Enabling Mutant Generation for Open- and Closed-Source Android Apps

Camilo Escobar-Velásquez, *Student Member, IEEE,* Mario Linares-Vásquez, *Member, IEEE,* Gabriele Bavota, *Member, IEEE,* Michele Tufano *Member, IEEE,* Kevin Moran, *Member, IEEE,* Massimiliano Di Penta, *Member, IEEE,* Christopher Vendome, *Member, IEEE,* Carlos Bernal-Cárdenas, *Student Member, IEEE,* and Denys Poshyvanyk, *Member, IEEE,*

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. The case for Android apps is not an exception. Therefore, in this paper we describe the process we followed to create (i) a taxonomy of mutation operations and, (ii) two tools, `MDroid+` and `MutAPK` for mutant generation of Android apps. To this end, 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 them in two tools, the first enabling mutant generation at the source code level, and the second designed to perform mutations at APK level. The rationale for having a dual-approach is based on the fact that source code is not always available when conducting mutation testing. Thus, mutation testing for APKs enables new scenarios in which researchers/practitioners only have access to APK files. The taxonomy, proposed operators, and tools have been evaluated in terms of the number of non-compilable, trivial, equivalent, and duplicate mutants generated and their capacity to represent real faults in Android apps as compared to other well-known mutation tools.

Index Terms—Mutation Testing, Fault taxonomy, Mutation Operators, Android

1 Introduction

Mobile apps play a paramount role in our daily lives. With millions of mobile apps available for download on Google Play [1] and the Apple App Store [2], 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 health monitoring. Given the increasing relevance and demand for high-quality apps, industrial practitioners and academic researchers have been devoting significant effort to improve 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 [3], [4], to automated approaches that generate and execute test cases [5], [6], [7], [8], rippers that systematically explore the apps GUI [9], [10], [11], and cloud-based services for large-scale multi-device testing [12]. Despite the availability of these tools/approaches, the field of mobile app testing is still very much under development, as highlighted by limitations of test data generation approaches [6], [13], and concerns regarding the effective assessment of the quality of mobile apps' test suites.

One way to evaluate test suites is to seed small faults, called mutations, into source code and assess the ability of a suite to detect these faults [14], [15]. Such mutations have been defined in the literature to reflect the typical errors developers make when writing source code [16], [17], [18], [19], [20], [21], [22]. Indeed, the extent to which mutants reflect typical bugs for a given application/domain can have an impact on the extent to which mutants can replace real bugs in software testing, *e.g.,* to evaluate a test suite effectiveness [23], or even prioritize bugs [24].

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 [16], [17] is sufficient to assess test quality and drive test case generation/selection for mobile apps.

In addition, mutation operators have been thought to be applied directly to the source code, which requires building/compiling the generated mutants. Building/compilation time is often more time consuming than injecting the mutants; in that sense, tools that avoid compilation time are desired by practitioners and

- *C. Escobar-Velásquez and M. Linares-Vásquez are with the Department of Systems and Computing Engineering, Universidad de los Andes, Bogotá, Colombia. E-mail:{ca.escobar2434, m.linaresv}@uniandes.edu.co*
- *G. Bavota is with the Faculty of Informatics, Universit'a della Svizzera Italiana, Lugano, Switzerland. Email: gabriele.bavota@usi.ch*
- *M. Tufano is with Microsoft, Redmond, WA. E-mail: michele.tufano@microsoft.com*
- *K. Moran, C. Bernal-Cárdenas and D. Poshyvanyk are with the Department of Computer Science, The College of William and Mary, Williamsburg, VA 23185. E-mail:{kpmoran, mtufano, cebernal, denys}@cs.wm.edu*
- *C. Vendome is with the Department of Computer Science & Software Engineering, Miami University, Oxford, OH 45056. E-mail:vendomcg@miamioh.edu*
- *M. Di Penta is with the Department of Engineering, University of Sannio, Benevento, Italy. Email: dipenta@unisannio.it*

Manuscript received July 20, 2019; revised October 01, 2019; revised February 21, 2020.

researchers.

Previous tools for Java applications like Jumble, PIT, and Javalanche generate the mutations directly on the bytecode, which reduces the total time required to generate executable mutants, and also avoids generating a potentially-large number of uncompilable mutants. In the case of Android apps, there is no existing approach that enables mutation testing on source code or directly on executable Android Packages (APKs). Having a tool that enables mutation testing at APK-level will help developers and practitioners to avoid building approaches/tools for each of the existing native Android languages (Java, Kotlin, or Dart), because APK-level mutation is language agnostic.

Moreover, having such an approach could make mutation testing for Android apps more suitable as in the case of outsourced/crowdsourced testing by third-parties that are not the app owners. Although mutation testing is not a service commonly offered by third parties, it could be used to evaluate the quality of test suites designed in a context of outsourced/crowdsourced testing, or to automatically generate test cases based on potential mutations. In general, APK-level mutation could help practitioners — that do not have access to source code of the analyzed apps— to enable different scenarios that are not widely explored yet by external services.

Currently, there does not exist an Android mutation testing framework capable of seeding a large set of realistic, Android-specific faults into open and closed source Android apps. In fact, relatively few Android-specific mutation operators have been proposed by the research community [25], [26] and as such these do not cover a large range of possible Android-specific bugs/faults. Mutation tools for Java apps, such as Pit [27] and Major [28], [29], lack any Android-specific mutation operator, and present challenges for their use in this context, resulting in common problems such as trivial mutants that always crash at runtime or difficulties in automating mutant compilation into APKs. To provide support for mutation testing of Android apps both when source code is available and when it is not, in this paper we explore mutation testing by following a data-driven approach that led us to build a taxonomy of real bugs in mobile apps. Then, we propose a set of specialized mutation operators, and build tools that allow mutant generation, both at source code-level and at APK-level, based on the proposed operators.

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, (iii) an analysis of the applicability of state-of-the-art mutation tools on Android apps, and (iv) a characterization of the pros and cons of conducting mutant generation at source code or APK level.

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. 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 (Fig. 1). Then, leveraging this fault taxonomy and focusing on Android-specific faults, we devised a set of 38 Android mutation operators and created their corresponding mutation rules for Java code and SMALI intermediate representation (IR).

Based on the proposed operators, we conceived and implemented two frameworks for mutant generation of Android apps, `MDroid+` and `MutAPK`. The former injects mutations at source-code level, while the latter injects the mutations directly in the APKs. Our rationale for having two different tools is based on the fact that source code is not always available (*e.g.*, as in the case of outsourced testing services). In addition, we wanted to identify the pros and cons of generating mutants of Android apps at source-code and APK levels. Both tools are publicly available [30], [31].

In addition, we conducted a study comparing `MDroid+` and `MutAPK` with other Java and Android-specific mutation tools. The study results indicate that both `MDroid+` and `MutAPK`, as compared to existing competitive tools, (i) can cover a larger number of bug types/instances present in Android apps; (ii) are highly complementary to the existing tools in terms of covered bug types; and (iii) generate fewer trivial, non-compilable, equivalent and duplicate[1] mutants. When comparing source-code vs. APK level mutation, we found that both mutation and compilation/assembling are performed quicker at APK level than at source code level, but with a lower quality of generated mutants. Our experiments show an improvement of 93.83% (*i.e.*, 4.32 from 4.61 seconds) in the mutation time and 87.05% (*i.e.*, 174.73 from 195 seconds) for compilation/assembling times. However, the source code-based mutation approach generates only 2.97% (*i.e.*, 263 of 8847 mutants) of non-compilable or trivial mutants as compared to the 6.8% (*i.e.*, 5105 of 75053 mutants) of the APK-based mutation approach. It is worth noting that our study does not conduct mutation testing (*i.e.*, executing test suites), since no test suite is available for the selected app dataset.

This paper is an extension of a previous paper published at the 11*th* Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'17) [32]. The extension includes: (i) a novel approach and publicly available tool for conducting mutant generation for Android apps at APK level; (ii) a larger study that compares existing tools with our two tools for mutation testing (`MDroid+` and `MutAPK`); and (iii) a new research question focused on the pros and cons of mutation testing for Android apps at source-code and APK levels.

Paper organization. Section 2 describes previous work on mutation testing and analysis of closed-source mobile apps. Then, Section 3 describes the process we followed to create (empirically) a taxonomy of real crashes/bugs in Android apps; the taxonomy establishes the foundations for the mutation operators at source-code and APK levels. Section 4 is devoted to describing the mutation operators and its detailed implementation at source-code and APK

1. We use the definition provided by Papadakis*et al.* [98]: two mutants that are equivalent to each other are called *duplicate mutants*

levels. In Section 5, we describe the design of the study and the corresponding results, while in Section 6 we discuss some implications of mutation testing at source-code and APK levels. In Section 7 we report the threats to the validity of the results. Finally, Section 8 outlines the conclusions.

2 Previous Work

This section describes related literature and publicly available, state-of-the-art tools and approaches on mutation testing and analysis of mobile apps. We do not discuss the literature on testing Android apps [5], [6], [7], [8], [9], [13], [33], [34], [35], since proposing a novel approach for testing Android apps is not the main goal of this work. For further details about mobile app testing we refer the reader to previous surveys and mapping studies [13], [36], [37].

2.1 Mutation Testing

Since the introduction of mutation testing in the 70s [14], [15], 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 [16] and Python [38]) but also for specific types of software like Web applications [39], NodeJS packages [40], JS applications [41] and data-intensive applications [42], [43] either to exercise their GUIs [44] or to alter complex, model-defined input data [45]. 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. For further details about the concepts, recent research, and future work in the field of mutation testing, we refer the reader to previous work by Jia and Harman [46].

To the best of our knowledge, the closest work to ours is that of Deng *et al.*, [25], 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.

Another closely-related work is that by Jabbarvand & Malek [26] which proposed a mutation analysis framework, called μDroid, aimed *specifically* at helping developers to design tests that identify energy problems in Android apps based on a set of fifty empirically derived energy anti-patterns. We view our work as complementary to this energy-aware mutation framework, as our empirically derived Android mutant taxonomy covers a wide range of both functional and non-functional bugs/faults in Android apps, sharing little overlap with the mutation operators proposed in [26]. In fact, developers could utilize both frameworks to help ensure their test cases are effective at exposing both general faults and energy-specific problems.

Mutation Testing Effectiveness and Efficiency. Several researchers have proposed approaches to measure the effectiveness and efficiency of mutation testing [47], [48], [49], [50] to devise strategies for reducing the effort required to generate effective mutant sets [51], [52], [53], and to define theoretical frameworks [46], [54]. 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.

Existing Tools. Most of the available mutation testing tools are in the form of research prototypes. Concerning Java, representative tools are μJava [55], Jester [56], Major [28], Jumble [57], PIT [27], and Javalanche [58]. 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 are only three available tools: First, muDroid [59], which performs the mutations at byte code level by generating one APK (*i.e.,* one version of the mobile app) for each mutant. Second, Deng *et al.* [60], which performs the mutation at source code level and then compiles it to obtain an APK. Third, μDroid [26], discussed earlier which performs energy-aware mutations. The tools for mutation testing can be also categorized according to the tool's capabilities (*e.g.,* the availability of automatic test 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 [61] and in the paper by Madeysky and Radyk [62]. As previously mentioned, similarly to muDroid [59], we also perform a language-agnostic, bytecode level mutation, which makes the mutation independent of the different programming languages supported by Android.

Empirical Studies on Mutation Testing. Daran and Thévenod-Fosse [63] 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.* [48], [64] 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.* [23] 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. Luo *et al.* [24] compared the effect of test case prioritization techniques on real faults vs. mutants, and found that mutants tend to overestimate the effectiveness of test suites.

Petrovic *et al.* [65] propose a diff-based probabilistic mutation approach. Since their work has been conducted in the context of a company (Google) coping with large-scale systems, their approach is targeted to systems of such a scale. They use a heuristic to identify non-interesting instructions in order to reduce the number of lines analyzed when generating mutants. The implementation of such a tool at a company the scale of Google illustrates the growing practical importance of mutation analysis and highlights the need for effective mutation testing frameworks for different software domains.

In relation to the aforementioned work, our work proposes the use of mutation operators that are as close as possible to actual faults occurring in Android apps.

2.2 Analysis of Closed-Source Mobile Apps

Most of the approaches that aid in automating or improving software engineering tasks are designed considering source code as the main artifact because of its familiarity to developers and its expressiveness (*e.g.,* possible existence of code comments). In contrast, intermediate representations (IR) that are closer to executable files are "finer-grained" in the sense that "higher-level" source code files are often translated to several "lower-level" concrete machine-level instructions represented in different IRs. Because these two representations encode different types of information, researchers must investigate the trade-offs of using each information for supporting different software engineering tasks.

Exploring both source-code and IR-based development tools has several practical implications. While many existing approaches often rely on source code for supporting automated software engineering tasks, there are situations in which solutions are untenable in commercial environments. For example, when third-party services are used to outsource software engineering tasks without releasing the source code (*i.e.,* the services work directly on executable files), traditional static analysis is often not possible. This is because, for a variety of different legal and organizational reasons, app source code is not made available to third-party contractors or software testers, making it difficult to enable cloud/crowd-source services that utilize state-of-the-art static analysis approaches. Furthermore, automated approaches for SE tasks that operate directly upon executable files are often more convenient for developers to work with, as they are often faster (due to lack of re-compilation) and can be more easily integrated into increasingly popular CI pipelines (as they require fewer files than the entire code base). However, any type of analysis that relies only on executable files (*i.e.,* dynamic analysis) can be limited when compared to static analysis that can be done directly on the code. An example of this is presented by Ghanbari *et al.* [66], while performing program reparation at Java Bytecode. They use mutant generation to create a set of suitable patches that are evaluated by their capacity of passing the app tests. The reported repair patches generation is 10x faster than state-of-the-art source-code-based approaches, while being somewhat limited more limited in scope compared to similar full-fledged static analysis techniques.

Given the lack of source code under certain contexts (*e.g.,* app execution on devices), there are previous efforts that have analyzed or proposed approaches for dealing with APKs, and in particular for specific security-related tasks. Li *et al.* [67] summarize those efforts in a systematic literature review of approaches for static analysis of Android Packages. In particular, their work reports 124 papers and classifies them into 8 categories of tasks: (i) Private Data Leaks (46 papers), (ii) Vulnerabilities (40 papers), (iii) Permission Misuse (15 papers), (iv) Energy Consumption (9 papers), (v) Clone Detection (7 papers), (vi) Test Case Generation (6 papers), (vii) Cryptography Implementation Issues (3 papers) and (vii) Code verification (3 papers). Note that, as of today, there is no support at APK level for tasks such as automated documentation or mutation testing. Also, the support at APK level for automated testing is only provided by rippers that systematically explore the apps GUI [36]. Concerning the intermediate representations used for the analyses, Li *et al.* [67] report that the top intermediate representations (IR) used are JIMPLE (38 papers), SMALI (26 papers) and Java Bytecode (22 papers).

In general, state-of-the-art approaches for automated software engineering can be enabled in local environments where the source code is available and can be manipulated by the owners. Conversely, state-of-the-practice approaches offered by third-parties (*e.g.,* testing) rely on manual analysis of apps, or on automated dynamic analysis that operates at the APK level. The mobile development community is quickly moving towards using cloud-services and crowd-sourced services for software engineering tasks as they can help to reduce the cost and time devoted to otherwise expensive activities. However, since these services have access only to the APKs, they can not take advantage of state-of-the-art approaches that rely on the existence of source code for extracting intermediate representations or models that drive the analysis execution or the artifacts generation. Therefore, it is clear that mobile app testing and mutation analysis should move towards supporting APK-only analyses to better support a wide range of popular development workflows.

3 A TAXONOMY OF CRASHES/BUGS IN ANDROID APPS

In this section, we describe our 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 $\simeq \pm 5\%$ 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 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 [68], [69]. 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.* [70]. Through 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 [71], [72], [73], [74]. 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 information about faults/errors that can happen in Android apps [75], [76]. 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[2] 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 [5], [6], [9], [75], [78], [79], [80], [81], [82]). 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 [73], [83], [84], [85], [86], [87]. However, only a reduced set of reviews are in fact informative and useful for developers [86], [88]. Therefore, to automatically detect informative reviews reporting bugs and crashes, we leverage CLAP, the tool developed by Villarroel *et al.* [89], 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.* [90]. This dataset reports user reviews for multiple releases of $\sim$21K apps, in which CLAP identified 718,132 reviews as bug-reporting. Our statistically significant sample included 384 reviews that we analyzed.

2. The complete list of papers is provided in our online appendix [77].

The dataset collected from the six sources listed above was manually analyzed by eight taggers following a procedure inspired by open coding [91]. The taggers were authors of this paper. 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.,* the 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.,* causes behind the bug) is extremely high, such a choice helps using consistent naming and does not introduce substantial bias.

In the cases for which there was no agreement between the two evaluators ($\sim$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 to solve the conflict and define the most appropriate tag to assign (this happened for $\sim$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).

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, 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

Fig. 1 depicts the taxonomy that we obtained through manual coding. The black rectangle in the bottom-right part of Fig. 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, indicate 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 Fig. 1 lets us conclude 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 the 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. To this extent, the study that we have conducted allows us to create mutation tools for Android apps, described in Section 4, that encompass Android-specific bugs as well as Java bugs frequently occurring in Android apps.

4 Mutation Testing for Android Apps

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 non-compilable and trivial mutants, (iv) have implementation rules of the operators for both Java and SMALI representations, and (v) 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.

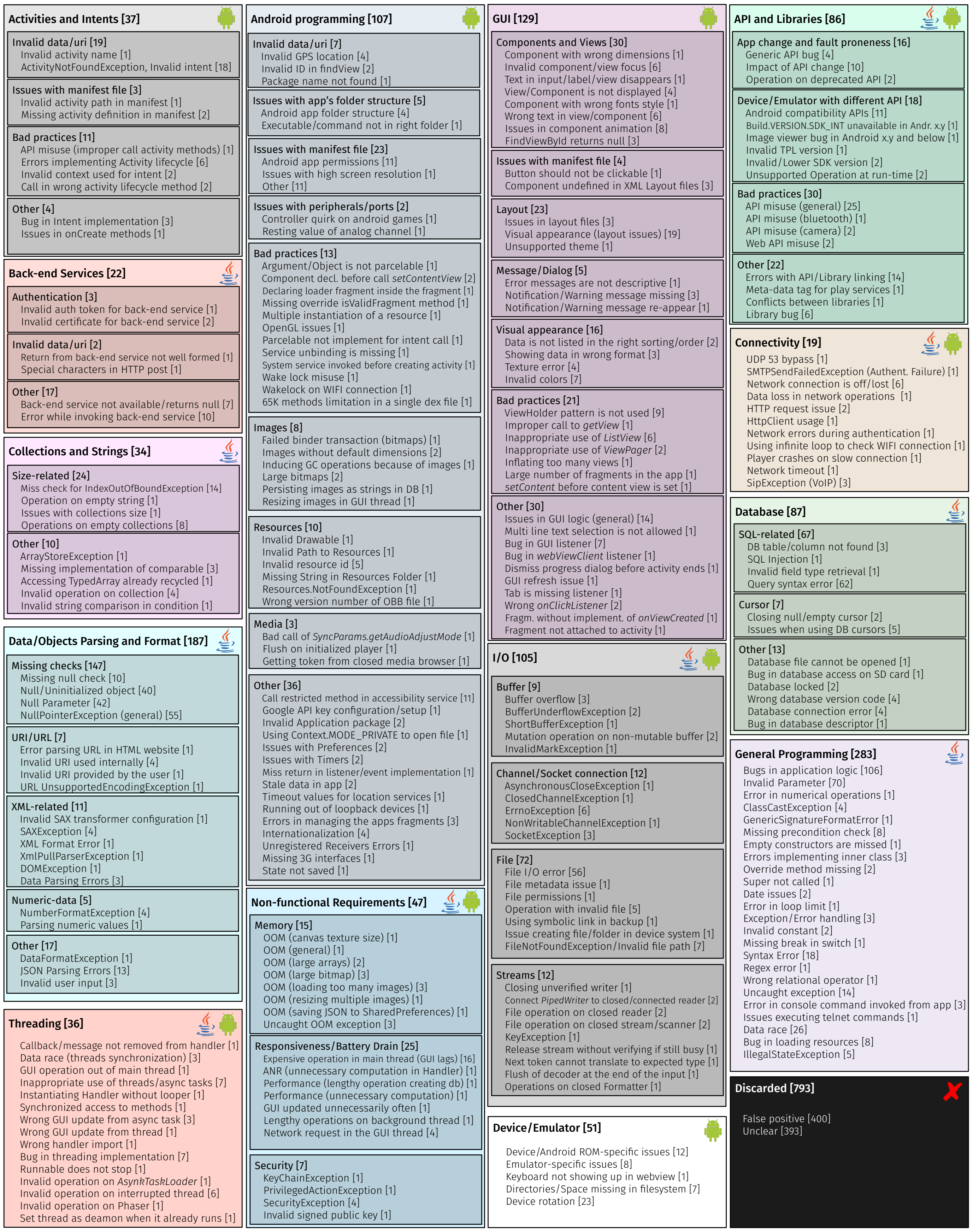
Our choice for having implementation rules also in SMALI is because this intermediate representation is one of the most used ones for analyzing APK files [67], and because of the availability of parsers/lexers that are easy to use and configure. Having access to SMALI code extracted directly from the APK makes it easier to repackage the app code in an APK, reducing the compilation/building time from Java source code to DEX. The availability of SMALI mutation could be particularly useful in circumstances where there is a need to generate (and deploy) several (mutated) APK instances, and therefore SMALI mutation could be faster than mutating and compiling source code.

To have an implementation of the proposed operators for both source code and APK, we created two tools, `MDroid+` [30] and `MutAPK` [31]. We describe the details of both tools for the remainder of this section.

4.1 Mutation Operators

Following the aforementioned conditions, we defined a set of 38 operators, covering as many fault categories as possible (10 out of the 14 categories in Fig. 1), and complementing the available classic Java mutation operators. We did not consider the following categories:

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;



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]

Fig. 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), a brief operator descriptions, and if it is implemented in `MDroid+` and `MutAPK`.

Mutation Operator	Det.	Cat.	Description	MDroid+	MutAPK
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	✓	x
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"true"	✓	✓
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	x	x
NotSerializable	AST	GP	Select a serializable class, remove "implements Serializable"	✓	x
NullMethodCallArgument	AST	GP	Randomly set null to a method call argument	x	✓
BuggyGUIListener	AST	GUI	Delete action implemented in a GUI listener	✓	x
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 Activity.findViewById call	✓	✓
InvalidViewFocus	AST	GU	Randomly focus a GUI component	x	✓
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	✓	✓

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. Previous work by Lin *et al.* [92] on refactoring workers and threads could be used as a foundation for defining operators.

The list of defined mutation operators is provided in Table 1. These operators were implemented in Java (for source code-based mutations in `MDroid+`) and SMALI (for APK-based mutations in `MutAPK`). The locations for the mutations are identified by using a Potential Failure Profile (PFP). The PFP lists code locations that could be modified to inject a mutation. The 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.

To extract the PFP, both `MDroid+` and `MutAPK` statically analyze 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+` and `MutAPK` generate a mutant for each location in the PFP. Mutants are initially generated as clones of the original app, and then the clones are automatically compiled/built/packaged 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, both tools automatically detect the corresponding mutation operator and apply the mutation either in the source code (for `MDroid+`) or intermediate representation (for `MutAPK`). Details of the detection rules and code transformations applied with each operator are provided in our replication package [93].

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.* [25]. Future work will be devoted to cover a larger number of fault categories and define/implement a larger number of operators.

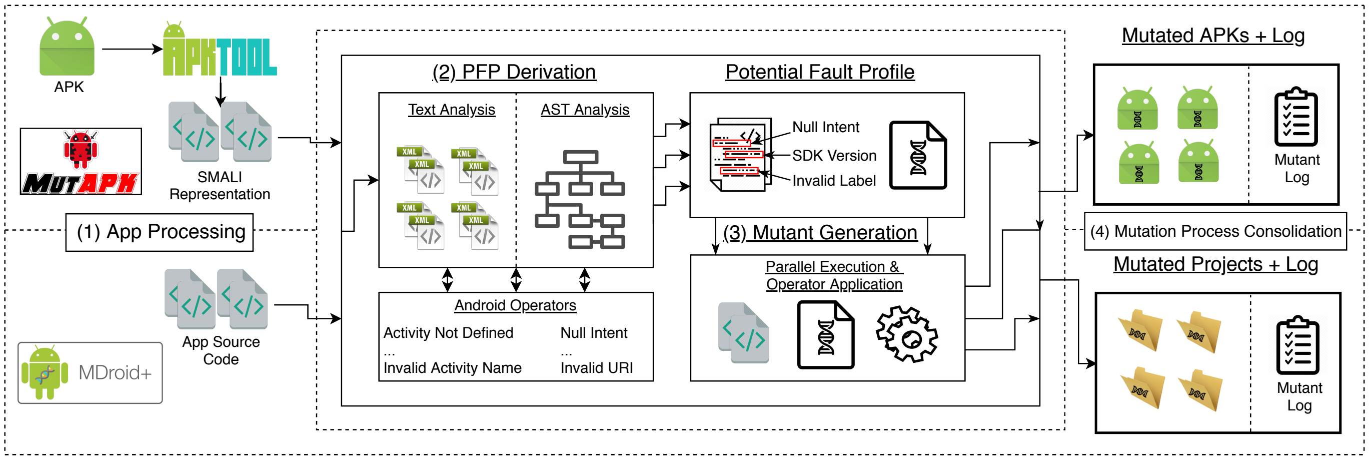


Fig. 2: Overview of `MDroid+` and `MutAPK` workflows.

4.2 `MDroid+` and `MutAPK`

To ensure that `MDroid+` and `MutAPK` are effective, practical, and flexible/extensible tools for mutation testing, both tools take into account the following design considerations: (i) an empirically derived set of mutation operators; (ii) a design embracing the open/closed principle (*i.e.,* open to extension, closed to modification); (iii) visitor and factory design patterns for deriving the Potential Failure Profile (PFP) and applying operators; (iv) parallel computation for efficient mutant seeding. Both tools are written in Java and are available as open source projects [30], [31].

Fig. 2 presents an overview of the workflow for both tools. There are four main stages for both tools: First, **App Processing** where `MutAPK` requires the *APKTool* [94] library to decode an app to get an intermediate representation of the compiled code and resources. In this stage, `MDroid+` requires the source code and resources folder of a given app (*e.g.,* the `/res/` folder).

Second, **PFP Derivation** consisting in two processes: (i) resource files processing: by taking advantage of the structure provided in given files (*i.e.,* XML files, resource files), to identify XML tags that match the different operators either by its tag name (*e.g.,* WrongStringResource that search for `<string>` tags) or tag attributes (*i.e.,* InvalidLabel that search for `android:label` in Manifest's tags), and (ii) Code-related files: by pattern matching API calls. In `MDroid+` case, the JDT Core DOM Library is used to generate an AST representation from JAVA code and matches AST's nodes with API call templates defined in a file called *target-apis.properties*. In contrast, `MutAPK` uses *Antlr*, *JFlex* and *GAP* libraries, to generate the AST given it works with SMALI code. Nevertheless, since SMALI code uses a larger set of instructions to represent a JAVA instruction, a larger set of API calls must be matched for each operator.

Third, **Mutant Generation** is performed based on the previously-generated *PFP* and the catalog of implemented operators (explained in Section 4.2.1). Therefore, using the mutation rules, for each location in the *PFP* a copy of the app processing result is generated and modified. To provide the most efficient process, both tools allow users to parallelize the generation process, utilizing the multi-core architecture of most modern hardware.

Finally, the fourth stage is the **Mutation Process Consolidation**: `MutAPK` generates for each mutant an APK using the *APKTool* [94] and signs it with the *Uber APK Signer* [95]; `MDroid+` stores the mutated source code folder. Finally, both tools generate a log file for the mutation process result.

4.2.1 *Implemented Mutation Operators*

One of the main components in the mutation process is the set of mutation operators that define the correct way to represent a naturally occurring fault in an Android project. Specifically, in this study we define 38 mutation operators that belong to 10 of the 14 categories extracted in the previous taxonomy (*i.e.,* Fig. 1). As it can be seen in Table 1, `MDroid+` implements 35 operators while `MutAPK` implements 34. It worth noticing that 32 of these implemented operators are shared between tools.

For example, a *Missing Permission on Manifest file* could be a fault likely to be found in an Android Project. Therefore, both `MDroid+` and `MutAPK` have an operator called *MissingPermissionManifest* that, given a permission on the manifest file, remove it from the file by replacing the complete line with a blank space.

Another mutation operator we defined is *DifferentActivityIntentDefinition* where, given an intent declaration (Listing 1), `MDroid+` replaces the Activity.class argument in the intent instantiation with the Activity.class value of another class belonging to the project (Listing 2).

Listing 1: Intent instantiation Java

```
Intent intent = new Intent(main.this, ImportActivity.
    class);
```

Listing 2: `MDroid+` operator result

```
Intent intent = new Intent(main.this, ExportActivity.
    class);
```

`MutAPK` also provides this operator. However, since it works at the APK level, the definition of the mutation rule is in terms of SMALI representation. Therefore, the intent instantiation seen in Listing 1 is represented in SMALI as it can be seen in Listing 5.2. Moreover, the mutation result seen in Listing 2 is represented as it is shown in Listing 4.

Listing 3: SMALI Intent instantiation Java

```
const-class v3, Lcom/fsck/k9/activity/ImportActivity;
invoke-direct {v1, v2, v3}, Landroid/content/Intent;-> <
    init>(Landroid/content/Context;Ljava/lang/Class;)V
```

Listing 4: `MutAPK` operator result

```
const-class v1, Lcom/fsck/k9/activity/ExportActivity;
invoke-direct {v1, v2, v3}, Landroid/content/Intent;-> <
    init>(Landroid/content/Context;Ljava/lang/Class;)V
```

5 Empirical Study: Applying Mutation Testing Operators to Android Apps

The *goal* of this study is to: (i) understand and compare the ***applicability*** of `MDroid+`, `MutAPK`, and other currently available mutation testing tools; (ii) understand the ***underlying reasons*** for mutants generated by these tools that cannot be considered useful, *i.e.,* non-compilable mutants, mutants that cannot be launched, mutants that are equivalent to the original app, and mutants that are duplicate; and (iii) understand the pros and cons of conducting mutation testing at source code and APK level. This study is conducted from the *perspective* of researchers interested in improving current tools and approaches for mutation testing 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 non-compilable (e.g., those leading to failed compilations), trivial (e.g., those leading to crashes on app launch), equivalent, and duplicate mutants produced by the studied tools when used with Android apps?*
- **RQ$_3$:** *What are the major causes for non-compilable, trivial, equivalent, and duplicate mutants produced by the mutation testing tools when applied to Android apps?*
- **RQ$_4$:** *What are the benefits and trade-offs of performing mutation testing at APK level vs source code level?*

5.1 Study Context and Methodology

To answer **RQ$_1$**, we analyzed the complete list of 102 mutation operators from seven mutation testing tools (Major [28], PIT [27], μJava [55], Javalanche [58], muDroid [59], Deng *et al.* [25], and `MDroid+`/`MutAPK` to investigate their ability to "*cover*" bugs described in 726 artifacts[3] (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 procedure previously described for the taxonomy building. In other words, there were two evaluators per document having the goal of tagging the type of bug described in the document; conflicts were solved by using a majority-rule schema; and the tagging process was 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 (Fig. 1). The evaluations were free to include new types of previously unseen bugs.

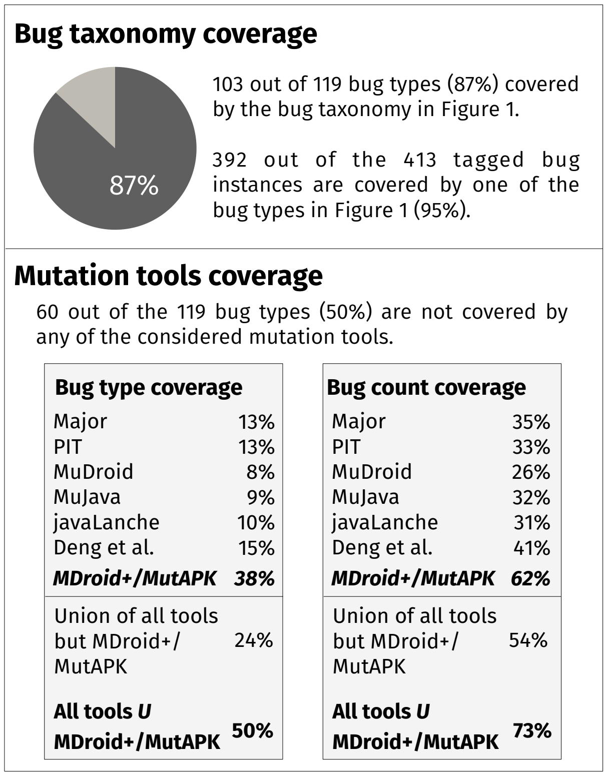
3. With "cover" we mean the ability to generate a mutant simulating the presence of a given type of bug.

Upon addressing RQ$_1$ we report (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 our tools with respect to the existing tools.

Concerning **RQ$_2$, RQ$_3$** and **RQ$_4$**, we compared the tools based on different mutation testing metrics. In particular, we compared Major, PIT, muDroid, `MDroid+`, and `MutAPK`. Major and PIT are popular open source mutation testing tools for Java, that can be tailored for Android apps. The tool by Deng *et al.* [25] is a context-specific mutation testing tool for Android available at GitHub. 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 (*i.e.,* Android-specific bytecode representation). Jabbarvand and Malek [26] present a tool called μDroid that generates mutants to validate the energy usage of apps. However, this tool is only compatible with Eclipse, precluding it from being used with the large set of apps collected for our empirical evaluation.

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 [6], 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 and APKS is available in our replication package [93].

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; and (v) compute the number of equivalent and duplicate mutants. We refer readers to our replication package for the complete technical methodology used for each mutation tool [93].



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+/MutAPK	***38%***
Union of all tools but MDroid+/ MutAPK	24%
All tools *U* MDroid+/MutAPK	**50%**

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

Fig. 3: Mutation tools and coverage of analyzed bugs.

To quantitatively assess the applicability and effectiveness of the considered mutation tools to Android apps, we used five metrics: **Total Number of Generated Mutants (TNGM)**, **Non-Compilable Mutants (NCM)**, **Trivial Mutants (TM)**, **Equivalent Mutants (EM)**, and **duplicate Mutants (DM)**. Additionally, we analyzed the time required by both `MutAPK` and `MDroid+` to: (i) generate a mutated copy of the app and to (ii) compile/build the copy.

In this paper, we consider *Non-Compilable 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 exhibited when launching the app. If a mutant crashes upon launch, we consider it as a *trivial mutant* because it could be detected by any test case that starts the app. Note that we use the term "Non-Compilable Mutants (NCM)" as a synonym of still-born mutants.

Two other metrics that one might consider to evaluate the effectiveness of a mutation testing tool is the number of *equivalent* and *duplicate mutants* the tool produces. However, in past work, the identification of equivalent mutants has been proven to be an undecidable problem [96], [97], and both equivalent and duplicate mutants require the existence of test suites (not always available and sufficiently complete for this purpose in the case of the Androtest apps).

Papadakis *et al.* [98] proposed a method to overcome the lack of test suites and to reduce the computational time required to detect equivalent and duplicate mutants by relying on proxies computed at the machine code level. Note that this idea has also been explored previously by Offutt *et al.* [99] and Kintis *et al.* [100].

In particular, Papadakis *et al.* [98] propose using "Trivial Compiler Equivalence (TCE)", which relies on comparing compiled machine code to detect equivalence between (i) mutants and original programs, and (ii) among mutants to detect duplicated ones. TCE has been shown to detect, on average, 30% of equivalent mutants [98] on a benchmark of 18 small/medium C/C+ programs [101].

We used TCE to compute equivalent and duplicate mutants at APK level. Instead of doing binary comparisons, we computed hashes, which is also proposed by Papadakis *et al.* [98] as an alternative for the comparisons. Because mutations of Android apps can be applied on source code, manifest files, or resource files (*i.e.*, XMLs), for each original APK and generated mutants, we computed four different hashes. Given and *APK* file under analysis, we computed:

1) $H(APK)$: hash of the whole apk file;
2) $H(APK_{resources})$: hash of the resource files in the *APK* file, which is computed as the concatenation of hash for each resource file;
3) $H(APK_{manifest})$: hash of the manifest file in the *APK* file;
4) $H(APK_{SMALI})$: hash of the SMALI files extracted from the *APK* file, which is computed as the concatenation of hash values for each SMALI file.

To detect equivalent mutants, we compared the four hashes of an original APK, with the four hashes of each of the generated mutants. In cases where all of the mutant hashes are equal to the original ones, the corresponding mutant is declared as equivalent. Similarly, to detect duplicate mutants, we compared the four hashes, but among the mutants. Note that in the case of duplicate mutants we report only the number of mutants that should be discarded.

5.2 Results

RQ$_1$: Fig. 3 reports (i) the percentage of bug types, identified during our manual tagging, that are covered by the taxonomy of bugs shown in Fig. 1 (top part of Fig. 3), 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 Fig. 3 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 bug categories 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 altogether 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 Fig. 3, our first important finding highlights the limitations of the experimented mutation tools (including `MDroid+/MutAPK`) in potentially unveiling the bugs subject of our study. Indeed, for 60 out of the 119 bug types (50%), none of the considered tools can 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

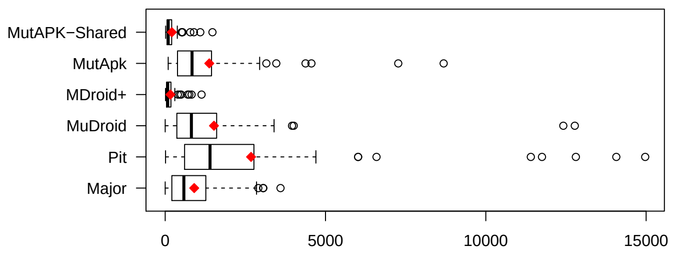


Fig. 4: Distribution of number of mutants generated per app.

generate mutants covering the *Bug in webViewClient listener* and the *Components with wrong dimensions* bug types.

When comparing the mutation tools considered in our study, MDroid+ and MutAPK clearly stand out as the tools ensuring the highest coverage both in terms of bug types and bug instances. In particular, mutators generated by MDroid+/MutAPK 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.* [25]) covers 15% of the bug types (61% less as compared to MDroid+/MutAPK) and 41% of the bug instances (34% less as compared to MDroid+/MutAPK). Also, we observe that MDroid+/MutAPK cover 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+/MutAPK increases the percentage of covered bug types to 50% (73% of the bug instances). Some of the categories covered by MDroid+/MutAPK and not by the other tools are: *Android app permissions*, thanks to the MissingPermissionManifest operator, and the *FindViewById returns null*, thanks to the FindViewByIdReturnsNull operator.

Finally, we statistically compare the proportion of bug types and the number of bug instances covered by MDroid+/MutAPK, by all other techniques, and by their combination, using Fisher's exact test and Odds Ratio (OR) [102]. The results indicate that:

1) The odds of covering bug types using MDroid+/MutAPK 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+/MutAPK 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+/MutAPK and other techniques are 1.35 times greater than other techniques alone, with a significant difference (p-value=0.008).

Summary of RQ$_1$ Findings: *MDroid+ and MutAPK outperformed the other 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 existing mutation operators for Java to generate mutants that are representative of real faults in mobile apps.*

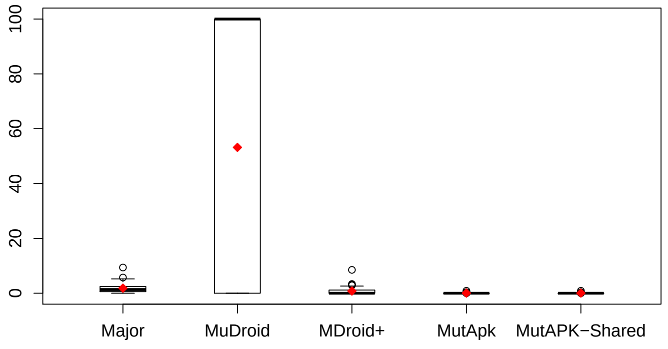


Fig. 5: Distribution (%) of non-compilable mutants.

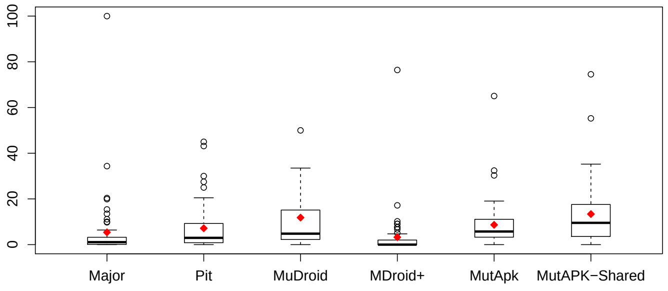


Fig. 6: Distribution (%) of trivial mutants.

RQ$_2$: Fig. 4 depict the total number of mutants generated by each tool on each analyzed app, while Fig. 5 and Fig. 6 show the percentage of (a) Non-Compilable Mutants (NCM) and (b) Trivial Mutants (TM) respectively. As stated in Section 4.2, MutAPK was based on MDroid+, therefore, in the following comparisons we will show the results for both MutAPK and MutAPK-Shared[4] to study the benefits of applying the MDroid+ operators at APK level.

On average, 167, 207, 1.3k+, 904, 2.6k+, and 1.5k+ mutants were generated by MDroid+, MutAPK-Shared, MutAPK, 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; note that PIT uses object oriented-based mutators for Java source code. muDroid tends to generate a larger number of mutants due to its more generic mutation operators, meaning that there are more potential instances in the source code for mutants to be seeded. MutAPK generates significantly more mutants than Major (Wilcoxon paired signed-rank test adjusted p-value< 0.001 with Holm's correction [103]) and significantly fewer mutants than PIT (Wilcoxon paired signed-rank test adjusted p-value < 0.001 with Holm's correction). However, MDroid+ and MutAPK-Shared generate significantly fewer mutants than Major, muDroid, and PIT (Wilcoxon paired signed-rank test adjusted p-value < 0.001).

The average percentage of **Non-Compilable Mutants** (NCM) generated by MutAPK, MutAPK-shared, MDroid+, Major and muDroid over all the apps is 0.04%, 0.31%, 0.56%, 1.8%, and 53.9%, respectively, while no NCM were generated by PIT (Fig. 5). MDroid+ produces significantly fewer NCM than Major (Wilcoxon paired signed rank test

4. MutAPK-Shared means MutAPK using only the operators shared with MDroid+

adjusted p-value < 0.001, and large Cliff's d=0.59) and than muDroid (adjusted p-value < 0.001, and medium Cliff's d=0.35). `MutAPK` and `MutAPK`-Shared produces significantly fewer NCM than Major and muDroid (Wilcoxon paired signed rank test adjusted p-value< 0.001)

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 NCM 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 and `MDroid+` work at the source code level and compile the app in a "traditional" manner. Thus, it is more prone to NCM and requires an overhead in terms of memory and CPU resources needed for compiling/building the mutants. Finally, muDroid and `MutAPK` operate on `APKs` and `smali` code, reducing the computational cost of mutant generation, but significantly increasing the chances of NCM; muDroid is the top-one generator of NCM with an average of 53.9% of NCMs per app. However, `MutAPK` and `MutAPK`-Shared are the ones generating the least amount of NCM with averages of 0.04% and 0.31% respectively. This due to the process designed to create the mutation rules, as explained in Section 4.2.

All the analyzed 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 the analysis. The average of the distribution of the percentage of TM over all apps for `MDroid+`, Major, PIT, `MutAPK`, muDroid and `MutAPK`-Shared is 2,42%, 5.4%, 7.2%, 9%, 11.8% and 13.62%, respectively (Fig. 6). `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). `MutAPK` generates significantly more TM than Major (Wilcoxon paired signed rank test adjusted p-value < 0.001) and `MutAPK`-Shared generates significantly more TM than PIT (Wilcoxon paired signed rank test adjusted p-value < 0.001)

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 GM by Major were TM. At the same time, for HNDroid app `MutAPK` generates 673 trivial mutants out of 1038 generated mutants but also for Anycut app generates only 2 TM out of 380 GM. Finally, `MDroid+` generates also for HNDroid 94 trivial mutants from 123 generated. Both `MutAPK` and `MDroid+` generate the smallest number of NCM with 55 and 37 respectively. However, `MDroid+` generates less TM, only 213 in total across apps due to being also the one generating less mutants, around 167 per app. At the same time, `MutAPK` belongs to the top generators of TM, with around 9% of TM per app.

Following the approach proposed by Papadakis *et al.* [98], we found that none of the tools generated equivalent mutants when comparing the hash values between the original APKs and the mutated APKs. As previously mentioned, the four hash values calculated for both original and mutated APK should be equal to identify a mutant as an equivalent mutant. Nevertheless, we used the same approach to find duplicate mutants by performing a pairwise comparison of all mutants. As a result, we found that `MutAPK`, MutAPK-Shared, PIT, and muDroid generate duplicate mutants. Specifically, 211, 43, 8, and 2,031 duplicate mutants were generated, respectively. Note that muDroid generates more duplicate mutants that other tools, with a percentage of 6.55% of the generated mutants; the second one is MutAPK-Shared with 0.38%; the third is `MutAPK` with 0.28%; and finally PIT with 0.006%.

Summary of RQ$_2$ findings: *As for the generation of mutants, all the analyzed tools (Major, Pit, muDroid, `MDroid+`, `MutAPK`) generated a relatively low rate of trivial mutants, with muDroid being the worst with a 11.8% average rate of trivial mutants. Additionally, no equivalent mutants were found for any tool, according to a hash-based comparison between the original APKs and the corresponding mutants. Nevertheless, 4 tools (`MutAPK`, MutAPK-Shared, PIT and muDroid) generated duplicate mutants, with muDroid being the worst with a 6.55% of total duplicate mutants.*

RQ$_3$: 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 NCM. It may seem surprising that ROR generated many NCM, 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 NCM (although all the operators have relatively high failure rates), and the Relational Value Replacement (RVR) has the highest number of TM. For `MutAPK`, the FindViewByIdReturnsNull and NullValueIntentPutExtra operators had the highest number of NCM, while the NullMethodCallArgument operator generates the highest number of TM.

The details of the mutation operators being the source of duplicate mutants are depicted in Tables 2, 3, and 4. Table 2 presents the results for muDroid. As previously mentioned, muDroid generates the largest number of duplicate mutants. One example is *"Relational Operator Replacement"* mutant operator with 1135 duplicate mutants of 28,560 generated ones, which account for a total of 4% of duplicate mutants generated with this operator.

PIT's results are shown in Table 3; in this case, there are only 8 duplicate mutants where 6 of them belong to a relation between mutants having *"NegateConditional"* and *"RemoveConditional"* operators.

Finally, for `MutAPK` (Table 4) the *"NullMethodCallArgument"* is the operator generating more duplicate mutants. Additionally, it is worth noticing that there are 12 duplicate mutants in `MutAPK` and MutAPK-shared that are between different pairs of operators; this behavior is further analyzed later.

TABLE 2: Number of duplicate mutants created by muDroid grouped by operator.

Mutation Operators	Amount
Relational Operator Replacement	849
Inline Constant Replacement	228
Arithmetic Operator Replacement	486
Return Value Replacement	34
Logical Connector Replacement	6
Negative Operator Inversion	1
Inline Constant Replacement & Relational Operator Replacement	180
Inline Constant Replacement & Arithmetic Operator Replacement	77
Arithmetic Operator Replacement & Relational Operator Replacement	57
Return Value Replacement & Inline Constant Replacement	57
Return Value Replacement & Relational Operator Replacement	24
Return Value Replacement & Arithmetic Operator Replacement	14
Logical Connector Replacement & Inline Constant Replacement	4
Relational Operator Replacement & Logical Connector Replacement	3
Negative Operator Inversion & Inline Constant Replacement	3
Arithmetic Operator Replacement & Logical Connector Replacement	2
Negative Operator Inversion & Relational Operator Replacement	2
Negative Operator Inversion & Arithmetic Operator Replacement	2
Logical Connector Replacement & Return Value Replacement	2
Total (MuDroid)	2,031

TABLE 3: Number of duplicate mutants created by PIT grouped by operator.

Mutation Operators	Amount
NegateConditional	1
RemoveSwitch	1
NegateConditional & RemoveConditional_ORDER_ELSE	6
Total (PIT)	8

To qualitatively investigate the causes behind the crashes and duplicate generation, four authors manually analyzed a randomly selected sample of 15 crashed mutants and 10 duplicate 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 into 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 the deletion of a statement initializing the main `Activity` leading to a `NullPointerException`. No duplicate mutants were identified among the mutants generated by Major.

TABLE 4: Number of duplicate mutants created by `MutAPK` and `MutAPK`-Shared grouped by operator.

Mutation Operators	Amount
DifferentActivityIntentDefinition	10
NullValueIntentPutExtra	6
NullIntent	6
WrongStringResource	3
InvalidIDFindView	3
LengthyGUICreation	2
InvalidActivityPATH	1
ActivityNotDefined	1
InvalidFilePath	1
NullValueIntentPutExtra & NullBackEndServiceReturn	3
NullValueIntentPutExtra & MissingPermissionManifest	2
WrongStringResource & NullIntent	2
MissingPermissionManifest & ViewComponentNotVisible	2
LengthyGUICreation & LengthyGUIListener	1
Subtotal (`MutAPK`-Shared - Common operators)	43
NullMethodCallArgument	165
InvalidViewFocus	1
InvalidActivityPATH & InvalidViewFocus	2
Total (`MutAPK`)	211

muDroid. This tool is the one exhibiting the highest percentage of NCM 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 classpaths). 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.

In terms of duplicate mutants, muDroid[5] is also the tool generating the highest amount of DM. As listed in Table 2, the 6 mutants operators generate duplicate mutants, and there are 13 combinations of operators that also generate duplicate mutants. The mutation operator that generates more duplicate mutants is *Relational Operator Replacement (ROR)*. This operator generates around 850 mutants that are duplicate with other ROR mutants. Additionally, there are around 266 duplicate mutant pairs with one of the mutants being a result of ROR operator being applied. After manually analyzing the duplicate mutants, we found that there are implementation errors in muDroid, since several mutants generated with the ROR, ICR and AOR operators have identical mutations in the same app, despite being reported in the log files as different mutants.

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. In terms of the duplicate mutants, PIT generated the lowest rate (8 out of 103k mutants). The most common duplicate mutant case is between *NegateConditional* and *RemoveConditional_ORDER_ELSE*. From its definition, the *RemoveConditional_ORDER_ELSE* operator is a specialization of the base mutant operator *RemoveConditional*, whose objective is to change "a==b" to "true". The specialized operator negates the condition to ensure the ELSE block is executed; however, in some cases, the effect of both operators is the same. Consider, for example, the following source code snippet:

```
if(a < b){
    // Do something
} else {
    // Do something else
}
```

If we apply *NegateConditional* operator, PIT will look for the conditional operator used in the if statement (*i.e.*, $<$) and it would negate it, replacing it with a $>=$. However,

5. For the time of the writing of this article last update of MuDroid's source code was done in May 3, 2016

if we apply *RemoveConditional_ORDER_ELSE*, PIT would make the necessary change in the if condition so the else statement is executed. This is represented in replacing the < condition, with a >= condition. Therefore, both operators will give a final result where the < condition was replaced with a >= condition.

MDroid+. Table 5 lists the mutants generated by MDroid+ across all the systems (information for the other tools is provided with our replication package). In MDroid+, the overall rate of NCM was quite low with the ClosingNullCursor operator having the highest total number of NCM (across all the apps) with 13. These instances stem from edge case that trigger compilation errors involving cursors that have been declared Final, thus causing the reassignment to trigger the compilation error. The small number of other NCMs are generally other edge cases, and current limitations of MDroid+ can be found in our replication package with detailed documentation. No duplicate mutants were identified among the mutants generated by MDroid+.

MutAPK. Table 5 lists the mutants generated by MutAPK across all the systems (information for the other tools is provided with our replication package). The overall rate of NCM is very low in MutAPK, and most failed compilations pertain to specialized cases that would require a more robust static analysis approach to inject the mutations. However, it is worth noting that MutAPK works at APK level, and the mutation rules require more modifications (when compared to source code level mutations) to generate a valid mutation. For example, the *FindViewByIdReturnsNull* mutation rule consists of replacing the statements used to find a specific component with a null value assignment to the register where the corresponding result is stored. This storage process could store the value in a register higher than 16. This value is the maximum register index normally accepted by DALVIK instructions. However MutAPK uses the *const/4 v#, 0x0* instruction to set the null value. Therefore, if the register used for the assignment is above 16 the instruction will not compile. To correctly mutate the app, an extensive search for an empty register must be done to store the null value. However, if there is no empty register, MutAPK would need to save the value from one of the below-16-registers into a temporal above-16-register, use the first selected register to storage the null value while the process uses it and then reassign its original value. This problem also applies to *NullValueIntentPutExtra* mutation operator.

The operator generating the highest number of TM is *NullMethodCallArgument* (84.43%, *i.e.,* 4,264 out of 5,050). The main reason for this behavior is due to the nature of the mutation rule; the *NullMethodCallArgument* operator replaces one parameter value in a method call with a null value. Therefore, all method invocations with the null value as an argument will throw an exception when the method does not handle the null value. It is worth noting that MutAPK generates 63,441 mutants using this operator, therefore only 6.72% of generated mutants are trivial under our definition. Future work must be focused on avoiding mutations of this type in the main activities to avoid the TM case. There are also 3 other operators that increase the amount of TM generated by MutAPK. First, *FindViewByIDReturnsNull*, modifies a *findViewByID* call to return null. Second, InvalidIDFindView replaces the parameter that represents the view Id required with a generated randomly, therefore, the result of the *findViewByID* call will be a null value. Third, the *NullValueIntentPutExtra* operator replaces the value sent as extra in an intent with a null value. Therefore, just like what happens with *NullMethodCallArgument*, all the method invocations and statements that do not correctly handle the null values will generate an exception breaking the app.

In terms of duplicate mutants, MutAPK generates the largest amount with the *NullMethodCallArgument* operator. By definition, this operator changes the value of a parameter in a method call to null. In source code, it is possible to find method calls that use the same value more than once in a call. For example, in the a2dp.Vol app, the method *deleteAll* calls the method *delete* by providing twice a null value as parameter: `this.db.delete(TABLE\_NAME, null, null)`. This instruction at APK level also makes the call using two times the same parameter:

```
invoke-virtual {v0, v1, v2, v2}, Landroid/database/
    sqlite/SQLiteDatabase;->delete(Ljava/lang/String;
    Ljava/lang/String;[Ljava/lang/String;)I
```

However, MutAPK does not validate the current value of the parameter before changing it to null; therefore, the value for *v2* was already null before MutAPK injected the null assignment. This is also an example of an equivalent mutant that cannot be found by following the TCE approach, since the SMALI bytecode was modified and the hash values were not affected by the change.

It is also important to see that there are some mutant operators shared with MDroid+ that are generating duplicate mutants only at APK level, such as *DifferentActivityIntentDefinition* that has different implementations in both tools. This is a good example since this operator requires finding a new Activity name for making the change. In the MutAPK case, the operator is not validating that the activity should not be replaced with the same name (picked randomly from the list of activities in the APK).

Summary of RQ$_3$ Findings: *The performed analysis indicate that the PIT tool outperforms others in terms of ratio between non-compilable and generated mutants, because it does not generate any non-compilable mutant. However, MDroid+ and MutAPK provide Android-specific mutations, which make the tools (i.e., Pit, MDroid+, MutAPK) complementary for mutation testing of Android apps. MDroid+ and MutAPK generated the lowest rate of both non-compilable and trivial mutants (when compared to Major and muDroid), illustrating its immediate applicability to Android apps. Major and muDroid generate non-compilable mutants, with the latter having a critical average rate of 58.7% non-compilable mutants per app. Also, even when PIT generates duplicate mutants, the number is insignificant when compared to the number of mutants generated; at the same time MutAPK and MutAPK-Shared also generate a low number of duplicate mutants; some of them can be fixed by improving the current implementation.*

RQ$_4$: Table 5 presents the results from both MutAPK and MDroid+ in terms of Generated Mutants (**GM**), Non-Compilable Mutants (**NCM**) and Trivial Mutants (**TM**) per mutation operator defined in this study. Note that each tool has implemented some operators that the other does not. Therefore, we first study the times required by both tools to (i) generate a mutated copy of the app and to (ii)

TABLE 5: Number of Generated (GM), Non-Compilable (NCM), and Trivial Mutants (TM) created by `MDroid+` and `MutAPK`

	`MDroid+`			`MutAPK`		
Mutation Operators	**GM**	**NCM**	**TM**	**GM**	**NCM**	**TM**
WrongStringResource	3,394	0	14	3,432	0	10
NullIntent	559	3	41	482	0	37
InvalidKeyIntentPutExtra	459	3	11	477	0	9
NullValueIntentPutExtra	459	0	14	477	22	103
InvalidIDFindView	456	4	30	1,313	0	193
FindViewByIdReturnsNull	413	0	40	1,313	28	190
ActivityNotDefined	384	1	8	385	0	11
InvalidActivityName	382	0	10	383	0	50
DifferentActivityIntentDefinition	358	2	8	482	0	7
ViewComponentNotVisible	347	5	7	398	0	58
MissingPermissionManifest	229	0	8	227	0	7
InvalidFilePath	220	0	1	228	0	36
InvalidLabel	214	0	3	214	0	5
ClosingNullCursor	179	13	5	222	0	14
LengthyGUICreation	129	0	1	336	0	15
LengthyGUIListener	122	0	0	339	0	5
NullInputStream	61	0	4	90	0	4
WrongMainActivity	56	0	0	56	0	8
InvalidColor	52	0	0	47	0	0
NullOuptutStream	45	0	2	59	0	2
InvalidDate	40	0	0	20	0	0
InvalidSQLQuery	33	0	2	82	0	7
NullBluetoothAdapter	9	0	0	9	0	0
LengthyBackEndService	8	0	0	15	15	0
NullBackEndServiceReturn	8	1	0	34	5	2
InvalidIndexQueryParameter	7	1	0	82	0	2
OOMLargeImage	7	4	0	7	0	4
BluetoothAdapterAlwaysEnabled	4	0	0	1	0	0
InvalidURI	2	0	0	2	0	0
NullGPSLocation	1	0	0	2	0	0
LongConnectionTimeOut	0	0	0	0	0	0
SDKVersion	66	0	2	0	0	0
Subtotal (Common operators)	8,703	37	211	11,214	55	779
NotParcelable	7	6	0	-	-	-
NotSerializable	15	7	0	-	-	-
BuggyGUIListener	122	0	2	-	-	-
NullMethodCallArgument	-	-	-	63,441	0	4,264
InvalidViewFocus	-	-	-	398	0	7
Total	8,847	50	213	75,053	55	5,050

TABLE 6: Summary of time results: `MutAPK` vs. `MDroid+`.

Metric Name	`MutAPK`	`MDroid+`
Avg. Mutation Time (secs.)	284.67 x 10^{-3}	4.61
Avg. Compilation Time (secs.)	25.265	195
Avg. Full Mutant Creation Time (secs.)	25.549	199.61

compile/assemble a given mutant into an APK. Then, we analyze the mutation results taking into account only the operators that are common in both tools, and finally, we study the impact of tool-specific operators in the results.

As it was mentioned previously, we ran `MutAPK` and `MDroid+` over 55 apps. `MDroid+`'s default behavior generates mutated copies of the original source code; therefore, we implemented a wrapper that based on the `MDroid+` results builds the corresponding APKs. `MutAPK` works directly on APKs; therefore, the APK generation is embedded in the mutation process.

As it can be seen in Table 6, `MutAPK` takes 6.17% of the time required by `MDroid+` to mutate a copy of the app and 12.95% of the time required to compile/assemble the mutant into an APK. Therefore, `MutAPK` executes the complete mutation process (*i.e.,* mutation of app copy plus compilation/assembling) 87.2% faster than `MDroid+`.

Common operators. Concerning the mutation type metrics (Table 5), when considering only the common operators, `MDroid+` and `MutAPK` generated 8,703 and 11,214 mutants respectively. This shows that at the APK level, the *PFP* (Section 3) detects more locations for implementing mutations. `MutAPK` generates on average 78 more mutants per operator with a median of 7.5 more than `MDroid+`. However, the *SDKVersion* operator does not find instances in any app due to latest modifications of the Android building process, where all the details for SDK Version must be defined in the *build.gradle* file instead of the Manifest file.

The number of NCM is very similar in both cases. However, the percentage of TMs is larger with `MutAPK` (6.94%) than with `MDroid+` (2.24%), and this happens because of the *FindViewByIdReturnsNull*, *InvalidIDFindView*, and *NullValueIntentPutExtra* operators that account for 62,39% of the trivial mutants in `MutAPK`. Note that `MutAPK` generates more mutants than `MDroid+` for those operators, because SMALI representation of code statements must express each instruction in a line. Therefore, as it can be seen in Listing 5 a Java statement can contain several instructions that are solved from inside to outside. However, a given Java statement in SMALI uses a line for each instruction as it can be seen in Listing 6. Knowing that, `MDroid+`'s search power is reduced by the Java capability of chaining instructions. For example, for *FindViewByIDReturnsNull* operator `MDroid+` search for statements where the view is stored in a variable (see *Listing 7*). However, if the result of *findViewById* method is used directly as parameter (see *Listing 5*), `MDroid+` does not recognize that statement as part of the PFP.

Listing 5: Java chained instructions

```
highlight(findViewById(R.id.load_data_button));
```

Listing 6: SMALI representation of JAVA chained instructions

```
invoke-virtual {p0, p1}, Lio/github/hidroh/materialistic
    /AboutActivity;->findViewById(I)Landroid/view/View;
move-result-object v0
check-cast v0, Landroid/widget/TextView;
invoke-virtual {v5, v0}, Lio/github/hidroh/materialistic
    /AboutActivity;->highlight(I)Ljava.awt.String;
```

Listing 7: `MDroid+` mutation rule

```
ImageButton loadButton = (ImageButton) findViewById(R.id
    .load_data_button);
```

If NCM, TM and DM are removed, we can see that `MDroid+` generated 8,455 mutants versus 10,337 mutants generated by `MutAPK`. Therefore, the results suggest that `MDroid+` takes 9.79 more hours to generate 21.9% less functional mutants (*i.e.,* mutants that compile and are not trivial nor equivalent nor duplicate) than `MutAPK`.

Whole set of operators. Considering all the operators available with each tool, `MutAPK` generates per operator on average 63% more mutants than `MDroid+` with a median of 3.9% and a mode of 0%. We found that `MutAPK` generates a significant amount of extra mutants because of the *NullMethodCallArgument* operator. This operator is capable of generating 63,441 additional mutants for the 55 apps, which is around 6 times the amount of mutants generated by the rest of the operators. However, 6.72% of those are trivial and 0.26% are duplicate. Additionally, on the one hand `MutAPK` also implements *InvalidViewFocus*, that generated 398 mutants (only 7 were trivial and 1 duplicate). On the other hand, `MDroid+` has 3 additional operators that add 144 mutants to the list; 13 of them are non-compilable and only 2 are trivial. Even in this case, `MutAPK` is able to generate more functional mutants: 69,742 vs 8,579 generated by `MDroid+`.

Summary of RQ$_4$ Findings: *The clear benefit of performing APK-level mutation analysis (`MutAPK`) as opposed to source code-*

level mutation analysis (`MDroid+`*) relates primarily to the ease of use of the system, as it only requires a single file (i.e., an APK file instead of several source files), and generates mutants with higher compilability ratio in less time. Moreover, this makes the mutation tool applicable to apps written with various languages, i.e., Java, Kotlin, and Dart. However, this ease of use comes with a slight trade-off in terms of generating a higher number of mutants (which leads to an extensive execution effort from developers), and a higher number of trivial and duplicate mutants for certain operators that are likely to be discarded during mutant analysis.*

6 Discussion & Future Work

The results of the study show that generating mutants at APK level has some benefits when compared to mutants generation at the source code level. With tools like `MutAPK` third-parties could improve their services by using mutation testing without the need of having access to the source code. However, there is a trade-off, because there are pros and cons on both sides. Therefore, the decision of whether to use APK-level mutation versus source level should be made carefully, with consideration for the specific context and the needs of researchers/practitioners. In the remainder of this section, we discuss the aforementioned trade-off and differing usage scenarios as well as promising future work that builds upon `MDroid+` and `MutAPK`.

Mutant comprehension. APK-level mutation is faster but with a larger number of NCMs and TMs. Another aspect to consider with APK-level mutation is related to mutant comprehension. Since the mutations are performed at SMALI level, the locations of the changes are not the same as the equivalent changes at the source code level. Therefore, when doing APK-level mutation and reporting killed and survived mutants, the mutations are presented to developers/testers as the type of mutation (*e.g., InvalidIDFindView*), but also in a location that is not the same when compared to source code. Therefore, this could require extra effort of the developer/tester to create a mapping between the mutation location at SMALI level and the equivalent location at source code. Future work should be devoted to automatically provide users with the location of the mutations at source code level when APK-level mutation is used in such a way that no extra time is required by the users to create the location mapping.

Opportunistic programming. A recent paper by Petrovic *et al.* [65] presents how mutation testing is used at Google during code-reviewing, in particular, by providing "inline" feedback at the statement level. This approach could be extended to provide early feedback to developers while coding, *i.e.,* providing mutation-based hints directly on the IDE or during commit operations.

Classic vs Android-Specific operators. Our Android bug taxonomy (see Section 3) and the results for **RQ$_1$** show that using only classic operators for mutation of Android apps is not sufficient. Therefore, classic and Android-specific operators should be combined. Our `MutAPK` and `MDroid+` tools do not include classic operators in their catalogs. Therefore, for a more effective mutation testing process, future work could be devoted to extend the tools to include classic operators. However, this would imply redoing work already done by tools like Pit and Major. Consequently, another option is to dedicate future efforts to create "meta" tools that combine mutants generated by different existing tools according to user preferences.

New languages for Android app development. New programming languages have emerged in the Android ecosystem; on the one side, there are Kotlin and Flutter/Dart for native apps, on the other side, there are the hybrid/cross-platform frameworks. Using source code-level mutation requires specific frameworks for each language (*e.g.,* Kotlin), which is not a problem for APK-level mutation with `MutAPK`. Therefore, `MutAPK` can be used for the Android native languages (*i.e.,* Java, Kotlin, Dart) with the limitation of mutants comprehension (already mentioned). However, in the case of hybrid apps there is no current tool for mutation testing. Future work should be also focused on building bugs taxonomies for and mutation testing tools for Android hybrid apps.

More mutants at APK level and mutant selection. Time is an issue in mutation testing, for both generation and testing time. Although mutation at APK level drastically reduces the time required to generate executable mutants, `MutAPK` also generates a larger number of mutants (on average 1.3k per apps) because it is easier to perform more mutations at SMALI code than at the source code level. This behavior occurs because SMALI instructions are closer to machine operations and some syntactic sugar is not allowed at this level. For example, it is common in Java Android programming to have inline declarations of arguments, like in the following statement of an Android app:

```
startActivity(new Intent(FrActivity.this, ScActivity.class))
```

In this case, we could think that the *DifferentActivityIntentDefinition* mutant operator could be applied; however, the mutant operator at source code looks for an instruction like the following:

```
Intent intent=new Intent(FrActivity.this, ScActivity.class)
```

And since the intent is defined inside the parameter space, the operator will not identify this line as a mutable one. However, at APK level, the aforementioned instruction is expressed in multiple lines (*See following code snippet*): First, the intent is declared (*Lines 1-4*), and then, it is used as a parameter in the *startActivity* method call (*Lines 5-7*).

```
1  new-instance v2, Landroid/content/Intent;
2  const-class v3, Lcom/example/myapplication/ScActivity;
3  invoke-direct {v2, p0, v3}, Landroid/content/Intent;
4    -><init>(Landroid/content/Context;Ljava/lang/Class;)V
5  invoke-virtual {p0, v2},
6    Lcom/example/myapplication/FrActivity;
7    ->startActivity(Landroid/content/Intent;)V
```

Because of this, even when instructions are concatenated at source code level, they are separated and are easier to identify when applying mutation operators at SMALI level.

There are some (trivial) features in the analyzed tools to deal with large sets of mutants. Both `MDroid+` and `MutAPK` allow users to select the type of mutants to be generated. In addition, `MutAPK` provides a feature to select the total number of mutants to be generated. Thus, before starting the mutation process, the user can identify a statistically significant sample of mutants to use during mutation analysis. For example, if `MutAPK` indicates that 1,174 mutants will be generated, then with a confidence level of 95% and confidence interval of ±5%, only 290 mutants are required to have a significant sample. Anyway, `MutAPK`, while generating more mutants and using more mutant operators than

`MDroid+`, has a generation time that is 5.5 hours less than the time required by `MDroid+`.

However, practitioners and researchers should consider using mutants prioritization/selection techniques [104] during mutation testing of Android apps; for instance, mutants could be generated from the most change-/fault-prone code or the impact set of the last change-set. Mutants selection techniques for mobile apps is still an open topic for future research.

Comprehensive tool for mutation testing of Android apps. As of today, there is no mutation testing tool for Android apps. The tools and approaches used in our study are a first step (mutant generation) towards the goal of having a comprehensive tool for mutation testing. However, to achieve this goal, researchers and practitioners should envision an approach that is agnostic of the testing framework. Note that in the case of testing of Android apps, there are a plethora of available tools widely used. Therefore, to enable mutation testing, existing tools should be extended to be able to execute test suites written with any of the available frameworks for testing Android apps.

7 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 non-compilable, trivial, equivalent and duplicate mutants. In particular, for detecting equivalent and duplicate, we relied on the TCE approach [98], which was able to detect (on average) 30% of equivalent mutants on a benchmark of 18 small/medium C/C+ programs [101]. Our choice for TCE is justified by the fact that the analyzed apps do not have test suites that can be used for mutation analysis. However, we can not claim that we are not detecting all the equivalent and duplicate mutants generated by the analyzed tools.

Threats to *internal validity* concern factors internal to our settings that could have influenced our results. This is, in particular, related to the 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 generalization 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$_2$, RQ$_3$, and RQ$_4$,** on the set of 68 apps considered [6]. Also, although we compared the proposed tools with several state-of-the-art mutation tools (six tools in **RQ$_1$** and three tools in **RQ$_2$ and RQ$_3$**), our results may not generalize to tools not included in the study.

8 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 composed of two mutation testing tools, `MDroid+` and `MutAPK`, to automatically seed mutations in Android apps at source code and APK level. To validate the taxonomy and our tools, we conducted a comparative study with existing Java and Android mutation tools. The study results show that the proposed operators are more representative of Android faults than other catalogs of mutation operators, including both Java and Android-specific operators previously proposed. Also, `MDroid+` and `MutAPK` are able (in general) to outperform state-of-the-art tools in terms of non-compilable and trivial mutants. Mutating at APK level with `MutAPK` is faster and reduces the proportion of non-compilable mutants, but it also requires a higher number of mutants than when working on the source code.

The obtained results make our taxonomy and our tools ready to be used and possibly extended by other researchers/practitioners. To this aim, `MDroid+`, `MutAPK` and the wrappers for using Major and Pit with Android apps are available as open source projects [30], [31], [105], [106]. Future work will extend `MDroid+` and `MutAPK` by implementing more operators, and creating a framework for mutation analysis. Also, we plan to experiment with `MDroid+` and `MutAPK` in the context of test case prioritization and mutation-driven test cases generation.

Acknowledgements

Escobar-Velásquez and Linares-Vásquez are partially supported by a Google Latin American Research Award 2018-2020. This work was also partially supported by the NSF CCF-1815186 and CNS-1815336 grants. Bavota acknowledges the financial support of the Swiss National Science Foundation for the project CCQR (SNF Project No. 175513). Any opinions, findings, and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

References

[1] Google, “Google play. https://play.google.com/store,” 2017.
[2] Apple, “App store. https://itunes.apple.com/us/genre/ios/id36?mt=8,” 2017.
[3] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, “Reran: Timing- and touch-sensitive record and replay for android,” in *ICSE 2013*, 2013.
[4] Y. Hu, T. Azim, and I. Neamtiu, “Versatile yet lightweight record-and-replay for android,” ser. OOPSLA 2015. ACM.
[5] A. Machiry, R. Tahiliani, and M. Naik, “Dynodroid: an input generation system for android apps,” in *ESEC/FSE 2013*.
[6] S. Roy Choudhary, A. Gorla, and A. Orso, “Automated test input generation for android: Are we there yet? (E),” in *ASE 2015*.
[7] M. L. Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, “Mining android app usages for generating actionable gui-based execution scenarios,” in *MSR 2015*.
[8] K. Mao, M. Harman, and Y. Jia, “Sapienz: Multi-objective automated testing for android applications,” ser. ISSTA 2016.

[9] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *ICST 2016*.
[10] S. Liñán, L. Bello-Jiménez, M. Arévalo, and M. Linares-Vásquez, "Automated extraction of augmented models for android apps," in *ICSME 2018*.
[11] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, Sep. 2015.
[12] "Perfecto. http://www.perfectomobile.com," 2017.
[13] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," ser. ICSME 2017.
[14] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Software Eng.*, vol. 3, no. 4, Jul. 1977.
[15] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*.
[16] Y. Ma, Y. R. Kwon, and J. Offutt, "Inter-class mutation operators for java," in *ISSRE 2002*, 2002.
[17] S. Kim, J. A. Clark, and J. A. McDermid, "Investigating the effectiveness of object-oriented testing strategies using the mutation method," *Softw. Test., Verif. Reliab.*, 2001.
[18] K. H. Moller and D. J. Paulish, "An empirical investigation of software fault distribution," ser. METRIC 1993, May 1993.
[19] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, Jul. 2002.
[20] H. Zhang, "On the distribution of software faults," *IEEE Trans. Software Eng.*, vol. 34, no. 2, 2008.
[21] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," ser. MSR 2013.
[22] B. Robinson and P. Brooks, "An initial study of customer-reported GUI defects," ser. ICSTW 2009, April 2009.
[23] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *FSE 2014*, 2014.
[24] Q. Luo, K. Moran, D. Poshyvanyk, and M. D. Penta, "Assessing test case prioritization on real faults and mutants," ser. ICSME 2018.
[25] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of android apps," in *ICSTW 2015*, April 2015.
[26] R. Jabbarvand and S. Malek, "mudroid: An energy-aware mutation testing framework for android," ser. ESEC/FSE 2017.
[27] H. Coles, "Pit. http://pitest.org/," 2017.
[28] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for java," in *ISSTA 2014*, 2014.
[29] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: an efficient and extensible tool for mutation analysis in a java compiler," in *ASE 2011*, 2011.
[30] SEMERU, "Mdroid+ repo at github https://gitlab.com/SEMERU-Code-Public/Android/Mutation/MDroidPlus."
[31] The Software Design Lab., "Mutapk repo at github https://github.com/TheSoftwareDesignLab/MutAPK."
[32] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling mutation testing for android apps," ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017.
[33] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI ripping for automated testing of android applications," in *ASE 2012*, 2012.
[34] S. Hao, B. Liu, S. Nath, W. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *MobiSys 2014*, 2014.
[35] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: segmented evolutionary testing of android apps," in *FSE 2014*.
[36] M. Linares-Vásquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" ser. ICSME 2017, Sep. 2017.
[37] S. Zein, N. Salleh, and J. Grundy, "A systematic mapping study of mobile application testing techniques," *J. Syst. Softw.*, 2016.
[38] A. Derezińska and K. Hałas, *Analysis of Mutation Operators for the Python Language*. Cham: Springer International Publishing, 2014.
[39] U. Praphamontripong, J. Offutt, L. Deng, and J. Gu, "An experimental evaluation of web mutation operators," in *ICSTW 2016*.
[40] D. Rodríguez-Baquero and M. Linares-Vásquez, "Mutode: generic javascript and node. js mutation testing tool," ser. ISSTA 2018, 2018.
[41] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient javascript mutation testing," ser. ICST 2013.
[42] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for SQL injection vulnerabilities: an input mutation approach," in *ISSTA 2014*, 2014.
[43] C. Zhou and P. G. Frankl, "Mutation testing for java database applications," ser. ICST 2009, 2009.
[44] R. A. P. Oliveira, E. Alégroth, Z. Gao, and A. Memon, "Definition and evaluation of mutation operators for GUI-level mutation analysis," in *ICSTW 2015*, 2015.
[45] D. Di Nardo, F. Pastore, and L. C. Briand, "Generating complex and faulty test data through model-based mutation analysis," in *ICST 2015*, 2015.
[46] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, Sept 2011.
[47] A. J. Offutt and S. D. Lee, "How strong is weak mutation?" in *TAV 1991*, 1991.
[48] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ICSE 2005*.
[49] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, , and A. Groce, "Measuring effectiveness of mutant sets," in *ICSTW 2016*.
[50] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis?" in *ICST 2012*, 2012.
[51] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *GECCO 2004*.
[52] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *ISSTA 2014*.
[53] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "On the limits of mutation reduction strategies," ser. ICSE 2016, 2016.
[54] D. Shin and D.-H. Bae, "A theoretical framework for understanding mutation-based testing methods," in *ICST 2016*, 2016.
[55] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: An automated class mutation system," *Softw. Test., Verif. Reliab.*, vol. 15, no. 2, Jun. 2005.
[56] I. Moore, "Jester - the junit test tester." 2017, http://goo.gl/cQZ0L1.
[57] R. Two, "Jumble," http://jumble.sourceforge.net.
[58] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for java," ser. ESEM/FSE 2009, 2009.
[59] Yuan-W, "mudroid project at github," 2017, https://goo.gl/sQo6EL.
[60] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing android apps," *Information and Software Technology*, vol. 81, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916300684
[61] H. Coles, "Mutation testing systems for java compared," 2017, http://pitest.org/java_mutation_testing_systems/.
[62] L. Madeyski and N. Radyk, "Judy - a mutation testing tool for Java," *IET Software*, vol. 4, no. 1, Feb 2010.
[63] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," ser. ISSTA 1996, 1996.
[64] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Software Eng.*, vol. 32, no. 8, 2006.
[65] G. Petrovic and M. Ivankovic, "State of mutation testing at google," ser. SEIP 2017, 2018.
[66] A. Ghanbari and L. Zhang, "Practical program repair via bytecode mutation," *arXiv preprint arXiv:1807.03512*.
[67] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *IST*, vol. 88, 2017.
[68] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile apps," in *ESEM 2013*.
[69] M. Nagappan and E. Shihab, "Future trends in software engineering research for mobile apps," ser. SANER 2016), 2016.
[70] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *ICSM 2003*.
[71] M. Linares-Vásquez, B. Dit, and D. Poshyvanyk, "An exploratory analysis of mobile development issues using stack overflow," ser. MSR 2013, May 2013.

[72] S. Beyer and M. Pinzger, "A manual categorization of android app development issues on stack overflow," in *ICSME 2014*, Sept 2014.
[73] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, vol. 32, no. 3, 2015.
[74] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using stack overflow," *Empirical Software Engineering*, vol. 21, no. 3, 2016.
[75] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code: An extended study in the mobile application domain," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, Sep. 2014.
[76] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen, "Unveiling exception handling bug hazards in android based on github and google code issues," ser. MSR 2015, 2015.
[77] "Online appendix/replication package for "enabling mutation testing for android apps". http://android-mutation.com/fse-appendix," 2017.
[78] L. Ravindranath, S. nath, J. Padhye, and H. Balakrishnan, "Automatic and scalable fault detection for mobile applications," in *MobiSys 2014*.
[79] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *ICST 2014*, 2014.
[80] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao, "Caiipa: Automated large-scale mobile app testing through contextual fuzzing," ser. MobiCom 2014, 2014.
[81] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *ISSTA 2015*, 2015.
[82] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Auto-completing bug reports for android applications," in *ESEC/FSE 2015*, 2015.
[83] D. Pagano and W. Maalej, "User feedback in the appstore: An empirical study," in *RE 2013*, 2013.
[84] C. Iacob and R. Harrison, "Retrieving and analyzing mobile apps feature requests from online reviews," in *MSR 2013*, 2013.
[85] S. Panichella, A. Di Sorbo, E. Guzman, C. Visaggio, G. Canfora, and H. Gall, "How can i improve my app? classifying user reviews for software maintenance and evolution," in *ICSME 2015*, 2015.
[86] F. Palomba, M. Linares-Vasquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "User reviews matter! tracking crowdsourced reviews to support evolution of successful apps," in *ICSME 2015*, 2015.
[87] M. L. Vásquez, C. Vendome, Q. Luo, and D. Poshyvanyk, "How developers detect and fix performance bottlenecks in android apps," in *ICSME 2015*, 2015.
[88] N. Chen, J. Lin, S. Hoi, X. Xiao, and B. Zhang, "AR-Miner: Mining informative reviews for developers from mobile app marketplace," in *ICSE 2014*, 2014.
[89] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, "Release planning of mobile apps based on user reviews," in *ICSE 2016*, 2016.
[90] N. Chen, S. C. Hoi, S. Li, and X. Xiao, "Simapp: A framework for detecting similar mobile applications by online kernel learning," ser. WSDM 2015. ACM, 2015.
[91] M. B. Miles, A. M. Huberman, and J. Saldaña, *Qualitative Data Analysis: A Methods Sourcebook*, 3rd ed. SAGE Publications, Inc, Apr 2013.
[92] Y. Lin, S. Okur, and D. Dig, ser. ASE 2015.
[93] "Android mutation web appendix https://thesoftwaredesignlab.github.io/android-mutation/."
[94] Apktool. https://code.google.com/p/android-apktool/.
[95] P. Favre-Bulle, "Uber apk signer https://github.com/patrickfav/uber-apk-signer."
[96] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Softw. Test., Verif. Reliab.*, vol. 7, no. 3, 1997.
[97] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, "Does choice of mutation tool matter?" *Software Quality Journal*, vol. 25, no. 3, Sep. 2017. [Online]. Available: https://doi.org/10.1007/s11219-016-9317-7
[98] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," ser. ICSE 2015, vol. 1, May 2015.
[99] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification and Reliability*, vol. 4, no. 3, 1994. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.4370040303
[100] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, April 2018.
[101] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," ser. ICSE 2014. New York, NY, USA: ACM, 2014.
[102] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures.*, second edition ed. Chapman & Hall/CRC.
[103] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal on Statistics*, vol. 6, 1979.
[104] J. M. Zhang, L. Zhang, D. Hao, L. Zhang, and M. Harman, "An empirical comparison of mutant selection assessment metrics," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April.
[105] SEMERU, "Android wrapper for the major tool https://gitlab.com/SEMERU-Code-Public/Android/Mutation/Android-Major-Wrapper."
[106] ——, "Android wrapper for the pit tool https://gitlab.com/SEMERU-Code-Public/Android/Mutation/Android-Pit-Wrapper."

Camilo Escobar-Velásquez is a Ph.D. student at Universidad de los Andes in Colombia. He received his M.S. in Software Engineering from Universidad de los Andes in 2019. He received his B.S. in Systems and Computing Engineering - Minor: Mathematics from Universidad de los Andes in 2017. He is part of The Software Design Lab, where he has been part of research projects around evolution, maintenance and analysis of closed-source android apps, automation of software engineering tasks and software testing. He received a Google Latin American Research Award in 2018-2020. He served as a student volunteer for ICSME'2018, ICSE'2019 and ASE'2019. More information is available at http://caev03.github.io.

Mario Linares-Vásquez is an Assistant Professor at Universidad de los Andes in Colombia. He received his Ph.D. degree in Computer Science from the College of William and Mary in 2016. He received his B.S. in Systems Engineering from Universidad Nacional de Colombia in 2005, and his M.S. in Systems Engineering and Computing from Universidad Nacional de Colombia in 2009. He is the leader of The Software Design Lab, which is focused on automated software engineering, mining software repositories, application of data mining and machine learning techniques to support software engineering tasks, design for everyone, and app development with societal impact. He received four ACM SIGSOFT distinguished paper awards, and the best paper award at ICSM'13. He has served as organizing and program committee member of international conferences in the field of software engineering, such as ICSE, ASE, ICSME, MSR, SANER, ICPC, SCAM, MOBILESOFT and others. Mario is member of the editorial board of the Journal of Systems and Software and the Information and Software Technology Journal.

Gabriele Bavota is an Assistant Professor at the Università della Svizzera italiana (USI), Switzerland. He received the PhD degree in computer science from the University of Salerno, Italy, in 2013. His research interests include software maintenance, empirical software engineering, and mining software repository. He is the author of over 120 papers appeared in international journals, conferences and workshops. He received five ACM SIGSOFT Distinguished Paper awards at ASE 2013, ESEC-FSE 2015, ICSE 2015, ASE 2017, and MSR 2019, an IEEE TCSE Distinguished Paper Award at ICSME 2018, the best paper award at SCAM 2012, and three distinguished reviewer awards at WCRE 2012, SANER 2015, and MSR 2015. He is the recipient of the 2018 ACM Sigsoft Early Career Researcher Award. He served as a Program Co-Chair for ICPC'16, SCAM'16, and SANER'17. He also serves and has served as organizing and program committee member of international conferences in the field of software engineering, such as ICSE, FSE, ASE, ICSME, MSR, SANER, ICPC, SCAM, and others

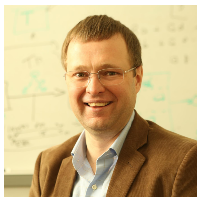
Michele Tufano received his Ph.D. in Computer Science from The College of William & Mary in May 2019. He received a B.S. in Computer Science from the University of Salerno in 2012 and his M.S. in Computer Science from the University of Salerno in 2014. His research interests include Deep Learning applied to Software Engineering, Automated Program Repair, Software Evolution and Maintenance, Mining Software Repositories, and Android Testing. He received an ACM SIGSOFT Distinguished Paper Award at ICSE 2015. More information available at: https://tufanomichele.com/.

Kevin Moran is currently a Research Assistant Professor in the Computer Science Department at the College of William & Mary and a senior member of the SEMERU research group. He graduated with a B.A. in Physics from the College of the Holy Cross in 2013 and an M.S. degree from William & Mary in August of 2015. He received his Ph.D. degree from William & Mary in August 2018. His main research interests involve facilitating the processes of software engineering, maintenance, and evolution with a focus on mobile platforms. He has published in several top peer-reviewed venues including: ICSE, ESEC/FSE, TSE, USENIX Security, ICST, ICSME, and MSR. He was recognized as the second-overall graduate winner in the ACM Student Research competition at ESEC/FSE 2015, received the best paper award at CODASPY 2019, and an ACM SIGSOFT distinguished paper award at ESEC/FSE'19. Moran has served on the organizing committees of MobileSOFT 2019 and ICSME 2019. More information is available at http://www.kpmoran.com.

Massimiliano Di Penta is a full professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and service-centric software engineering. He is an author of over 270 papers appeared in international journals, conferences, and workshops. He serves and has served in the organizing and program committees of more than 100 conferences, including ICSE, FSE, ASE, ICSME. He is in the editorial board of the Empirical Software Engineering Journal edited by Springer, ACM Transactions on Software Engineering and Methodology, and of the Journal of Software: Evolution and Processes edited by Wiley, and has served the editorial board of the IEEE Transactions on Software Engineering.

Christopher Vendome is an Assistant Professor at Miami University in Oxford, Ohio. He received a B.S. in Computer Science from Emory University in 2012 and he received his M.S. in Computer Science from The College of William & Mary in 2014. He received his PhD degree in Computer Science from the College of William & Mary in 2018. His main research areas are software maintenance and evolution, mining software repositories, program comprehension, software provenance, and software licensing. He has received an ACM Distinguished Paper Award at ASE 2017. More information is available at http://www.christophervendome.com

Carlos Bernal-Cárdenas is currently Ph.D. candidate in Computer Science at the College of William & Mary as a member of the SEMERU research group advised by Dr Denys Poshyvanyk. He received the B.S. degree in systems engineering from the Universidad Nacional de Colombia in 2012 and his M.E. in Systems and Computing Engineering in 2015. His research interests include software engineering, software evolution and maintenance, information retrieval, software reuse, mining software repositories, mobile applications development. He has published in several top peer-reviewed software engineering venues including: ICSE, ESEC/FSE, ICST, and MSR. He has also received the ACM SIGSOFT Distinguished paper award at ESEC/FSE'15 and ESEC/FSE'19. Bernal-Cárdenas is a student member of IEEE and ACM and has served as an external reviewer for ICSE, ICSME, FSE, APSEC, and SCAM. More information is available at http://www.cs.wm.edu/~cebernal/.



Denys Poshyvanyk is the Class of 1953 Term Distinguished Associate Professor of Computer Science at the College of William and Mary in Virginia. He received the MS and MA degrees in Computer Science from the National University of Kyiv-Mohyla Academy, Ukraine, and Wayne State University in 2003 and 2006, respectively. He received the PhD degree in Computer Science from Wayne State University in 2008. He served as a program co-chair for MobileSoft'19, ICSME'16, ICPC'13, WCRE'12 and WCRE'11. He currently serves on the editorial board of IEEE Transactions on Software Engineering (TSE), Empirical Software Engineering Journal (EMSE, Springer) and Journal of Software: Evolution and Process (JSEP, Wiley). His research interests include software engineering, software maintenance and evolution, program comprehension, reverse engineering, software repository mining, source code analysis and metrics. His research papers received several Best Paper Awards at ICPC'06, ICPC'07, ICSM'10, SCAM'10, ICSM'13 and ACM SIGSOFT Distinguished Paper Awards at ASE'13, ICSE'15, ESEC/FSE'15, ICPC'16, ASE'17 and ESEC/FSE'19. He also received the Most Influential Paper Awards at ICSME'16 and ICPC'17. He is a recipient of the NSF CAREER award (2013). He is a member of the IEEE and ACM. More information available at: http://www.cs.wm.edu/~denys/.