DroidMutator: An Effective Mutation Analysis Tool for Android Applications

Jian Liu
East China Normal University
China

Xusheng Xiao
Case Western Reserve University
United States

Lihua Xu
New York University
Shanghai, China

Liang Dou
East China Normal University
China

Andy Podgurski
Case Western Reserve University
United States

ABSTRACT

With the rapid growth of Android devices, techniques that ensure high quality of mobile applications (i.e., apps) are receiving more and more attention. It is well-accepted that mutation analysis is an effective approach to simulate and locate realistic faults in the program. However, there exist few practical mutation analysis tools for Android apps. Even worse, existing mutation analysis tools tend to generate a large number of mutants, hindering broader adoption of mutation analysis, let alone the remaining high number of stillborn mutants. Additionally, mutation operators are usually pre-defined by such tools, leaving users less ability to define specific operators to meet their own needs. To address the aforementioned problems, we propose DroidMutator, a mutation analysis tool specifically for Android apps with configurability and extensibility. DroidMutator reduces the number of generated stillborn mutants through type checking, and the scope of mutation operators can be customized so that it only generates mutants in specific code blocks, thus generating fewer mutants with more concentrated purposes. Furthermore, it allows users to easily extend their mutation operators. We have applied DroidMutator on 50 open source Android apps and our experimental results show that DroidMutator effectively reduces the number of stillborn mutants and improves the efficiency of mutation analysis.

Demo link: https://github.com/SQS-JLiu/DroidMutator
Video link: https://youtu.be/dtD0oTVioHM

CCS CONCEPTS

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

KEYWORDS

Android, Mutation analysis, Operators

ACM Reference Format:
Jian Liu, Xusheng Xiao, Lihua Xu, Liang Dou, and Andy Podgurski. 2020. DroidMutator: An Effective Mutation Analysis Tool for Android Applications. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377812.3382134

1 INTRODUCTION

Android applications (i.e., apps) have grown rapidly over the past decade. It has penetrated into every part of our lives and has become an integral part of people's lives. Therefore, it is very important to ensure the quality and reliability of apps. Among the technologies that ensure the quality of apps, mutation analysis is an effective approach for various software analysis tasks, such as fault localization [6, 7, 12, 13], and test generation [4, 9, 15].

Mutation analysis is a fault-based technique that measures the effectiveness of a test suite. It injects faults into the original program, and executes test cases to kill the mutants and optimizes the test suite via the non-killed mutants. Faults are introduced into the program by creating a set of faulty versions, called mutants, each of which contains one injected fault. Mutants and the original program are then executed and compared. The mutants that return different results as the original program are considered "killed". The non-killed mutants are then considered to contain the yet-to-be-exercised faults, and hence more test cases are generated to cover them.

Although well recognized, there exist very few mutation analysis tools for Android apps. To the best of authors' knowledge, muDroid [1, 16] and MDroid+ [10] are the only mutation analysis tools for Android apps. However, a number of challenges exist: First, the number of generated mutants tends to be very large, rendering many mutation analysis techniques less effective and thus hindering the wider adoption of mutation analysis. Second, injecting faults into the original program may lead to mutants with incorrect syntax, which further hinders the effectiveness of mutation analysis. Third, the quality of mutation operators, which are used to inject faults into the original program, affects the quality of mutants. We believe that not only the pre-defined general purpose operators, but the user-defined operators are also useful to generate high quality of mutants. More specifically, muDroid tends to generate a large number of stillborn mutants (i.e., mutants with incorrect syntax), while MDroid+ mutates the entire source file and cannot generate mutants within specific scopes (i.e., applying mutation operators in specific code blocks). Moreover, both of them provide little support for Java-specific mutation operators.

To address these challenges, we present a mutation analysis tool called DroidMutator. DroidMutator utilizes type checking to reduce the generation of stillborn mutants (i.e., those leading to failed compilations). It also provides configuration capability to limit the scope of mutation (i.e., applying mutation operators in specific code blocks), and allow users to define specific mutation operators to serve their own needs.

The main contributions of DroidMutator can be summarized as follows:

(1) We have developed DroidMutator, with type checking capability for variable expressions, hence reducing the number of stillborn mutants with incorrect syntax;

(2) DroidMutator configures the mutation scope to only generate mutants within the user's preferred scope, hence further reducing the number of generated overall mutants with more concentrated purposes;

(3) DroidMutator provides extension capability that allows user to easily define their own mutation operators for various purposes, especially Java-specific mutation operators;

(4) We conducted experiments with 50 open source apps. The experimental results show that the number of stillborn mutants generated by DroidMutator is only 0.1%, which is greatly improved compared to 1.7% of MDroid+. Moreover, DroidMutator mutates 5 times faster than MDroid+.

2 DROIDMUTATOR

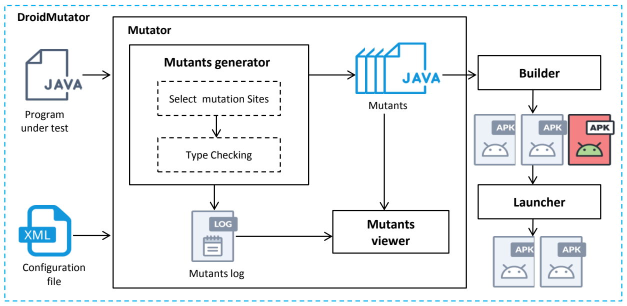


Figure. 1: Overview of DroidMutator

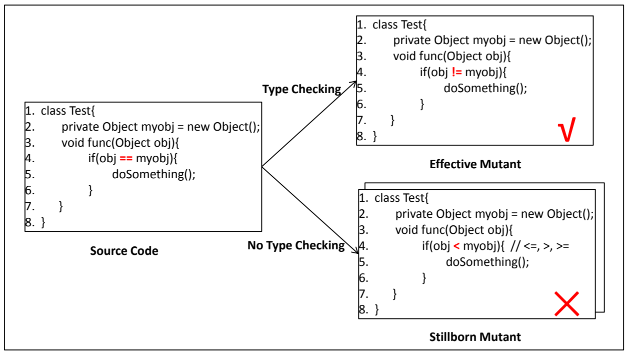
Figure 1 shows the overview of DroidMutator. DroidMutator takes the source file of an app and the mutation configuration file as input, parses the target source file, and matches the mutation location (i.e., the code blocks to be mutated) in source code file. After finding the mutation location, DroidMutator resolves the type of selected statements, aiming to assure whether mutation operators can be applied. Further, DroidMutator applies mutation operators in the chosen mutation locations to generate mutant files and change log. Finally, each mutant can be build into an installable APK (Android Application Package) file and mutants that failed compilation are discareded. Each APK file will also be launched and discarded if it crashes on launching.

DroidMutator consists of three components. The first component is *Mutator*, including two parts: *Mutants generator* and *Mutants viewer*. The *Mutants generator* first uses the JavaParser [8] to parse a source file into an abstract syntax tree, and then select feasible mutation locations. Next, it uses Java reflection and static analysis to check the types of the objects in the mutation locations. Finally, it mutates the source file to generate mutants and output the change log. The *Mutants viewer* shows the change log, so that the developers can easily inspect every mutant generated by DroidMutator.

The second component is an *Builder* to compile mutant into APK file and filter mutants that failed build. It can filter stillborn mutants whose syntax is wrong completely. Specifically, *Builder* uses Gradle [5] to build each mutant into an installable APK file.

The third component is *Launcher*, which is used to install an APK file and launch the APK file. In this way, it can filter trivial mutants that crash on app launch. After this step, the remaining mutants are the ones we need. *Launcher* is implemented as a Python component that calls the Android Emulator [2] or the physical Android device to install the mutated APK file and launch the installed APK to determine whether the mutant is working properly within a reasonably time (i.e., whether it crashes shortly after the launch). Finally, if the app crashes, it is considered a trivial mutants.

2.1 Type Checking



```
1. class Test{
2.     private Object myobj = new Object();
3.     void func(Object obj){
4.         if(obj == myobj){
5.             doSomething();
6.         }
7.     }
8. }
```

```
1. class Test{
2.     private Object myobj = new Object();
3.     void func(Object obj){
4.         if(obj != myobj){
5.             doSomething();
6.         }
7.     }
8. }
```

```
1. class Test{
2.     private Object myobj = new Object();
3.     void func(Object obj){
4.         if(obj < myobj){  // <=, >, >=
5.             doSomething();
6.         }
7.     }
8. }
```

Figure. 2: Example code for type checking

To reduce the number of generated stillborn mutants, DroidMutator includes a type checking, i.e., the *Type Checking* in Figure 1, whose job is to analyze variable types (e.g., Integer, String, Array), and return type of calling function. The types of variables that can be analyzed include Java primitive data types and object types, Android component types, and developer-defined object types. By checking the type of a variable, we can more accurately locate the location of the mutation, reduce the number of generated stillborn mutants, and improve the effectiveness of the mutation.

Figure 2 shows an example code after a mutation. In this code, the mutation operator's modification rule is to select an if conditional statement and replace relational operators (i.e., using the ROR mutation operator). First, DroidMutator will search *Source Code* statements to find the conditional statements. Further, *Type Checking* analyzes the type of variable `obj` (Line 4), which is an object type. Based on the analysis result, the relational operator (Line 4) is mutated to "!=" in the Line 4 of *Effective Mutant*, and generate a new mutant. If there is no type checking, the relational operator "==" will be replaced by ">", ">=", "<", "<=" (In *Stillborn Mutant*), which will produce more stillborn mutants.

2.2 Configurability

The MUSIC mutation tool proposed by Loc Duy Phan [14] requires the specification of the mutation scope using a starting line number and a ending line number, and the tool will only perform mutation within the scope in the source code file. However, the disadvantage of this mechanism is that the specified line numbers depends on the lines of code in a source code file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <LocationSettings control_dependence="N" active="Y">
3    <Class name="TestDemo" qualifiedName="com.example.TestDemo"
          active="Y">
4      <Methods>
5        <Method name="runFunc">
6          <Modifiers>public</Modifiers>
7          <returnType>void</returnType>
8          <Parameters></Parameters>
9        </Method>
10         ...
11     </Methods>
12   </Class>
13    ...
14 </LocationSettings>
15
```

Figure. 3: Fragment of scope configuration

Unlike existing mutation testing tools, DroidMutator provides a configurable scope of mutation for more flexible mutations. Figure 3 shows a fragment of the mutation scope configuration, and we can configure multiple methods of multiple classes to mutate. This demo configuration means that we only mutate in the runFunc() method in the class com.example.TestDemo or in the overridden runFunc() method. If we want to mutate all scopes of the target file, just set active (Line 2) to 'N'. By default, the mutating tool will mutate all mutation locations in the source code file under the target directory (or selected code files). Of course, DroidMutator can also mutate the methods we configure, or even a specific method statement. In addition, if a class inherits the configured class and overrides the method, then this overridden method will also be mutated. Thus, the scope of mutation becomes the directions to mutate a certain parts of the code, improving the efficiency of DroidMutator's mutation.

2.3 Extensibility

With the rapid development of Android apps, the existing mutation operators are often insufficient to meet the testing needs, so adding custom mutation operators is inevitable. The more effective mutation operators, the more effective the mutation test is. One of the advantages of DroidMutator is that developers can easily add their own mutation operators. Adding a mutation operator is like adding a rule to modify the source file. To help developers add their own mutation operators, DroidMutator provides an abstract class MethodLevelMutator. The developers can write a new class with MethodLevelMutator as its base class, and implement only the methods for generating mutants. Then the implemented class will become a new mutation operator and can be plugged into DroidMutator for generating mutants. Last but not least, the implemented mutation operator needs to be added to the DroidMutantsGenerator class and the configuration file.

2.4 Visibility

DroidMutator provides a GUI to the user so that the user can freely select the source file and the mutation operators for the mutation operation. After the mutation is completed, DroidMutator provides an viewer window that shows the mutation results for each of the mutated source files.

For each source file that is mutated, the viewer window displays the number of mutants generated by each mutation operator and the total number of mutants generated. Moreover, the user can use the filtering function to display all mutants of a source file or to display the mutants generated in each method separately. If a user select one of the mutants, then the GUI will display the changed statements (i.e., the mutated statements) between the source file and the mutant file, so that the user can view and analyze the mutant.

2.5 Mutation Operators

DroidMutator implements two types of mutation operators as defined in [10, 11]: *Java-specific* mutation operators and *Android-specific* mutation operators. The Java-specific mutation operators handle the primitive features of programming languages. For example, They modify expressions by replacing, adding, and deleting primitive operators. The Android-specific mutation operators handle Android-specific features such as intents, views, and locations.

In this paper, DroidMutator implements a total of 32 mutation operators, of which 28 are based on existing mutation operators, and 4 are new mutation operators proposed for our context. Due to space limit, we do not list all the mutators, but the detailed description is available at our project website[1].

3 EVALUATION

3.1 Evaluation Subjects and Setup

To evaluate the effectiveness of type checking, we randomly selected 50 open source apps from F-Droid [3] as empirical subjects. F-Droid is the largest repository for open-source Android apps.

We compare DroidMutator with a closely related tool, MDroid+ [10], to demonstrate the effectiveness of our mutant generators and the mutation operators. To demonstrate the effectiveness of type checking, we also implemented another version of DroidMutator without type checking version, called DroidMutator*, as an experimental comparison tool. Therefore, our evaluations will compare the effectiveness of MDroid+, DroidMutator, and DroidMutator*. In our evaluations, DroidMutator used the 32 mutation operators mentioned above, and used the full mutation (i.e., mutated all the positions where the mutants can be generated) for each code file.

Furthermore, to demonstrate the usability of DroidMutator, we also compare with muDroid, MDroid+ and DroidMutator.

3.2 Effectiveness

Table 1 shows the experimental results MDroid+, DroidMutator, and DroidMutator*. We mutated all Java files from 50 apps.

Table. 1: Mutant generation result

Mutation tool	Mutation Time (sec)	#Gen. Mutants	#Stillborn Mutants	#Trivial Mutants
MDroid+	5568	8900	154	213
DroidMutator*	1231	75213	26297	1030
DroidMutator	1082	48703	56	1030

*Mutating 50 apps, a total of 1304 Java files.

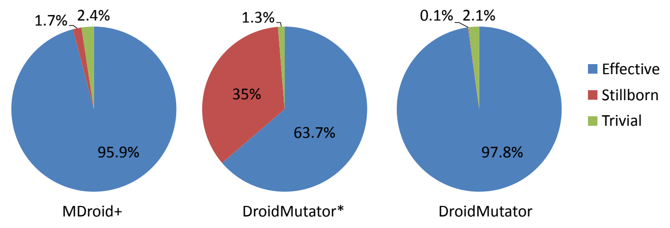


Figure. 4: Mutant ratio

Filtering unusable mutants. As we can see, DroidMutator has the lowest rate of stillborn mutants, compared to DroidMutator* and MDroid+. The number and the ratio of mutants can be

[1]https://github.com/SQS-JLiu/DroidMutator/blob/master/OperatorsDescription.md

obtained from the Table 1 and Figure 4. DroidMutator generates 48703 mutants, 56 stillborn mutants, and thus the stillborn rate is 0.1%. DroidMutator* generates 75213 mutants, 26297 stillborn mutants, and thus the stillborn rate is 35%. MDroid+ generates 8900 mutants, 154 stillborn mutants, and thus the stillborn rate is 1.7%. This clearly demonstrates the superiority of DroidMutator over DroidMutator* and MDroid+ in terms of the stillborn rate.

However, the generation of stillborn mutants is closely related to the rules of mutation operators. Type checking cannot completely avoid the generation of stillborn mutants. For example, the mutation operator InvalidIDFindView is used to replace the id parameter in the Activity .findViewById call. If id does not exist, it cannot be compiled.

Also, it shows that generating trivial mutants is inevitable. For example, the injected faulty statement may be executed when the program starts and throws an exception to cause the program to crash. From Figure 4, we can see that the rates of trivial mutants generated by all three tools are similar.

Table. 2: Mutants generated by DroidMutator

Mutation operator	Mutation Time (sec)	#Gen. Mutants	#Stillborn Mutants	%Stillborn Mutants
Java-specific	896	44187	34	0.08
Android-specific	186	4516	22	0.5

Another observation from Table 2 is that the Java-specific mutation operator implemented by the DroidMutator generates more mutants: the Java-specific mutation operator generates 44187 mutants, and the Android-specific mutation operator generates 4516 mutants. The main reason is that user code in Android apps is mainly traditional Java code.

In summary, DroidMutator effectively reduces the generation of a large number of stillborn mutants and improves the efficiency of mutation, compared to DroidMutator* and MDroid+.

Execution Time. As shown in Table 1, DroidMutator took a total of 1082 seconds to run, averaging 22 seconds per app. DroidMutator* took a total of 1231 seconds to run, averaging 25 seconds per app. MDroid+ took a total of 5568 seconds to run, averaging 111 seconds per app. As such, DroidMutator is the most efficient tool among the three tools but still produces more mutants than MDroid+. By analyzing the results of the experimental data, we found that both DroidMutator* and MDroid+ produced more intermediate analysis files, which required more IO processing and thus took more time than DroidMutator for mutant generation. In other words, they generate a large number of stillborn mutants and copies of the source files.

3.3 Usability

DroidMutator provides both a graphical interface and a command line interface, which allows the developers to visually select the files to be mutated and the mutation operators to generate the desired mutants. Moreover, DroidMutator also visually compares the code differences before and after the mutation to improve the usability of the mutation analysis tool. DroidMutator also provides the flexible scope configurability that allows mutation in the user-defined and rewritten functions. In addition, compared to MDroid+, DroidMutator offers better extensibility on the mutation operators for non-interface mutations. For example, DroidMutator makes it easier to extend the substitutions of the relational and arithmetic operators. But compared to MDroid+, DroidMutator supports mutation operators that manipulate only Java files.

4 CONCLUSION

In this work, we propose DroidMutator, a mutation analysis tool designed specifically for Android apps. DroidMutator checks the types of variables to be mutated and reduces the number of generated stillborn mutants. DroidMutator can also configure the scope of each mutation operator, and then generate mutants in specific code blocks accordingly. Moreover, DroidMutator implements two types of mutation operators, including Android-specific mutation operators and Java-specific mutation operators, and users can easily extend their mutation operators on top of DroidMutator. The experimental results show that DroidMutator effectively reduces the number of stillborn mutants, thus improving the efficiency of mutation analysis for Android apps.

DroidMutator is publicly available at https://github.com/SQS-JLiu/DroidMutator.

ACKNOWLEDGMENTS

This research is supported in part by the Science and Technology Commission of Shanghai Municipality (No.18511106202) and National Science Foundation (CNS-1755772).

REFERENCES

[1] Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt. 2015. Towards mutation analysis of android apps. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–10.

[2] Android Emulator. 2018. . Retrieved August 2, 2018 from https://developer.android.com/studio/run/emulator

[3] F-Droid. 2018. Free and Open Source Android App Repository. Retrieved 2018-08-08 from https://f-droid.org

[4] Gordon Fraser and Andreas Zeller. 2012. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering* 38, 2 (2012), 278–292.

[5] Gradle. 2018. . Retrieved August 1, 2018 from https://gradle.org

[6] Shin Hong, Taehoon Kwak, Byeongcheol Lee, Yiru Jeon, Bongseok Ko, Yunho Kim, and Moonzoo Kim. 2017. MUSEUM: Debugging real-world multilingual programs using mutation analysis. *Information and Software Technology* 82 (2017), 80–95.

[7] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2016. Mutation-Based Fault Localization for Real-World Multilingual Programs (T). In *IEEE/ACM International Conference on Automated Software Engineering*. 464–475.

[8] JavaParser. 2018. *a library to generate, analyze, and process Java code.* Retrieved 2018-09-25 from https://javaparser.org

[9] Yunho Kim, Shin Hong, Bongseok Ko, Duy Loc Phan, and Moonzoo Kim. 2018. Invasive Software Testing: Mutating Target Programs to Diversify Test Exploration for High Test Coverage. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 239–249.

[10] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling mutation testing for android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 233–244.

[11] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. 2006. MuJava: a mutation system for Java. In *Proceedings of the 28th international conference on Software engineering*. ACM, 827–830.

[12] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *IEEE Seventh International Conference on Software Testing*. 153–162.

[13] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[14] Duy Loc Phan, Yunho Kim, and Moonzoo Kim. 2018. MUSIC: Mutation Analysis Tool with High Configurability and Extensibility. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 40–46.

[15] W. Eric Wong, Ruizhi Gao, Yihao Li, Abreu Rui, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[16] Yuan-W. 2017. *muDroid project at GitHub.* Retrieved 2019-05-01 from https://goo.gl/sQo6EL