MDroid+: A Mutation Testing Framework for Android

Kevin Moran
College of William and Mary
United States

Michele Tufano
College of William and Mary
United States

Carlos Bernal-Cárdenas
College of William and Mary
United States

Mario Linares-Vásquez
Universidad de los Andes
Colombia

Gabriele Bavota
Università della Svizzera italiana
Switzerland

Christopher Vendome
College of William and Mary
United States

Massimiliano Di Penta
University of Sannio
Italy

Denys Poshyvanyk
College of William and Mary
United States

ABSTRACT

Mutation testing has shown great promise in assessing the effectiveness of test suites while exhibiting additional applications to test-case generation, selection, and prioritization. Traditional mutation testing typically utilizes a set of simple language specific source code transformations, called operators, to introduce faults. However, empirical studies have shown that for mutation testing to be most effective, these simple operators must be augmented with operators specific to the domain of the software under test. One challenging software domain for the application of mutation testing is that of mobile apps. While mobile devices and accompanying apps have become a mainstay of modern computing, the frameworks and patterns utilized in their development make testing and verification particularly difficult. As a step toward helping to measure and ensure the effectiveness of mobile testing practices, we introduce MDroid+, an automated framework for mutation testing of Android apps. MDroid+ includes 38 mutation operators from ten empirically derived types of Android faults and has been applied to generate over 8,000 mutants for more than 50 apps.
Video URL: https://youtu.be/yzE5_-zN5GA

CCS CONCEPTS

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

KEYWORDS

Mutation testing, Operators, Android

ACM Reference Format:
Kevin Moran, Michele Tufano, Carlos Bernal-Cárdenas, Mario Linares-Vásquez, Gabriele Bavota, Christopher Vendome, Massimiliano Di Penta, and Denys Poshyvanyk. 2018. MDroid+: A Mutation Testing Framework for Android. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183440.3183492

1 INTRODUCTION

Mobile devices have become a mainstay of the modern computing landscape. The increasing popularity of these devices is driven primarily by a rich ecosystem of applications, commonly referred to as "apps". Strong user demand for mobile apps has driven increased competition on storefronts such as Google Play [10] and Apple's App Store [4]. These competitive marketplaces necessitate that mobile developers devote time to extensively test and validate their apps to ensure a positive user experience, as new app releases have been shown to affect users' perception via marketplace ratings [21]. Android is currently the most popular OS in the world [1], and hence garners a large amount of interest from developers.

However, effectively testing Android apps is difficult, as developers face constraints specific to the domain of mobile software including change-prone APIs [5, 16], platform fragmentation [12], and a lack of automated testing tools that meet developer needs [19]. Furthermore, when devising testing strategies or writing test cases, their effectiveness needs to be evaluated. To this aim, mutation analysis has been defined as a process for evaluating the efficacy of a test suite and functions by injecting small changes, meant to represent common bugs, into a "correct" program and then evaluating the number of injected faults uncovered by the test suite [8, 11]. The higher the ratio of uncovered bugs, or *mutants*, the more effective a test suite is said to be. However, mutation analysis typically relies on a concept called the *coupling effect* that states that *"complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults"* [23]. The validity of this effect has come under scrutiny from the software testing research community [2, 3, 7, 15, 20], and while studies have generally indicated a correlation between mutants and real faults, they also point out that non-negligible portion of real faults do not effectively map to mutants [15]. The number of simple mutants mapping to real faults in the mobile domain is likely to be lower, due to heavy usage of APIs and frameworks that enable the varied feature sets of mobile devices. Thus, in order to develop an effective mutation testing tool for Android, one must take into account the software domain, and model operators according to faults that naturally occur in mobile software development.

In this paper, we describe MDroid+, a mutation testing framework for Android apps that aims to support developers in writing mobile tests. MDroid+ includes 38 Android and Java specific mutation operators that were designed according to an empirically derived taxonomy of common, naturally occurring faults in Android applications [17]. The main contributions of MDroid+ can be summarized as follows:

- A set of empirically derived mutation operators designed to simulate commonly observed faults in Android apps;
- An efficient, automated methodology for deriving a Potential Fault Profile (PFP) from subject apps, dictating possible locations for mutant injection;
- A process for applying operator transformations to locations dictated by the PFP to create mutant apps;
- Built-in extensibility that facilitates new operator definitions.

2 APPROACH

In order to ensure that MDroid+ is an effective, practical, and flexible/extensible tool for mutation testing, it takes 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. MDroid+ is written in Java and available as an open source project [18]. In the following sections, we describe MDroid+ according to its workflow described in Figure 1.

2.1 Implemented Operators

As mentioned in Section 1, in order to create a mutation testing framework for Android, mutation operators must be defined according to naturally occurring faults to ensure a strong coupling between operators and faults likely to befall an Android project. To this end, the authors undertook an extensive empirical study following a procedure inspired by open coding [22]. More specifically, the authors analyzed 2,007 documents, assigning a label describing an observed bug, from the following sources: (i) bug reports of open source Android apps, (ii) bug-fixing commits of open source Android apps, (iii) Stack-Overflow discussions, (iv) exception hierarchies of Android APIs, (v) bugs described in previous studies, and (vi) reviews from Google Play. The open-coding procedure was supported by a custom designed web-application, where each document was tagged by at least two authors, and labeling conflicts were resolved. For the complete methodology, we refer readers to our previous technical paper [17].

The study described above resulted in a taxonomy of faults for Android apps that spans 14 different categories. We implemented 38 operators corresponding to ten different categories, excluding certain categories of operators such as hardware configurations, as they do not map well to the process of mutation testing. A detailed list and description of all implemented operators is available on the MDroid+ website [18].

2.2 Derivation of the Potential Fault Profile

In the context of MDroid+, we define a PFP that stipulates locations in analyzed apps—which can correspond to source code statements, XML tags, or locations in other resourcefi les—that coincide with potential fault locations, given the empirically derived taxonomy [17]. Consequently, the PFP stipulates the locations where our defined mutation operators can be applied. To extract the PFP, MDroid+ statically analyzes a target mobile app, looking for locations where

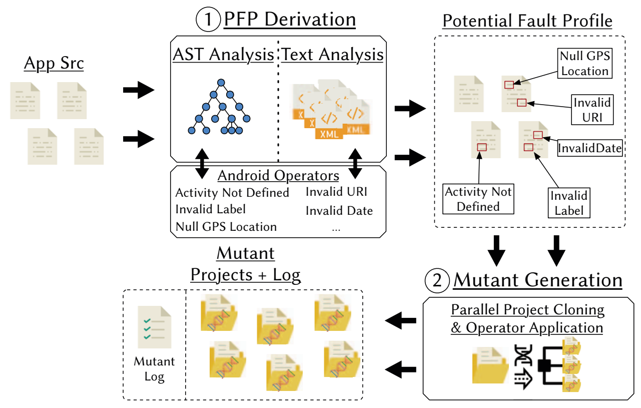

Figure 1: Overview tool Workflow for MDroid+

operators can be implemented. The locations are detected automatically by parsing XMLfi les in the case of resources or through Abstract Syntax Tree (AST) based analysis for Java code.

For implemented operators targeting one of the Android resource XMLfi les (*i.e.,* archives in the `/res` folder), the structure of each XMLfi le is analyzed and a pattern matching process for different attributes within the XML is used. However, for operators that are applied to the Java source code, a two-phase AST-based and text-based analysis is utilized that is capable of identifying the location of target API calls. The identification of API call sites is implemented utilizing the visitor design pattern, allowing for extensible, decoupled operations to be performed on the AST of a target app. This helps to ensure that MDroid+ is capable of supporting additional operators in the future that may require more advanced AST analysis. After the AST-based location of specific API calls, afi ne-grained, text-based pattern matching is performed on identified API calls to derive the precise textual location where the mutation operator transformation will be applied. The end result of the PFP derivation process is a list stipulating all potential injection points in code of the Android-specific mutation operators.

2.3 Mutant Creation

Given an automatically derived PFP for an app and the catalog of Android-specific operators, MDroid+ generates a mutant for each location in the PFP. This process is performed using text or AST manipulation rules specific to each implemented operator. Thus, for each location related to an operator, the text/AST transformation is applied to the specified location in either the code or `.xml` file.

Due to the event driven nature of Android applications, testing is generally performed at the GUI-level and is centered around app use-cases [19]. Therefore, in order to operationalize MDroid+ to fit in a typical testing workflow, the mutated applications must be compilable to an `.apk` file that can be run on an emulator or real device. Thus, during the mutant creation process, MDroid+ creates a project-level clone of a target and applies a single mutation to a specified location in the cloned project, resulting in one mutant project for each seeded instance of a mutation operator. It is important to note that the cost of applying the transformations to the cloned projects is trivial in practice, and a majority of the execution time cost is related to the cloning of the app project, and is thus I/O bound. To ensure the cloning and mutant generation process proceeds as efficiently as possible, MDroid+ implements an option to parallelize the process, utilizing the multi-core architecture of most modern hardware. The end result of the mutation creation process is the set of cloned mutant projects and a log that delineates the operator and location of injection for each created mutant. Currently, we do not offer a universal interface for the compilation of mutants, as the parameters and build systems used by Android apps can vary dramatically. However, the compilation process can be easily scripted using the CLI support offered by `ant`, `gradle`, and the Android SDK[1].

2.4 Tool Usage and Extensibility

MDroid+ is implemented as a Java command-line utility in which the user can select the specific set of mutation operators to be applied during mutant generation, as well as an option for enabling multi-threading. A README and user guide can be found on the project repository, which is accessible from the tool website [18].

Given that the Android platform is prone to rapid evolution [5, 16], it is important that MDroid+ allows for easy modification/extension of the operators list, in order to keep pace with rapid evolution. To add a new operator to MDroid+, there are two major components that must be implemented: (i) an operator locator/detector, and (ii) the operator transformation rules. If a proposed operator applies to the source code of an application, then either the target API-call of the operator must be added to an existing AST visitor or a new visitor to identify the AST pattern required by the operator must be defined. Next, an operator specific pattern locator must be implemented to derive the precise operator location after API-calls in question have been detected via AST-analysis. To allow for streamlined extensibility, MDroid+ offers a generic `Locator` interface that includes a `findExactLocations()` method that can be implemented to support additional operators. An example of this interface implemented for the `BuggyGUIListener` operator is given in Listing 1. This locator simply returns the start and end of the API-call location passed in the `MutationLocation` object so that it can be manipulated later. Note, that this is a simple example, and most of MDroid+'s pre-defined operators need to parse particular properties or patterns from identified API calls. If the proposed operator applies to the resource `.xml`, MDroid+ implements a `TextBasedDetector` abstract class that can be extended to account for matching the patterns of xml attributes.

Listing 1: Example of Operator Locator

```java
public class BuggyGUIListenerLocator implements Locator {

    private void findExactLocation(MutationLocation loc) {
        //Fix start column
        loc.setStartColumn(loc.getStartColumn()+1);
        //Build exact mutation location
        loc.setEndColumn(loc.getStartColumn()+loc.
            getLength());
    }
}
```

Once the precise locator/detector for a proposed mutation operator has been implemented, the actual transformation rule must be delineated so that it can be applied to any detected injection points during the MDroid+ analysis. To facilitate this process, MDroid+ includes a `MutationOperator` interface and implements the factory design pattern for managing and instantiating operators. To add an additional transformation rule for a desired operator, the `performMutation()` method of the interface must be implemented according to the `MutationLocation`. An example of the `NullInputStream` operator that sets an `inputStream` to null before it is closed is shown in Listing 2.

Listing 2: Example of Operator Definition

```java
public class NullStream  implements MutationOperator{
    @Override
    public boolean performMutation(MutationLocation
         location) {
        ObjectMutationLocation mLocation = (
             ObjectMutationLocation) location;
        List<String> newLines = new ArrayList<String>();
        List<String> lines = FileHelper.readLines(
             location.getFilePath());
        for(int i=0; i < lines.size(); i++){
            String currLine = lines.get(i);
            //Null object
            if(i == location.getLine()){
                String newLine = mLocation.getObject() +
                     " = null;";
                newLines.add(newLine);
            }
            newLines.add(currLine);
        }
        FileHelper.writeLines(location.getFilePath(),
             newLines);
        return true;
    }
}
```

3 EVALUATION

3.1 Study Context

To evaluate MDroid+, we conducted a study with the following *goals*: (i) understand and compare the *applicability* of MDroid+ and other currently available mutation testing tools to Android apps; (ii) to understand the *underlying reasons* for non-compilable or non-executable (*i.e., trivial*) mutants. To accomplish this, we compare MDroid+ with two popular open source mutation testing tools (Major [14] and PIT [6]), which we adapted to be applicable to Android apps, and with one context-specific mutation testing tool for Android called muDroid [9]. We chose these tools because of their diversity (in terms of functionality and mutation operators), their compatibility with Java, and their representativeness of tools working at different representation levels: source code, Java bytecode, and smali bytecode (*i.e.,* Android-specific bytecode representation). The modifications to the PIT and Major tools can be accessed using our replication package [18]. To compare the applicability of each mutation tool, we used the Androtest suite of apps [24], which includes 68 Android apps from 18 Google Play categories. The mutation testing tools exhibited issues in 13 of the considered 68 apps, *i.e.,* the mutant-injection process rendered them non-compilable. Thus, in the end, we considered 55 subject apps in our study. For more information see our replication package [18].

To quantitatively assess the applicability and effectiveness of the considered mutation tools to Android apps, we defined three metrics: *Total Number of Generated Mutants* **(TNGM)**, *Non-Compilable Mutants* **(NCM)**, and *Trivial Mutants* **(TM)**. In this paper, we consider *non-compilable mutants* as those that are syntactically incorrect and cause compilation/assembly errors, and *trivial mutants* as those that are killed arbitrarily by most test cases (*e.g.,* crashing on launch). The trivial mutant study was supported by a large-scale dynamic analysis framework [17].

[1]https://developer.android.com/studio/build/building-cmdline.html

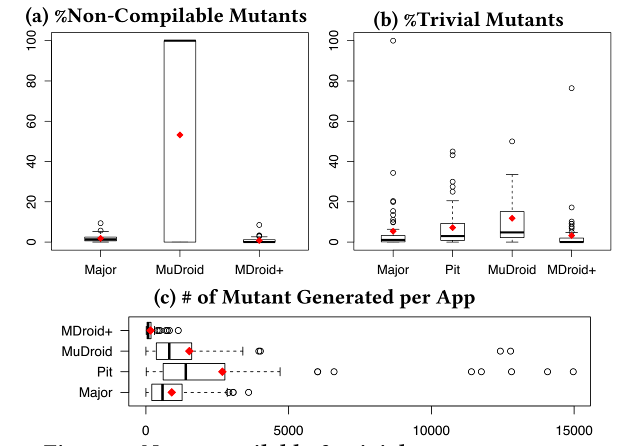

Figure 2: Non-compilable & trivial mutants per app.

3.2 Study Results

Figure 2 reports the results of (i) the percentage of non-compilable mutants (NCM), (ii) the percentage of trivial mutants (TM), and (iii) the total number of generated mutants per app. On average, 167, 904, 2.6k+, and 1.5k+ mutants were generated by MDroid+, Major, PIT, and muDroid, respectively for each app. MDroid+ had an average runtime of 19 seconds per app on server-class hardware. While the number of mutants generated is high for other tools, this is mostly due to the fact they implement far more general operators with a higher number of potential injection points. However, MDroid+'s operators are more specific, and coupled to features that may or may not be implemented by a given app, resulting in fewer, but stronger, mutants. The average percentage of *non-compilable mutants* (NCM) generated by MDroid+, Major and muDroid over all the apps is 0.56%, 1.8%, and 53.9%, respectively, while no NCM are generated by PIT due to its mutant assembly process (Figure 2a). Thus, MDroid+ achieves the lowest ratio of non-compilable mutants by a statistically significant margin (Wilcoxon paired signed rank test p-value< 0.001 for Major and muDroid – adjusted with Holm's correction [13], Cliff's d=0.59 - large for Major, and Cliff's d=0.35 - medium for muDroid), allowing for a mutation testing procedure more capable of evaluating the efficacy of a set of tests . The overall rate of NCM is very low for MDroid+, and most instances pertain to edge cases requiring more robust static analyses.

All four tools generated *trivial mutants*, and the mean percentage of their distribution over all apps for MDroid+, Major, PIT, and muDroid is 2.42%, 5.4%, 7.2%, and 11.8%, respectively (Figure 2b). MDroid+ generates significantly fewer TM than muDroid (Wilcoxon paired signed rank test adjusted p-value=0.04, Cliff's d=0.61 - large) and PIT (adjusted p-value=0.004, Cliff's d=0.49 - large), while there is no statistically significant difference with Major (adjusted p-value=0.11). While these percentages may appear small, the raw values show that TMs 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. The biggest underlying reason for TM in MDroid+ results from null objects or references in the `MainActivity`, causing a crash on startup. Future improvements to the tool could avoid mutants seeded in components related to the MainActivity.

4 DEMO REMARKS & FUTURE WORK

In this demo, we have presented MDroid+, a mutation testing tool for Android applications that supports 38 empirically derived mutation operators, automates the process of detecting potential mutant locations and generating mutants, and facilitates the addition of new operators and the maintenance of existing operators through an extensible architecture. MDroid+ was evaluated against other popular mutation testing tools for the Java language and was shown to generate fewer non-compilable and trivial mutants. In the future, we plan to add to the functionality of MDroid+ by allowing users to stipulate specific activities for mutant injection and supporting the increasingly popular Kotlin language.

Acknowledgments. Bavota was supported in part by the SNF project JITRA, No. 172479. We would like to thank Sebastián Sierra and Camilo Escobar from Universidad de los Andes for their help testing MDroid+.

REFERENCES

[1] Google's Android just beat Microsoft Windows as most popular OS http://fortune.com/2017/04/03/google-android-microsoft-os/, 2017.
[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE'05*, pages 402–411, 2005.
[3] 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.*, 32(8):608–624.
[4] Apple. App store. https://itunes.apple.com/us/genre/ios/id36?mt=8, 2017.
[5] G. Bavota, M. Linares-Vásquez, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. The impact of API change- and fault-proneness on the user ratings of Android apps. *IEEE Trans. Software Eng.*, 41(5):384–407, 2015.
[6] H. Coles. Pit. http://pitest.org/, 2017.
[7] M. Daran and P. Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *ISSTA'96*, pages 158–171, 1996.
[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.
[9] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt. Towards mutation analysis of Android apps. In *ICSTW'15*, pages 1–10, April 2015.
[10] Google. Google play. https://play.google.com/store, 2017.
[11] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Software Eng.*, 3(4):279–290, July 1977.
[12] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia. Understanding Android fragmentation with topic analysis of vendor-specific bugs. In *WCRE'12*, pages 83–92, 2012.
[13] S. Holm. A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal on Statistics*, 6:65–70, 1979.
[14] R. Just. The Major mutation framework: Efficient and scalable mutation analysis for java. In *International Symposium on Software Testing and Analysis, ISSTA'14.*
[15] 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 *ESEC/FSE'14.*
[16] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. API change and fault proneness: A threat to the success of Android apps. In *ESEC/FSE'13*, pages 477–487, 2013.
[17] 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. In *ESEC/FSE'17*, pages 233–244, 2017.
[18] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk. Online appendix for "enabling mutation testing for android apps". http://android-mutation.com, 2017.
[19] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. How do developers test android applications? In *ICSME'17*, pages 613–622, Sept 2017.
[20] Q. Luo, K. Moran, and D. Poshyvanyk. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 559–570, New York, NY, USA, 2016. ACM.
[21] W. Martin, F. Sarro, and M. Harman. Causal impact analysis for app releases in google play. In *ESEC/FSE'16*, pages 435–446, 2016.
[22] M. B. Miles, A. M. Huberman, and J. Saldaña. *Qualitative Data Analysis: A Methods Sourcebook*. SAGE Publications, Inc, 3rd edition, Apr 2013.
[23] A. J. Offutt and R. H. Untch. *Mutation 2000: Uniting the Orthogonal*, pages 34–44. Springer US, Boston, MA, 2001.
[24] S. Roy Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (E). In *ASE'15*, pages 429–440, 2015.