Enabling Mutation Testing for Android Apps

Mario Linares-Vasquez

Gabriele Bavota

Michele Tufano
*College of William and Mary*

Kevin Moran
*College of William and Mary*, kpmoran@cs.wm.edu

Massimiliano Di Penta

*See next page for additional authors*

Follow this and additional works at: [https://scholarworks.wm.edu/aspubs](https://scholarworks.wm.edu/aspubs)

**Recommended Citation**

Linares-Vasquez, Mario; Bavota, Gabriele; Tufano, Michele; Moran, Kevin; Di Penta, Massimiliano; Vendome, Christopher; Bernal-Cardenas, Carlos; and Poshyvanyk, Denys, Enabling Mutation Testing for Android Apps (2017). 10.1145/3106237.3106275

This Other is brought to you for free and open access by the Arts and Sciences at W&M ScholarWorks. It has been accepted for inclusion in Arts & Sciences Articles by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.
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 known mutation tools.

CCS CONCEPTS
- Software and its engineering → Software verification and validation;

KEYWORDS
Mutation testing, Fault taxonomy, Operators, Android

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.

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 assess 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 assess 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
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 apps, (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 testing1 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.

---

1The 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.
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.

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

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 simulates 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 to 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:

- **RQ1**: Are the mutation operators (available for Java and Android apps) representative of real bugs in Android apps?
- **RQ2**: 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?
Web app—details in Section 3.1). We targeted the tagging of Android-specific bytecode representation.

RQ1: What are the major causes for stillborn and trivial mutants produced by the mutation testing tools when applied to Android apps?

To answer RQ1, 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 RQ2, 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 RQ3, 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 RQ1, we analyzed the complete list of 102 mutation operators from the seven considered tools to investigate their ability to “cover” bugs described in 726 artifacts2 (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 RQ1 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 RQ2 and RQ3, 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.
5.2 Results

RQ1: 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

![Figure 2: Mutation tools and coverage of analyzed bugs.](image)

![Figure 3: Stillborn and trivial mutants generated per app.](image)
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 APKs and small1 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.

RQ3: 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, whereas the FindViewByIdByldReturnsNull operator had the highest number of TM. It is due to the fact that PIT requires 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 generates a much smaller set of TM, whereas the FindViewByIdByldReturnsNull operator had the highest number of TM.

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(\( \text{param.getString}(1) \)) 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 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

<table>
<thead>
<tr>
<th>Mutation Operators</th>
<th>CM</th>
<th>SM</th>
<th>TM</th>
</tr>
</thead>
<tbody>
<tr>
<td>WrongStringResource</td>
<td>3394</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>NullIntent</td>
<td>559</td>
<td>3</td>
<td>41</td>
</tr>
<tr>
<td>InvalidKeyIntentPutExtra</td>
<td>459</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>NullValueIntentPutExtra</td>
<td>459</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>InvalidFindByldReturnValue</td>
<td>456</td>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>ActivityNotDefined</td>
<td>384</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>InvalidActivityName</td>
<td>838</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>DifferentActivityIntentDefinition</td>
<td>358</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>ViewComponentNotVisible</td>
<td>347</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>MissingPermissionManifest</td>
<td>229</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Invalid RipplePath</td>
<td>220</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>invalidLabel</td>
<td>214</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>ClosingNullCursor</td>
<td>179</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>LengthyGUILabelCreation</td>
<td>129</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>BuggyGUIListener</td>
<td>122</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>MDVersion</td>
<td>122</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>NullOutputStream</td>
<td>64</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>WrongMainActivity</td>
<td>56</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Invalid.color</td>
<td>52</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NullOutputStream</td>
<td>45</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>InvalidDate</td>
<td>40</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>InvalidSQLQuery</td>
<td>33</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>NotSerializable</td>
<td>15</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>NullBluetoothAdapter</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LengthyBackEndService</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NullBackEndServiceReturn</td>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NotParcelable</td>
<td>7</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>InvalidIndexQueryParameter</td>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>OOMUnloadImage</td>
<td>7</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>BluetoothAdapterAlwaysEnabled</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>InvalidUrl</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NullGPSLocation</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LongConnectionTimeOut</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>8847</td>
<td>50</td>
<td>213</td>
</tr>
</tbody>
</table>
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 photo stream app where the “100” value has been replaced with “101” in bitmap.compress(Bitmap.Compress-Format.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 InvalidDFindView(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 A2dpVol app, in which a bluetooth service, intended 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 findViewByIdViewByIdReturnsNull and InvalidDFindView 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 (RQ1). MDroid+ generated the smallest rate of both stillborn and trivial mutants illustrating its immediate applicability to Android apps. Major and madDroid generate stillborn mutants, with the latter having a critical average rate of 58.7% stillborn mutants per app (RQ2). All four tools generated a relatively low rate of trivial mutants, with madDroid again being the worst with an 11.8% average rate of trivial mutants (RQ3). 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 mutation 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 RQ1. 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 RQ2 are based on the new sample of data sources, and results of RQ3 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 analysis. 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.
Enabling Mutation Testing for Android Apps

ESEC/FSE’17, September 4–8, 2017, Paderborn, Germany

REFERENCES


