiveness/Battery Drain [25]
  - Expensive operation in main thread (GUI lags) [16]
  - ANR (unnecessary computation in Handler) [1]
  - Performance (lengthy operation creating db) [1]
  - Performance (unnecessary computation) [1]
  - GUI updated unnecessarily often [1]
  - Lengthy operations on background thread [1]
  - Network request in the GUI thread [4]
- Security [7]
  - KeyChainException [1]
  - PrivilegedActionException [1]
  - SecurityException [4]
  - Invalid signed public key [1]

GUI [129]

- Components and Views [30]
  - Component with wrong dimensions [1]
  - Invalid component/view focus [6]
  - Text in input/label/view disappears [1]
  - View/Component is not displayed [4]
  - Component with wrong fonts style [1]
  - Wrong text in view/component [6]
  - Issues in component animation [8]
  - FindViewById returns null [3]
- Issues with manifest file [4]
  - Button should not be clickable [1]
  - Component undefined in XML Layout files [3]
- Layout [23]
  - Issues in layout files [3]
  - Visual appearance (layout issues) [19]
  - Unsupported theme [1]
- Message/Dialog [5]
  - Error messages are not descriptive [1]
  - Notification/Warning message missing [3]
  - Notification/Warning message re-appear [1]
- Visual appearance [16]
  - Data is not listed in the right sorting/order [2]
  - Showing data in wrong format [3]
  - Texture error [4]
  - Invalid colors [7]
- Bad practices [21]
  - ViewHolder pattern is not used [9]
  - Improper call to *getView* [1]
  - Inappropriate use of *ListView* [6]
  - Inappropriate use of *ViewPager* [2]
  - Inflating too many views [1]
  - Large number of fragments in the app [1]
  - *setContent* before content view is set [1]
- Other [30]
  - Issues in GUI logic (general) [14]
  - Multi line text selection is not allowed [1]
  - Bug in GUI listener [7]
  - Bug in *webViewClient* listener [1]
  - Dismiss progress dialog before activity ends [1]
  - GUI refresh issue [1]
  - Tab is missing listener [1]
  - Wrong *onClickListener* [2]
  - Fragm. without implement. of *onViewCreated* [1]
  - Fragment not attached to activity [1]

I/O [105]

- Buffer [9]
  - Buffer overflow [3]
  - BufferUnderflowException [2]
  - ShortBufferException [1]
  - Mutation operation on non-mutable buffer [2]
  - InvalidMarkException [1]
- Channel/Socket connection [12]
  - AsynchronousCloseException [1]
  - ClosedChannelException [1]
  - ErrnoException [6]
  - NonWritableChannelException [1]
  - SocketException [3]
- File [72]
  - File I/O error [56]
  - File metadata issue [1]
  - File permissions [1]
  - Operation with invalid file [5]
  - Using symbolic link in backup [1]
  - Issue creating file/folder in device system [1]
  - FileNotFoundException/Invalid file path [7]
- Streams [12]
  - Closing unverified writer [1]
  - Connect *PipedWriter* to closed/connected reader [2]
  - File operation on closed reader [2]
  - File operation on closed stream/scanner [2]
  - KeyException [1]
  - Release stream without verifying if still busy [1]
  - Next token cannot translate to expected type [1]
  - Flush of decoder at the end of the input [1]
  - Operations on closed Formatter [1]

Device/Emulator [51]

- Device/Android ROM-specific issues [12]
- Emulator-specific issues [8]
- Keyboard not showing up in webview [1]
- Directories/Space missing in filesystem [7]
- Device rotation [23]

API and Libraries [86]

- App change and fault proneness [16]
  - Generic API bug [4]
  - Impact of API change [10]
  - Operation on deprecated API [2]
- Device/Emulator with different API [18]
  - Android compatibility APIs [11]
  - Build.VERSION.SDK_INT unavailable in Andr. x.y [1]
  - Image viewer bug in Android x.y and below [1]
  - Invalid TPL version [1]
  - Invalid/Lower SDK version [2]
  - Unsupported Operation at run-time [2]
- Bad practices [30]
  - API misuse (general) [25]
  - API misuse (bluetooth) [1]
  - API misuse (camera) [2]
  - Web API misuse [2]
- Other [22]
  - Errors with API/Library linking [14]
  - Meta-data tag for play services [1]
  - Conflicts between libraries [1]
  - Library bug [6]

Connectivity [19]

- UDP 53 bypass [1]
- SMTPSendFailedException (Authent. Failure) [1]
- Network connection is off/lost [6]
- Data loss in network operations [1]
- HTTP request issue [2]
- HttpClient usage [1]
- Network errors during authentication [1]
- Using infinite loop to check WIFI connection [1]
- Player crashes on slow connection [1]
- Network timeout [1]
- SipException (VoIP) [3]

Database [87]

- SQL-related [67]
  - DB table/column not found [3]
  - SQL Injection [1]
  - Invalid field type retrieval [1]
  - Query syntax error [62]
- Cursor [7]
  - Closing null/empty cursor [2]
  - Issues when using DB cursors [5]
- Other [13]
  - Database file cannot be opened [1]
  - Bug in database access on SD card [1]
  - Database locked [2]
  - Wrong database version code [4]
  - Database connection error [4]
  - Bug in database descriptor [1]

General Programming [283]

- Bugs in application logic [106]
- Invalid Parameter [70]
- Error in numerical operations [1]
- ClassCastException [4]
- GenericSignatureFormatError [1]
- Missing precondition check [8]
- Empty constructors are missed [1]
- Errors implementing inner class [3]
- Override method missing [2]
- Super not called [1]
- Date issues [2]
- Error in loop limit [1]
- Exception/Error handling [3]
- Invalid constant [2]
- Missing break in switch [1]
- Syntax Error [18]
- Regex error [1]
- Wrong relational operator [1]
- Uncaught exception [14]
- Error in console command invoked from app [3]
- Issues executing telnet commands [1]
- Data race [26]
- Bug in loading resources [8]
- IllegalStateException [5]

Discarded [793]

- False positive [400]
- Unclear [393]

Figure 1: The defined taxonomy of Android bugs.

Table 1: Proposed mutation operators. The table lists the operator names, detection strategy (AST or TEXTual), the fault category (Activity/Intents, Android Programming, Back-End Services, Connectivity, Data, DataBase, General Programming, GUI, I/O, Non-Functional Requirements), and a brief operator description. The operators indicated with * are not implemented in MDroid+ yet.

Mutation Operator	Det.	Cat.	Description
ActivityNotDefined	Text	A/I	Delete an activity <android:name="Activity"/> entry in the Manifest file
DifferentActivityIntentDefinition	AST	A/I	Replace the Activity.class argument in an Intent instantiation
InvalidActivityName	Text	A/I	Randomly insert typos in the path of an activity defined in the Manifest file
InvalidKeyIntentPutExtra	AST	A/I	Randomly generate a different key in an Intent.putExtra(key, value) call
InvalidLabel	Text	A/I	Replace the attribute "android:label" in the Manifest file with a random string
NullIntent	AST	A/I	Replace an Intent instantiation with null
NullValueIntentPutExtra	AST	A/I	Replace the value argument in an Intent.putExtra(key, value) call with new Parcelable[0]
WrongMainActivity	Text	A/I	Randomly replace the main activity definition with a different activity
MissingPermissionManifest	Text	AP	Select and remove an <uses-permission /> entry in the Manifest file
NotParcelable	AST	AP	Select a parcelable class, remove"implements Parcelable" and the @override annotations
NullGPSLocation	AST	AP	Inject a Null GPS location in the location services
SDKVersion	Text	AP	Randomly mutate the integer values in the SdkVersion-related attributes
WrongStringResource	Text	AP	Select a <string /> entry in /res/values/strings.xml file and mutate the string value
NullBackEndServiceReturn	AST	BES	Assign null to a response variable from a back-end service
BluetoothAdapterAlwaysEnabled	AST	C	Replace a BluetoothAdapter.isEnabled() call with"tru"
NullBluetoothAdapter	AST	C	Replace a BluetoothAdapter instance with null
InvalidURI	AST	D	If URIs are used internally, randomly mutate the URIs
ClosingNullCursor	AST	DB	Assign a cursor to null before it is closed
InvalidIndexQueryParameter	AST	DB	Randomly modify indexes/order of query parameters
InvalidSQLQuery	AST	DB	Randomly mutate a SQL query
InvalidDate	AST	GP	Set a random Date to a date object
InvalidMethodCallArgument*	AST	GP	Randomly mutate a method call argument of a basic type
NotSerializable	AST	GP	Select a serializable class, remove "implements Serializable"
NullMethodCallArgument*	AST	GP	Randomly set null to a method call argument
BuggyGUIListener	AST	GUI	Delete action implemented in a GUI listener
FindViewByIdReturnsNull	AST	GUI	Assign a variable (returned by Activity.findViewById) to null
InvalidColor	Text	GUI	Randomly change colors in layout files
InvalidIDFindView	AST	GUI	Replace the id argument in an Activitity.findViewById call
InvalidViewFocus*	AST	GU	IRandomly focus a GUI component
ViewComponentNotVisible	AST	GUI	Set visible attribute (from a View) to false
InvalidFilePath	AST	I/O	Randomly mutate paths to files
NullInputStream	AST	I/O	Assign an input stream (*e.g.,* reader) to null before it is closed
NullOutputStream	AST	I/O	Assign an output stream (*e.g.,* writer) to null before it is closed
LengthyBackEndService	AST	NFR	Inject large delay right-after a call to a back-end service
LengthyGUICreation	AST	NFR	Insert a long delay (*i.e.,* Thread.sleep(..)) in the GUI creation thread
LengthyGUIListener	AST	NFR	Insert a long delay (*i.e.,* Thread.sleep(..)) in the GUI listener thread
LongConnectionTimeOut	AST	NFR	Increase the time-out of connections to back-end services
OOMLargeImage	AST	NFR	Increase the size of bitmaps by explicitly setting large dimensions

The list of defined mutation operators is provided in Table 1 and these operators were implemented in a tool named MDroid+. In the context of this paper, we define a Potential Failure Profile (PFP) that sipulates locations of the analyzed apps—which can be source code statements, XML tags or locations in other resource files—that can be the source of a potential fault, given the faults catalog from Section 3. Consequently, the PFP lists the locations where a mutation operator can be applied.

In order the extract the PFP, MDroid+ statically analyzes the targeted mobile app, looking for locations where the operators from Table 1 can be implemented. The locations are detected automatically by parsing XML files or through AST-based analysis for detecting the location of API calls. Given an automatically derived PFP for an app, and the catalog of Android-specific operators, MDroid+ generates a mutant for each location in the PFP. Mutants are initially generated as clones (at source code-level) of the original app, and then the clones are automatically compiled/built into individual Android Packages (APKs). Note that each location in the PFP is related to a mutation operator. Therefore, given a location entry in the PFP, MDroid+ automatically detects the corresponding mutation operator and applies the mutation in the source code. Details of the detection rules and code transformations applied with each operator are provided in our replication package [1].

It is worth noting that from our catalog of Android-specific operators only two operators (DifferentActivityIntentDefinition and MissingPermissionManifest) overlap with the eight operators proposed by Deng *et al.*, [18]. Future work will be devoted to cover a larger number of fault categories and define/implement a larger number of operators.

5 APPLYING MUTATION TESTING OPERATORS TO ANDROID APPS

The *goal* of this study is to: (i) understand and compare the ***applicability*** of MDroid+ and other currently available mutation testing tools to Android apps; (ii) to understand the ***underlying reasons*** for mutants—generated by these tools—that cannot be considered useful for the mutant analysis purposes, *i.e.,* mutants that do not compile or cannot be launched. This study is conducted from the *perspective* of researchers interested in improving current tools and approaches for mutation testing in the context of mobile apps. The study addresses the following research questions:

- **RQ$_1$:** *Are the mutation operators (available for Java and Android apps) representative of real bugs in Android apps?*
- **RQ$_2$:** *What is the rate of stillborn mutants (e.g., those leading to failed compilations) and trivial mutants (e.g., those leading to crashes on app launch) produced by the studied tools when used with Android apps?*

- **RQ$_3$:** *What are the major causes for stillborn and trivial mutants produced by the mutation testing tools when applied to Android apps?*

To answer **RQ$_1$**, we measured the applicability of operators from seven mutation testing tools (Major [34], PIT [15], μJava [45], Javalanche [69], muDroid [76], Deng *et al.* [18], and MDroid+) in terms of their ability of representing real Android apps' faults documented in a sample of software artifacts not used to build the taxonomy presented in Section 3. To answer **RQ$_2$**, we used a representative subset of the aforementioned tools to generate mutants for 55 open source Android apps, quantitatively and qualitatively examining the stillborn and trivial mutants generated by each tool. Finally, to answer **RQ$_3$**, we manually analyzed the mutants and their crash outputs to qualitatively determine the reasons for trivial and stillborn mutants generated by each tool.

5.1 Study Context and Data Collection

To answer **RQ$_1$**, we analyzed the complete list of 102 mutation operators from the seven considered tools to investigate their ability to "*cover*" bugs described in 726 artifacts[2] (103 exceptions hierarchy and API methods throwing exceptions, 245 bug-fixing commits from GitHub, 176 closed issues from GitHub, and 202 questions from SO). Such 726 documents were randomly selected from the dataset built for the taxonomy definition (see Section 3.1) by excluding the ones already tagged and used in the taxonomy. The documents were manually analyzed by the eight authors using the same exact procedure previously described for the taxonomy building (*i.e.,* two evaluators per document having the goal of tagging the type of bug described in the document; conflicts solved by using a majority-rule schema; tagging process supported by a Web app—details in Section 3.1). We targeted the tagging of ∼150 documents per evaluator (600 overall documents considering eight evaluators and two evaluations per document). However, some of the authors tagged more documents, leading to the considered 726 documents. Note that we did not constrain the tagging of the bug type to the ones already present in our taxonomy (Figure 1): The evaluations were free to include new types of previously unseen bugs.

We answer RQ$_1$ by reporting (i) the new bug types we identified in the tagging of the additional 726 documents (*i.e.,* the ones not present in our original taxonomy), (ii) the *coverage* level ensured by each of the seven mutation tools, measured as the percentage of bug types and bug instances identified in the 726 documents covered by its operators. We also analyze the complementarity of MDroid+ with respect to the existing tools.

Concerning **RQ$_2$** and **RQ$_3$**, we compare MDroid+ with two popular open source mutation testing tools (Major and PIT), which are available and can be tailored for Android apps, and with one context-specific mutation testing tool for Android called muDroid [18]. We chose these tools because of their diversity (in terms of functionality and mutation operators), their compatibility with Java, and their representativeness of tools working at different representation levels: source code, Java bytecode, and smali bytecode (*i.e.,* Android-specific bytecode representation).

To compare the applicability of each mutation tool, we need a set of Android apps that meet certain constraints: (i) the source code of the apps must be available, (ii), the apps should be representative of different categories, and (iii) the apps should be compilable (*e.g.,* including proper versions of the external libraries they depend upon). For these reasons, we use the Androtest suite of apps [68], which includes 68 Android apps from 18 Google Play categories. These apps have been previously used to study the design and implementation of automated testing tools for Android and met the three above listed constraints. The mutation testing tools exhibited issues in 13 of the considered 68 apps, *i.e.,* the 13 apps did not compile after injecting the faults. Thus, in the end, we considered 55 subject apps in our study. The list of considered apps as well as their source code is available in our replication package [1].

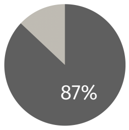
Note that while Major and PIT are compatible with Java applications, they cannot be directly applied to Android apps. Thus, we wrote specific wrapper programs to perform the mutation, the assembly of files, and the compilation of the mutated apps into runnable Android application packages (*i.e., APKs*). While the procedure used to generate and compile mutants varies for each tool, the following general workflow was used in our study: (i) generate mutants by operating on the original source/byte/smali code using all possible mutation operators; (ii) compile or assemble the APKs either using the `ant`, `dex2jar`, or `baksmali` tools; (iii) run all of the apps in a parallel-testing architecture that utilizes Android Virtual Devices (AVDs); (iv) collect data about the number of apps that crash on launch and the corresponding exceptions of these crashes which will be utilized for a manual qualitative analysis. We refer readers to our replication package for the complete technical methodology used for each mutation tool [1].

To quantitatively assess the applicability and effectiveness of the considered mutation tools to Android apps, we used three metrics: **Total Number of Generated Mutants (TNGM)**, **Stillborn Mutants (SM)**, and **Trivial Mutants (TM)**. In this paper, we consider *stillborn mutants* as those that are syntactically incorrect to the point that the APK file cannot be compiled/assembled, and *trivial mutants* as those that are killed arbitrarily by nearly any test case. If a mutant crashes upon launch, we consider it as a trivial mutant. Another metric one might consider to evaluate the effectiveness of a mutation testing tool is the number of equivalent and redundant mutants the tool produces. However, in past work, the identification of equivalent mutants has been proven to be an undecidable problem [58], and both equivalent and redundant mutants require the existence of test suites (not available for the Androtest apps). Therefore, this aspect is not studied in our work.

After generating the mutants' `APKs` using each tool, we needed a viable methodology for launching all these mutants in a reasonable amount of time to determine the number of *trivial mutants.* To accomplish this, we relied on a parallel Android execution architecture that we call the Execution Engine (EE). EE utilizes concurrently running instances of Android Virtual Devices based on the `android-x86` project [23]. Specifically, we configured 20 AVDs with the `android-x86` v4.4.2 image, a screen resolution of 1900x1200, and 1GB of RAM to resemble the hardware configuration of a Google Nexus 7 device. We then concurrently instantiated these AVDs and launched each mutant, identifying app crashes.

[2] With "cover" we mean the ability to generate a mutant simulating the presence of a give type of bug.

Bug taxonomy coverage



103 out of 119 bug types (87%) covered by the bug taxonomy in Figure 1.

392 out of the 413 tagged bug instances are covered by one of the bug types in Figure 1 (95%).

Mutation tools coverage

60 out of the 119 bug types (50%) are not covered by any of the considered mutation tools.

Bug type coverage	
Major	13%
PIT	13%
MuDroid	8%
MuJava	9%
javaLanche	10%
Deng et al.	15%
MDroid+	***38%***
Union of all tools but MDroid+	24%
All tools *U* MDroid+	**50%**

Bug count coverage	
Major	35%
PIT	33%
MuDroid	26%
MuJava	32%
javaLanche	31%
Deng et al.	41%
MDroid+	***62%***
Union of all tools but MDroid+	54%
All tools *U* MDroid+	**73%**

Figure 2: Mutation tools and coverage of analyzed bugs.

5.2 Results

RQ$_1$: Figure 2 reports (i) the percentage of bug types identified during our manual tagging that are covered by the taxonomy of bugs we previously presented in Figure 1 (top part of Figure 2), and (ii) the coverage in terms of bug types as well as of instances of tagged bugs ensured by each of the considered mutation tools (bottom part). The data shown in Figure 2 refers to the 413 bug instances for which we were able to define the exact reason behind the bug (this excludes the 114 entities tagged as *unclear* and the 199 identified as *false positives*).

87% of the bug types are covered in our taxonomy. In particular, we identified 16 new categories of bugs that we did not encounter before in the definition of our taxonomy (Section 3). Examples of these categories (full list in our replication package) are: *Issues with audio codecs*, *Improper implementation of sensors as Activities*, and *Improper usage of the static modifier*. Note that these categories just represent a minority of the bugs we analyzed, accounting all together for a total of 21 bugs (5% of the 413 bugs considered). Thus, our bug taxonomy covers 95% of the bug instances we found, indicating a very good coverage.

Moving to the bottom part of Figure 2, our first important finding highlights the limitations of the experimented mutation tools (including MDroid+) in potentially unveiling the bugs subject of our study. Indeed, for 60 out of the 119 bug types (50%), none of the considered tools is able to generate mutants simulating the bug. This stresses the need for new and more powerful mutation tools tailored for mobile platforms. For instance, no tool is currently able to generate mutants covering the *Bug in webViewClient listener* and the *Components with wrong dimensions* bug types.

When comparing the seven mutation tools considered in our study, MDroid+ clearly stands out as the tool ensuring the highest coverage both in terms of bug types and bug instances. In particular, mutators generated by MDroid+ have the potential to unveil 38% of the bug types and 62% of the bug instances. In comparison, the best competitive tool (*i.e.,* the catalog of mutants proposed by Deng *et al.* [18]) covers 15% of the bug types (61% less as compared to MDroid+) and 41% of the bug instances (34% less as compared to MDroid+). Also, we observe that MDroid+ covers bug categories (and, as a consequence, bug instances) missed by all competitive tools. Indeed, while the union of the six competitive tools covers 24% of the bug types (54% of the bug instances), adding the mutation operators included in MDroid+ increases the percentage of covered bug types to 50% (73% of the bug instances). Examples of categories covered by MDroid+ and not by the competitive tools are: *Android app permissions*, thanks to the MissingPermissionManifest operator, and the *FindViewById returns null*, thanks to the FindViewByIdReturnsNull operator.

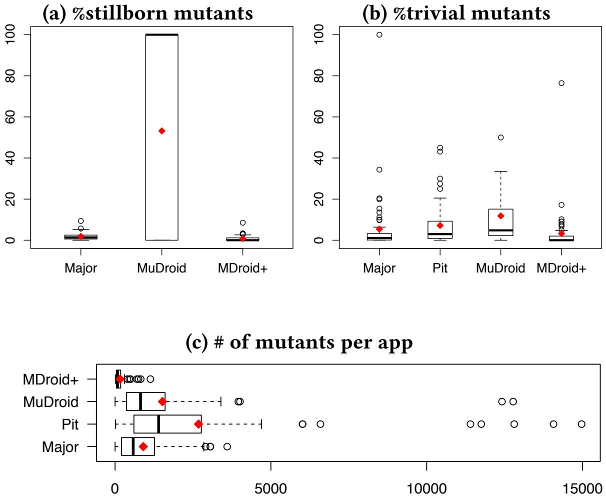


Figure 3: Stillborn and trivial mutants generated per app.

Finally, we statistically compared the proportion of bug types and the number of bug instances covered by MDroid+, by all other techniques, and by their combination, using Fisher's exact test and Odds Ratio (OR) [70]. The results indicate that:

(1) The odds of covering bug types using MDroid+ are 1.56 times greater than other techniques, although the difference is not statistically significant (*p*-value=0.11). Similarly, the odds of discovering faults with MDroid+ are 1.15 times greater than other techniques, but the difference is not significant (*p*-value=0.25);
(2) The odds of covering bug types using MDroid+ combined with other techniques are 2.0 times greater than the other techniques alone, with a statistically significant difference (*p*-value=0.008). Similarly, the odds of discovering bugs using the combination of MDroid+ and other techniques are 1.35 times greater than other techniques alone, with a significant difference (*p*-value=0.008).

RQ$_2$: Figure 3 depicts the achieved results as percentage of (a) Stillborn Mutants (SM), and (b) Trivial Mutants (TM) generated by each tool on each app. On average, 167, 904, 2.6k+, and 1.5k+ mutants were generated by MDroid+, Major, PIT, and muDroid, respectively for each app. The larger number of mutants generated by PIT is due in part to the larger number of mutation operators available for the tool. The average percentage of **stillborn mutants** (SM) generated by MDroid+, Major and muDroid over all the apps is 0.56%, 1.8%, and 53.9%, respectively, while no SM are generated by PIT (Figure 3a). MDroid+ produces significantly less

SM than Major (Wilcoxon paired signed rank test p-value< 0.001 – adjusted with Holm's correction [29], Cliff's d=0.59 - large) and than muDroid (adjusted p-value< 0.001, Cliff's d=0.35 - medium).

These differences across the tools are mainly due to the compilation/assembly process they adopt during the mutation process. PIT works at Java bytecode level and thus can avoid the SM problem, at the risk of creating a larger number of TM. However, PIT is the tool that required the highest effort to build a wrapper to make it compatible with Android apps. Major works at the source code level and compiles the app in a "traditional" manner. Thus, it is prone to SM and requires an overhead in terms of memory and CPU resources needed for generating the mutants. Finally, muDroid operates on `APK`s and `smali` code, reducing the computational cost of mutant generation, but significantly increasing the chances of SM.

All four tools generated **trivial mutants** (TM) (*i.e.,* mutants that crashed simply upon launching the app). These instances place an unnecessary burden on the developer, particularly in the context of mobile apps, as they must be discarded from analysis. The mean of the distribution of the percentage of TM over all apps for MDroid+, Major, PIT and muDroid is 2.42%, 5.4%, 7.2%, and 11.8%, respectively (Figure 3b). MDroid+ generates significantly less TM than muDroid (Wilcoxon paired signed rank test adjusted p-value=0.04, Cliff's d=0.61 - large) and than PIT (adjusted p-value=0.004, Cliff's d=0.49 - large), while there is no statistically significant difference with Major (adjusted p-value=0.11).

While these percentages may appear small, the raw values show that the TM can comprise a large set of instances for tools that can generate thousands of mutants per app. For example, for the Translate app, 518 out of the 1,877 mutants generated by PIT were TM. For the same app, muDroid creates 348 TM out of the 1,038 it generates. For the Blokish app, 340 out of the 3,479 mutants generated by Major were TM. Conversely, while MDroid+ may generate a smaller number of mutants per app, this also leads to a smaller number of TM, only 213 in total across all apps. This is due to the fact that MDroid+ generates a much smaller set of mutants that are specifically targeted towards emulating *real* faults identified in our empirically derived taxonomy, and are applied on specific locations detected by the PFP.

RQ$_3$: In terms of mutation operators causing the highest number of stillborn and TM we found that for Major, the Literal Value Replacement (LVR) operator had the highest number of TM, whereas the Relational Operator Replacement (ROR) had the highest number of SM. It may seem surprising that ROR generated many SM, however, we discovered that the reason was due to improper modifications of loop conditions. For instance, in the A2dp.Vol app one mutant changed this loop: `for (int i = 0; i < cols; i++)` and replaced the condition "$i < cols$" with "`false`", causing the compiler to throw an unreachable code error. For PIT, the Member Variable Mutator (MVM) is the one causing most of the TM; for muDroid, the Unary Operator Insertion (UOI) operator has the highest number of SM (although all the operators have relatively high failure rates), and the Relational Value Replacement (RVR) has the highest number of TM. For MDroid+, the WrongStringResource operator had that highest number of SM, whereas the FindViewByIdReturnsNull operator had the highest number of TM.

Table 2: Number of Generated, Stillborn, and Trivial Mutants created by MDroid+ operators.

Mutation Operators	**GM**	**SM**	**TM**
WrongStringResource	3394	0	14
NullIntent	559	3	41
InvalidKeyIntentPutExtra	459	3	11
NullValueIntentPutExtra	459	0	14
InvalidIDFindView	456	4	30
FindViewByIdReturnsNull	413	0	40
ActivityNotDefined	384	1	8
InvalidActivityName	382	0	10
DifferentActivityIntentDefinition	358	2	8
ViewComponentNotVisible	347	5	7
MissingPermissionManifest	229	0	8
InvalidFilePath	220	0	1
InvalidLabel	214	0	3
ClosingNullCursor	179	13	5
LengthyGUICreation	129	0	1
BuggyGUIListener	122	0	2
LengthyGUIListener	122	0	0
SDKVersion	66	0	2
NullInputStream	61	0	4
WrongMainActivity	56	0	0
InvalidColor	52	0	0
NullOuptutStream	45	0	2
InvalidDate	40	0	0
InvalidSQLQuery	33	0	2
NotSerializable	15	7	0
NullBluetoothAdapter	9	0	0
LengthyBackEndService	8	0	0
NullBackEndServiceReturn	8	1	0
NotParcelable	7	6	0
InvalidIndexQueryParameter	7	1	0
OOMLargeImage	7	4	0
BluetoothAdapterAlwaysEnabled	4	0	0
InvalidURI	2	0	0
NullGPSLocation	1	0	0
LongConnectionTimeOut	0	0	0
Total	8847	50	213

To qualitatively investigate the causes behind the crashes, three authors manually analyzed a randomly selected sample of 15 crashed mutants per tool. In this analysis, the authors relied on information about the mutation (*i.e.,* applied mutation operator and location), and the generated stack trace.

Major. The reasons behind the crashing mutants generated by Major mainly fall in two categories. First, mutants generated with the LVR operator that changes the value of a literal causing an app to crash. This was the case for the *wikipedia* app when changing the "1" in the invocation `setCacheMode(params.getString(1))` to "0". This passed a wrong asset URL to the method `setCacheMode`, thus crashing the app. Second, the Statement Deletion (STD) operator was responsible for app crashes especially when it deleted needed methods' invocations. A representative example is the deletion of invocations to methods of the superclass when overriding methods, *e.g.,* when removing the `super.onDestroy()` invocation from the `onDestroy()` method of an `Activity`. This results in throwing of an `android.util.SuperNotCalledException`. Other STD mutations causing crashes involved deleting a statement initializing the main `Activity` leading to a `NullPointerException`.

muDroid. This tool is the one exhibiting the highest percentage of stillborn and TM. The most interesting finding of our qualitative analysis is that 75% of the crashing mutants lead to the throwing of a `java.lang.VerifyError`. A `VerifyError` occurs when Android tries to load a class that, while being syntactically correct, refers to resources that are not available (*e.g.,* wrong class paths). In the remaining 25% of the cases, several of the crashes were due to the Inline Constant Replacement (ICR) operator. An example is the crash observed in the `photostream` app where the "100" value has been replaced with "101" in `bitmap.compress(Bitmap.CompressFormat.PNG, 100, out)`. Since "100" represents the quality of the compression, its value must be bounded between 0 and 100.

PIT. In this tool, several of the manually analyzed crashes were due to (i) the RVR operator changing the return value of a method to null, causing a `NullPointerException`, and (ii) removed method invocations causing issues similar to the ones described for Major.

MDroid+. Table 2 lists the mutants generated by MDroid+ across all the systems (information for the other tools is provided with our replication package). The overall rate of SM is very low in MDroid+, and most failed compilations pertain to edge cases that would require a more robust static analysis approach to resolve. For example, the ClosingNullCursor operator has the highest total number of SM (across all the apps) with 13, and some edge cases that trigger compilation errors involve cursors that have been declared `Final`, thus causing the reassignment to trigger the compilation error. The small number of other SM are generally other edge cases, and current limitations of MDroid+ can be found in our replication package with detailed documentation.

The three operators generating the highest number of TM are NullIntent(41), FindViewByIdReturnsNull(40), and InvalidIDFindView(30). The main reason for the NullIntent TM are intents invoked by the Main Activity of an app (*i.e.,* the activity loaded when the app starts). Intents are one of the fundamental components of Android apps and function as asynchronous messengers that activate Activities, Broadcast Receivers and services. One example of a trivial mutant is for the A2dp.Vol app, in which a bluetooth service, inteneded to start up when the app is launched, causes a NullPointerException when opened due to NullIntent operator. To avoid cases like this, more sophisticated static analysis could be performed to prevent mutations from affecting Intents in an app's MainActivity. The story is similar for the FindViewView-ByIdReturnsNull and InvalidIDFindView operators: TM will occur when views in the MainActivity of the app are set to null or reference invalid Ids, causing a crash on startup. Future improvements to the tool could avoid mutants to be seeded in components related to the MainActivity. Also, it would be desirable to allow developers to choose the activities in which mutations should be injected.

Summary of the RQs. *MDroid+ outperformed the other six mutation tools by achieving the highest coverage both in terms of bug types and bug instances. However, the results show that Android-specific mutation operators should be combined with classic operators to generate mutants that are representative of real faults in mobile apps (**RQ**$_1$). MDroid+ generated the smallest rate of both stillborn and trivial mutants illustrating its immediate applicability to Android apps. Major and muDroid generate stillborn mutants, with the latter having a critical average rate of 58.7% stillborn mutants per app (**RQ**$_2$). All four tools generated a relatively low rate of trivial mutants, with muDroid again being the worst with an 11.8% average rate of trivial mutants (**RQ**$_3$). Our analysis shows that the PIT tool is most applicable to Android apps when evaluated in terms of the ratio between failed and generated mutants. However, MDroid+ is both practical and based on Android-specific operations implemented according to an empirically derived fault-taxonomy of Android apps.*

6 THREATS TO VALIDITY

This section discusses the threats to validity of the work related to devising the fault taxonomy, and carrying out the study reported in Section 5.

Threats to *construct validity* concern the relationship between theory and observation. The main threat is related to how we assess and compare the performance of mutation tools, *i.e.,* by covering the types, and by their capability to limit stillborn and trivial mutants. A further, even more relevant evaluation would explore the extent to which different mutant taxonomies are able to support test case prioritization. However, this requires a more complex setting which we leave for our future work.

Threats to *internal validity* concern factors internal to our settings that could have influenced our results. This is, in particular, related to possible subjectiveness of mistakes in the tagging of Section 3 and for **RQ**$_1$. As explained, we employed multiple taggers to mitigate such a threat.

Threats to *external validity* concern the generalizability of our findings. To maximize the generalizability of the fault taxonomy, we have considered six different data sources. However, it is still possible that we could have missed some fault types available in sources we did not consider, or due to our sampling methodology. Also, we are aware that in our study results of **RQ**$_1$ are based on the new sample of data sources, and results of **RQ**$_2$ on the set of 68 apps considered [68].

7 CONCLUSIONS

Although Android apps rely on the Java language as a programming platform, they have specific elements that make the testing process different than other Java applications. In particular, the type and distribution of faults exhibited by Android apps may be very peculiar, requiring, in the context of mutation analysis, specific operators.

In this paper, we presented the first taxonomy of faults in Android apps, based on a manual analysis of 2,023 software artifacts from six different sources. The taxonomy is composed of 14 categories containing 262 types. Then, based on the taxonomy, we have defined a set of 38 Android-specific mutation operators, implemented in an infrastructure called MDroid+, to automatically seed mutations in Android apps. To validate the taxonomy and MDroid+, we conducted a comparative study with Java mutation tools. The study results show that MDroid+ operators are more representative of Android faults than other catalogs of mutation operators, including both Java and Android-specific operators previously proposed. Also MDroid+ is able to outperform state-of-the-art tools in terms of stillborn and trivial mutants.

The obtained results make our taxonomy and MDroid+ ready to be used and possibly extended by other researchers/practitioners. To this aim, MDroid+ and the wrappers for using Major and Pit with Android apps are available as open source projects [1]. Future work will extend MDroid+ by implementing more operators, and creating a framework for mutation analysiss. Also, we plan to experiment with MDroid+ in the context of test case prioritization.

ACKNOWLEDGMENTS

Bavota was supported in part by the SNF project JITRA, No. 172479.

REFERENCES

[1] 2017. Online appendix/Replication package for "Enabling Mutation Testing for Android Apps". http://android-mutation.com/fse-appendix. (2017).

[2] 2017. Perfecto. http://www.perfectomobile.com. (2017).

[3] Konstantinos Adamopoulos, Mark Harman, and Robert M. Hierons. 2004. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. In *Genetic and Evolutionary Computation - GECCO 2004, Genetic and Evolutionary Computation Conference, Seattle, WA, USA, June 26-30, 2004, Proceedings, Part II.* 1338–1349.

[4] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic Execution of Android Test Suites in Adverse Conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis.* 83–93.

[5] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012.* 258–261.

[6] James H. Andrews, Lionel C. Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA.* 402–411.

[7] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Trans. Software Eng.* 32, 8 (2006), 608–624.

[8] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. 2014. Automated testing for SQL injection vulnerabilities: an input mutation approach. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014.* 259–269.

[9] Apple. 2017. App Store. https://itunes.apple.com/us/genre/ios/id36?mt=8. (2017).

[10] S. Beyer and M. Pinzger. 2014. A Manual Categorization of Android App Development Issues on Stack Overflow. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME'14).* 531–535.

[11] Ning Chen, Steven C.H. Hoi, Shaohua Li, and Xiaokui Xiao. 2015. SimApp: A Framework for Detecting Similar Mobile Applications by Online Kernel Learning. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining (WSDM '15).* ACM, 305–314.

[12] N. Chen, J. Lin, S. Hoi, X. Xiao, and B. Zhang. 2014. AR-Miner: Mining Informative Reviews for Developers from Mobile App Marketplace. In *36th International Conference on Software Engineering (ICSE'14).*

[13] Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. 2015. Unveiling Exception Handling Bug Hazards in Android Based on GitHub and Google Code Issues. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15).* 134–145.

[14] Henry Coles. 2017. Mutation testing systems for Java compared. (2017). http://pitest.org/java_mutation_testing_systems/.

[15] Henry Coles. 2017. PIT. http://pitest.org/. (2017).

[16] Muriel Daran and Pascale Thévenod-Fosse. 1996. Software Error Analysis: A Real Case Study Involving Real Faults and Mutations. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis, ISSTA 1996, San Diego, CA, USA, January 8-10, 1996.* 158–171.

[17] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41.

[18] Lin Deng, N. Mirzaei, P. Ammann, and J. Offutt. 2015. Towards mutation analysis of Android apps. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on.* 1–10.

[19] Anna Derezińska and Konrad Hałas. 2014. *Analysis of Mutation Operators for the Python Language.* Springer International Publishing, Cham, 155–164.

[20] Daniel Di Nardo, Fabrizio Pastore, and Lionel C. Briand. 2015. Generating Complex and Faulty Test Data through Model-Based Mutation Analysis. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015.* 1–10.

[21] Michael Fischer, Martin Pinzger, and Harald Gall. 2003. Populating a Release History Database from Version Control and Bug Tracking Systems. In *19th International Conference on Software Maintenance (ICSM 2003), 22-26 September 2003, Amsterdam, The Netherlands.* 23–.

[22] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. 2013. RERAN: Timing- and Touch-Sensitive Record and Replay for Android. In *Proceedings of the International Conference on Software Engineering, ICSE'13.* 72–81.

[23] Google. 2017. Android x86 project. http://www.android-x86.org. (2017).

[24] Google. 2017. Google Play. https://play.google.com/store. (2017).

[25] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, , and Alex Groce. 2016. Measuring Effectiveness of Mutant Sets. In *IEEE International Conference on Software Testing, Verification and Validation - Workshops, ICSTW 2016.* 132–141.

[26] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2016. On the Limits of Mutation Reduction Strategies. In *IEEE/ACM International Conference on Software Engineering, ICSE'16.* 511–522.

[27] R. G. Hamlet. 1977. Testing Programs with the Aid of a Compiler. *IEEE Trans. Software Eng.* 3, 4 (July 1977), 279–290.

[28] S. Hao, B. Liu, S. Nath, W.G.J. Halfond, and R. Govindan. 2014. PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps. In *MobiSys'14.* 204–217.

[29] S. Holm. 1979. A Simple Sequentially Rejective Bonferroni Test Procedure. *Scandinavian Journal on Statistics* 6 (1979), 65–70.

[30] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile Yet Lightweight Record-and-replay for Android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015).* ACM, New York, NY, USA, 349–366.

[31] C. Iacob and R. Harrison. 2013. Retrieving and analyzing mobile apps feature requests from online reviews. In *10th Working Conference on Mining Software Repositories (MSR'13).* 41–44.

[32] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sept 2011), 649–678.

[33] M. Erfani Joorabchi, A. Mesbah, and P. Kruchten. 2013. Real Challenges in Mobile Apps. In *International Conference on Empirical Software Engineering and Measurement, ESEM'13.*

[34] René Just. 2014. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. In *International Symposium on Software Testing and Analysis, ISSTA'14.* 4.

[35] René Just, Michael D. Ernst, and Gordon Fraser. 2014. Efficient mutation analysis by propagating and partitioning infected execution states. In *International Symposium on Software Testing and Analysis, ISSTA'14.*

[36] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014.* 654–665.

[37] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. 2012. Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis?. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012.* 720–725.

[38] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011.* 612–615.

[39] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E. Hassan. 2015. What Do Mobile App Users Complain About? *IEEE Software* 32, 3 (2015), 70–77.

[40] Sun-Woo Kim, John A. Clark, and John A. McDermid. 2001. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Softw. Test., Verif. Reliab.* 11, 3 (2001), 207–225.

[41] Chieh-Jan Mike Liang, Nicholas D. Lane, Niels Brouwers, Li Zhang, Börje F. Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, and Feng Zhao. 2014. Caiipa: Automated Large-scale Mobile App Testing Through Contextual Fuzzing. In *20th Annual International Conference on Mobile Computing and Networking (MobiCom '14).* 519–530.

[42] M. Linares-Vásquez, B. Dit, and D. Poshyvanyk. 2013. An exploratory analysis of mobile development issues using stack overflow. In *2013 10th IEEE Working Conference on Mining Software Repositories (MSR'13).* 93–96.

[43] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing. In *ICSME'17.* to appear.

[44] Yu-Seung Ma, Yong Rae Kwon, and Jeff Offutt. 2002. Inter-Class Mutation Operators for Java. In *13th International Symposium on Software Reliability Engineering (ISSRE 2002), 12-15 November 2002, Annapolis, MD, USA.* 352–366.

[45] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: An Automated Class Mutation System. *Softw. Test., Verif. Reliab.* 15, 2 (June 2005), 97–133.

[46] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013.* 224–234.

[47] L. Madeyski and N. Radyk. 2010. Judy - a mutation testing tool for Java. *IET Software* 4, 1 (Feb 2010), 32–42.

[48] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014.* 599–609.

[49] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *ISSTA'16.* ACM, New York, NY, USA, 94–105.

[50] Matthew B. Miles, A. Michael Huberman, and Johnny Saldaña. 2013. *Qualitative Data Analysis: A Methods Sourcebook* (3rd ed.). SAGE Publications, Inc.

[51] K. H. Moller and D. J. Paulish. 1993. An empirical investigation of software fault distribution. In *Software Metrics Symposium, 1993. Proceedings., First International.* 82–90.

[52] Ivan Moore. 2017. Jester - the JUnit test tester. (2017). http://goo.gl/cQZ0L1.

[53] Kevin Moran, Mario Linares Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Auto-completing bug reports for Android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015.* 673–686.

[54] Kevin Moran, Mario Linares Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016.* 33–44.

[55] M. Nagappan and E. Shihab. 2016. Future Trends in Software Engineering Research for Mobile Apps. In *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16).*

[56] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, Reporting, and Fixing Performance Bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13).* IEEE Press, Piscataway, NJ, USA, 237–246.

[57] A. Jefferson Offutt and Stephen D. Lee. 1991. How Strong is Weak Mutation?. In *Proceedings of the Symposium on Testing, Analysis, and Verification, TAV 1991, Victoria, British Columbia, Canada, October 8-10, 1991.* 200–213.

[58] A. Jefferson Offutt and Jie Pan. 1997. Automatically detecting equivalent mutants and infeasible paths. *Softw. Test., Verif. Reliab.* 7, 3 (1997), 165–192.

[59] R. A. P. Oliveira, E. Alégroth, Z. Gao, and A. Memon. 2015. Definition and evaluation of mutation operators for GUI-level mutation analysis. In *International Conference on Software Testing, Verification, and Validation - Workshops, ICSTW'15.* 1–10.

[60] Thomas J. Ostrand and Elaine J. Weyuker. 2002. The Distribution of Faults in a Large Industrial Software System. *SIGSOFT Softw. Eng. Notes* 27, 4 (July 2002), 55–64.

[61] D. Pagano and W. Maalej. 2013. User feedback in the appstore: An empirical study. In *Requirement Engineering Conference, RE'13.*

[62] F. Palomba, M. Linares-Vasquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. 2015. User reviews matter! Tracking crowdsourced reviews to support evolution of successful apps. In *IEEE International Conference on Software Maintenance and Evolution, ICSME'15.* 291–300.

[63] S. Panichella, A. Di Sorbo, E. Guzman, C.A. Visaggio, G. Canfora, and H.C. Gall. 2015. How can i improve my app? Classifying user reviews for software maintenance and evolution. In *IEEE International Conference on Software Maintenance and Evolution, ICSME'15.* 281–290.

[64] Upsorn Praphamontripong, Jeff Offutt, Lin Deng, and Jingjing Gu. 2016. An Experimental Evaluation of Web Mutation Operators. In *International Conference on Software Testing, Verification, and Validation, ICSTW'16.* 102–111.

[65] L. Ravindranath, S. nath, J. Padhye, and H. Balakrishnan. [n. d.]. Automatic and Scalable Fault Detection for Mobile Applications. In *MobiSys'14.*

[66] B. Robinson and P. Brooks. 2009. An Initial Study of Customer-Reported GUI Defects. In *ICSTW '09. International Conference on Software Testing, Verification and Validation Workshops 2009.* 267–274.

[67] Christoffer Rosen and Emad Shihab. 2016. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering* 21, 3 (2016), 1192–1223.

[68] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015.* 429–440.

[69] David Schuler and Andreas Zeller. 2009. Javalanche: efficient mutation testing for Java. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009.* 297–298.

[70] D.DJ. Sheskin. 2000. *Handbook of Parametric and Nonparametric Statistical Procedures.* (second edition ed.). Chapman & Hall/CRC.

[71] Donghwan Shin and Doo-Hwan Bae. 2016. A Theoretical Framework for Understanding Mutation-Based Testing Methods. In *International Conference on Software Testing, Verification, and Validation, ICST'16.*

[72] Reel Two. [n. d.]. Jumble. ([n. d.]). http://jumble.sourceforge.net.

[73] Mario Linares Vásquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. 2015. How developers detect and fix performance bottlenecks in Android apps. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015.* 352–361.

[74] Mario Linares Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2015. Mining Android App Usages for Generating Actionable GUI-Based Execution Scenarios. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015.* 111–122.

[75] Lorenzo Villarroel, Gabriele Bavota, Barbara Russo, Rocco Oliveto, and Massimiliano Di Penta. 2016. Release Planning of Mobile Apps Based on User Reviews. In *38th International Conference on Software Engineering (ICSE'2016).*

[76] Yuan-W. 2017. muDroid project at GitHub. (2017). https://goo.gl/sQo6EL.

[77] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *International Conference on Software Testing, Verification, and Validation, ICST 2014.* 183–192.

[78] Hongyu Zhang. 2008. On the Distribution of Software Faults. *IEEE Trans. Software Eng.* 34, 2 (2008), 301–302.

[79] Pingyu Zhang and Sebastian Elbaum. 2014. Amplifying Tests to Validate Exception Handling Code: An Extended Study in the Mobile Application Domain. *ACM Trans. Softw. Eng. Methodol.* 23, 4, Article 32 (Sept. 2014), 32:1–32:28 pages.

[80] Chixiang Zhou and Phyllis G. Frankl. 2009. Mutation Testing for Java Database Applications. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009.* 396–405.