MutAPK: Source-Codeless Mutant Generation for Android Apps

Camilo Escobar-Velásquez, Michael Osorio-Riaño, Mario Linares-Vásquez
Universidad de los Andes, Bogotá, Colombia
{ca.escobar2434, ms.osorio, m.linaresv}@uniandes.edu.co

***Abstract*—The amount of Android application is having a tremendous increasing trend, exerting pressure over practitioners and researchers around application quality, frequent releases, and quick fixing of bugs. This pressure leads practitioners to make usage of automated approaches based on using source-code as input. Nevertheless, third-party services are not able to use these approaches due to privacy factors. In this paper we present `MutAPK`, an open source mutation testing tool that enables the usage of Android Application Packages (APKs) as input for this task. `MutAPK` generates mutants without the need of having access to source code, because the mutations are done in an intermediate representation of the code (*i.e.,* SMALI) that does not require compilation. `MutAPK` is publicly available at GitHub: https://bit.ly/2KYvgP9 VIDEO: https://bit.ly/2WOjiyy**

***Index Terms*—Mutation testing, closed-source apps, Android**

THE SW DESIGN LAB
ASE'19 preprint

I. Introduction

The power and usefulness of a large number of state-of-the-art approaches for automated software engineering of Android apps rely on the existence of the source code for extracting intermediate representations or models that drive the analysis execution or the artifacts generation.

However, existing approaches that rely on source code for supporting automated software engineering tasks are untenable in a commercial environment where practitioners outsource software engineering tasks, but without releasing the source code.

One of the automated software engineering tasks that has got a lot of attention from researchers, at source-code level, is mutation testing; it consists of modifying an application (by injecting bugs) with the purpose of enhancing and evaluating the quality of a test suite that accompanies the application under analysis. Each injected bug generates a new version of the application, *i.e.,* a **mutant**, that represents a potential bug that should be detected by the existing test suite. Each mutant differs from the original version in a simple modification, called **mutation**. The changes are generated by following a set of rules (*a.k.a.,* operators); the rules can be specific of a programming language (*e.g.,* launch an Activity with an invalid intent) or general to a paradigm as object oriented programming (*e.g.,* assign a null value to a method parameter).

Using mutation testing generates more reliable results in terms of test suite quality and completeness than code coverage [1]. Most of the existing and publicly available approaches for mutation testing use source-code as input, which limits the applicability of mutation testing to scenarios in which source code is not available.

Acknowledgment. This work is founded by a Google Latin America Research Award 2018.

The case of Android apps is not an exception, therefore, in this paper we present **`MutAPK`**, an open-source tool that enables source-codeless mutation testing at APK level. To create **`MutAPK`** we first translated the 38 source-code mutation operations defined by Linares-Vásquez *et. al.* [2] to their intermediate representation's analogous; then, since 3 of previously mentioned operators did not produced compilable results, we implemented 35 out of the 38 operators, using the SMALI intermediate representation; finally, we created a tool that implements the whole mutation process, including the generation of APK mutants ready to be installed on Android devices. **`MutAPK`** (to the best of our knowledge), is the most comprehensive tool for generating mutants of Android apps at APK level. **`MutAPK`** can be downloaded from its public repository at GitHub https://bit.ly/2KYvgP9.

II. Related Work

Several mutation operators have been proposed for different types of applications such as web apps [3], data-centric apps [4], NodeJS packages [5], among many other approaches. For the case of Java applications, there are representative tools such as PIT [6] that implements 29 Java mutation operators. As Android apps are mainly written using Java, approaches for mutation of Java apps can be used in the context of mobile apps. However, Android programming strongly differentiate from Java, which leads to a extensive specialization gap for Android apps.

Consequently, Linares-Vásquez *et. al.,* [2], [7] created a taxonomy of Android bugs with the purpose of defining a list of 38 mutation operators for Android apps. Those operators were implemented in a tool called MDroid+, which performs the mutations at source code level. Deng *et al.* [8], [9] also presented a set of 8 Android specific mutation operators; the mutations are oriented to change core components of Android apps (*e.g.,* intents, event handlers, XML files and activity lifecycle). Additionally, Luna *et al.* [10] presented Edroid, a tool that uses 10 mutation operators oriented to validate changes in the GUI.

Paiva *et al.* [11] propose 3 mutation operators aimed at validating the preservation of user's data and UI state after an application has changed from background to foreground. Jabbarvand *et al.* [12] implement μDroid a mutation testing tool oriented to identify energy-related defects. It is worth

noting that the aforementioned approaches work at source code level and some of them are not publicly available.

muDroid [13] is the only mutation testing tool we found that works at APK level. However, muDroid implements only classic mutation operators (*i.e.,* that are not Android specific). Additionally, MDroid+ authors [2], [7] state that muDroid generates around 53% of non-compilable mutants, *i.e.,* a lost of half of the time invested on executing muDroid.

III. MutAPK

In the following section, we describe **MutAPK** according to its workflow described in Fig. 1. **MutAPK** starts by unpackaging an APK into a temporal folder. After that, the **Potential Fault Profile (PFP)** [2] is derived and a list of potential fault injection locations is created. Given the locations from the PFP, **MutAPK** generates mutants serially, or using multi-threading to reduce the mutant generation time. For each location in the PFP, a copy of the disassembled APK is created. After the copying ends, the mutation is generated. Finally, a compilation process is triggered in order to generate an APK.

There is a noticeable difference with the MDroid+ implementation; because in MDroid+ the source code must be compiled, then the libraries required to compile the source code must be available. This is not the case of **MutAPK**, because we are already working with "compiled" code. Also, while MDroid+ [7] only generates the source code of the mutants, **MutAPK** is able to generate APKs files ready to install and test in Android devices. In the following paragraphs we provide details for each step in the **MutAPK** workflow.

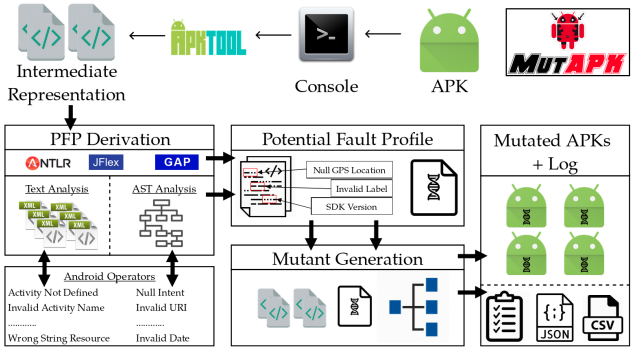

Fig. 1: **MutAPK** architecture and workflow

1) Unpackaging APK: We use APKTool, which allows us to process an APK and decode it into a folder with all the resource files and the source files disassembled into SMALI files. Code-related files are presented in a file structure that resembles the one used at source-code level. Having individual SMALI files, similarly to the original file structure, makes it easier to perform individual mutations and report the location of the changes at method level.

2) Derivation of the Potential Fault Profile: We followed the approach proposed by Linares-Vásquez *et al.* [2], [7], which includes extracting a PFP for the app under analysis. Therefore, we detect mutation locations by first extracting a PFP and then we implement mutation operations on those locations. The PFP is a set of code locations that represent potential points were a fault can be injected. These potential fault injection points are the starting point for the mutation operators described in Section III-3.

To extact the PFP, both XML and SMALI files of an app are statically analyzed searching for instructions that comply with the characteristics defined in the mutation operators. In the case of XML files, **MutAPK** goes through the content of XML files looking for matches between the file tags and the different potential fault injection points. For SMALI files the process is based on the Abstract Syntax Tree (AST). The AST is obtained using the lexer and parser created by APKTool. In particular, **MutAPK** uses the visitor design pattern to identify the potential mutation locations. Using the visitor pattern **MutAPK** can be easily extended to add new operators and to provide more comprehensive analysis of the app (*i.e.,* resource and SMALI files) if needed. The final result of the PFP derivation process is a list that joins the potential fault injection points with the mutation operators that can be applied to those locations.

3) Operators: We built upon the 38 operators proposed by Linares-Vásquez *et al.* [2], [7], which are representative of potential fault in Android apps and can be found either on source code statements, XML tags, or locations in other resource files. In **MutAPK** we implemented (i) the 33 operators implemented in MDroid+ that do not lead to compilation errors, and (ii) two additional operators not available in MDroid+.

To reuse the MDroid+ operators, we translated their implementation from the original source code-based rules to the corresponding implementations in the SMALI representation. The correctness of the implementation rules at SMALI level was manually validated on a set of 11 Android apps with PFPs leading to potential locations for implementing the mutation operators. The apps were selected from the dataset of 54 apps used by MDroid+ [2], [7]. For the 11 apps, we built the APKs and disassembled manually each one to recognize the direct translation of each mutation performed by **MutAPK**. We also manually mutated the source code of these 11 applications to generate a second set of APKs. A third set of APKs was created by generating mutants at source code level using MDroid+. Finally we performed a diff comparison between the SMALI representation of the three sets of APKs. Because of this, we were able to translate successfully each mutation operator from source code to SMALI representation. With this procedure we derived the list of operators implementation at APK level; the details of each operator are available with our online appendix [14].

4) Mutants Creation: In order to generate the mutants, **MutAPK** uses a set of *processors* that are associated to one or more mutation operators. The *processors* understand the location listed in the PFP and are able to modify the file that contains the PFP's location; the changes done by the processors are based on the mutation rules at SMALI level. Therefore, **MutAPK** generates a copy of the decoded application for each location in the PFP that will host one and only one mutation.

THE SW DESIGN LAB
ASE'19 preprint

5) Repackaging and signing: At the end of the generation of each mutant, **MutAPK** uses again the APKTool to package the mutated copy of the app into a valid APK. However, when APKTool builds the APK it loses the signature provided by the developer. Therefore, **MutAPK** uses the Uber APK Signer [15] library to resign the APK. This resigning is required to have APKs that are installable on Android devices/emulators.

6) Extensibility: Due to the fast change of the Android framework, **MutAPK** must provide the possibility of adding new mutation operators easily. Therefore, in order to enable a new mutation operator, some changes must be implemented: (i) create a new detector/locator that is capable of finding the correct position that provides all the information needed to create a *Mutation Location* defined in **MutAPK**; (ii) a mutator, that is capable of using the previously identified location information to mutate the code's SMALI representation or resource file; (iii) update the *operator-types.properties* file found under the "src/uniandes/tsdl/mutapk" folder to add the new mutation operator file path with its defined id; (iv) modify *OperatorBundle.java* (in case the new operator is text-based) to add the new text detector; and (v) update the *operators.properties* file.

It is worth noting that **MutAPK** has an *extra* folder where the external libraries are located. Therefore, if a user wants to improve the file analysis process or wants to execute a more specialized process over an APK, she can locate the library files in this folder and manage them easily.

IV. MutAPK on Action

MutAPK has been designed to work as a command line tool. It requires Maven and Java to be installed on the machine that executes **MutAPK**. To start the usage, the **MutAPK** repository [14] must be cloned and then packaged using the following commands:

```
git clone https://github.com/TheSoftwareDesignLab/
     MutAPK.git
mvn clean
mvn package
```

After that, a *.jar* file will be generated in the *target* folder. After the building process, the **MutAPK** jar can be moved to another location. To run **MutAPK** the following command must be executed (in a command line console):

```
java -jar MutAPK-<version>.jar <APKPath> <AppPackage>
     <OutputFolder> <ExtraComponentFolder>
     <operatorsDir> <multithread> <amountOfMutants>?
```

The description of the parameters in the **MutAPK** command is the following:

1) <**APKPath**>: path to the app's APK
2) <**APKPackage**>: app package name used to identify the Android app
3) <**OutputFolder**>: path to the folder where all the mutants will be generated
4) <**ExtraComponentFolder**>: path to the folder that has the extra libraries used by MutAPK
5) <**operatorsDir**>: path to the *operators.properties* folder, that describes the operators to be used during the mutation
6) <**multithread**>: boolean value, defines if MutAPK must be executed using multiple threads
7) <**amountOfMutants**>: positive number, in case of been provided, it defines the amount of mutants that MutAPK will generate. It is an optional argument

When the command is executed in the console, the selected operators and the amount of mutants that are going to be generated for each operator are logged, along with messages that notify the state of the mutation process, as it is presented in following snippet:

```
Amount Mutants  Mutation Operator
1               OOM_LARGE_IMAGE
3               NULL_INTENT
5               NULL_OUTPUT_STREAM
1               INVALID_FILE_PATH
5               INVALID_LABEL
19              NULL_VALUE_INTENT_PUT_EXTRA
7               INVALID_COLOR
9               FINDVIEWBYID_RETURNS_NULL
19              INVALID_KEY_INTENT_PUT_EXTRA
5               LENGTHY_GUI_CREATION
8               VIEW_COMPONENT_NOT_VISIBLE
3               NULL_INPUT_STREAM
0               SDK_VERSION
7               INVALID_ACTIVITY_PATH
8               INVALID_VIEW_FOCUS
2               CLOSING_NULL_CURSOR
39              WRONG_STRING_RESOURCE
3               WRONG_MAIN_ACTIVITY
1072            NULL_METHOD_CALL_ARGUMENT
4               NULL_BACKEND_SERVICE_RETURN
2               LENGTHY_GUI_LISTENER
9               INVALID_ID_FINDVIEW
7               ACTIVITY_NOT_DEFINED
5               MISSING_PERMISSION_MANIFEST
2               LENGTHY_BACKEND_SERVICE
3               DIFFERENT_ACTIVITY_INTENT_DEFINITION
Total Locations: 1248
Mutant: 1 - LenghtyGUICreation
Creating folder for mutant 1
Copying app information into mutant 1 folder
Mutant: 2 - LenghtyGUICreation
Creating folder for mutant 2
Mutant 1 has survived the mutation process. Now its
     source code has been modified.
Building mutant 1 ...
Copying app information into mutant 1 folder
Mutant: 3 - LenghtyGUICreation
Creating folder for mutant 3
SUCESS: The 1 mutant's APK has been generated
Copying app information into mutant 1 folder
Mutant 2 has survived the mutation process. Now its
     source code has been modified.
Building mutant 2 ...
ERROR: The 2 mutant's APK has not been generated
.........
```

```
<packageName>/
├── <packageName>-mutant1/
│   ├── <packageName>.apk
│   ├── <packageName>-aligned-debugSigned.apk
│   └── (*.smali|*.xml)
├── <packageName>-mutant2/
│   ├── src/
│   └── (*.smali|*.xml)
├── ..........
├── <packageName>-locations.json
├── <packageName>-mutants.log
└── <packageName>-times.csv
```

Fig. 2: **MutAPK** result's folder structure

When **MutAPK** execution ends, it stores in the folder defined as <*OutputFolder*> a set of directories and files that

THE SW DESIGN LAB
ASE'19 preprint

follow the structure depicted in Fig. 2. The result consist of three files and a folder for each generated mutant. First, the content of the mutants folders depends on the result of the building+signing process. Due to nature of the mutation operators, some mutants might not compile. Thus, in case the process is executed successfully a mutant folder will have (*see green lines in Fig. 2*): (i) an unsigned APK, in case the user wants to sign it with different keyset, (ii) a signed APK, ready to be installed and tested, and (iii) a copy of the mutated file (*i.e., .smali* or *.xml*). If the mutant is not successfully generated the resulting folder will have (*see red lines in Fig. 2*): (i) a source folder with XML and SMALI files, in case the user wants to run *apktool* build process and identify what was the specific error, and (ii) the mutated file.

MutAPK also creates a log file that allows testers to identify what was the mutation applied on each mutant. Snippet 1 shows an example of the log; the first line shows the amount of threads that were used with the **MutAPK** execution. The remaining lines show the general information related to a mutant; lines 2, 4 and 6, present the file path, the mutation operator and line number where the mutation was injected; lines 3, 5 and 7 present a textual description of the mutation effect over the app.

Snippet 1: Example of a mutation log for the PhotoStream app

```
1 ThreadPool: 8
2 Mutant 1: com/butacapremium/play/activity/LoginActivity
      .smali; LengthyGUICreation in line { 459 }
3 For mutant 1 a large delay has been injected after GUI
      Creation at line 459
4 Mutant 2: /com/butacapremium/play/activity/LoadActivity
      .smali; LengthyGUICreation in line { 2328 }
5 For mutant 2 a large delay has been injected after GUI
      Creation at line 2328
6 Mutant 3: /com/butacapremium/play/activity/
      SplashActivity.smali; LengthyGUICreation in line {
      343 }
7 For mutant 3 a large delay has been injected after GUI
      Creation at line 343
```

A JSON file is generated as a representation of the location found in the PFP. Therefore, for each location it presents information about the applied mutation operator and mutation location (*i.e.,* starting and ending line numbers, starting column number, length and file path). A detailed report of mutant generation times is stored in a *.csv* file, where copying times, mutation times and building+signing times are shown for each mutant.

Finally, **MutAPK** also allows users to generate a specific amount of mutants by its parameter *amountOfMutants*. When a user provides this parameter, **MutAPK** selects at least one mutant per mutation operator.

V. MutAPK Usage Examples

In order to validate the feasibility of using **MutAPK** on closed-sourced apps, we executed it with two apps from the Google Play top-list: ***Play!*** and ***Tasker Settings***. We downloaded both apks and ran **MutAPK** on a server with 16GB RAM and Intel Core i7 Processor. In both cases, **MutAPK** was able to generate APK mutants ready to be installed, and without having access to the original source code.

For ***Play!*** *(4.1MB)*, **MutAPK** generated 5346 mutants distributed in 20 different mutation operators (8 text-based and 12 code-related). From the generated mutants, 5323 were successfully compiled and the average time for mutation was 416.8 milliseconds, while the average building+signing time was 3.1 minutes.

For ***Tasker Settings*** *(1.06MB)*, 289 mutants were generated within 11 mutation operators (8 text-based and 3 core-related). From the generated mutants 263 were successfully compiled and the average time for mutation was 5.05 miliseconds, while the average building+signing time was 20 seconds.

Note that we did not use the existing tools (*i.e.,* MDroid+ and muDroid) with the two apps, because MDroid+ requires the app source code for the mutations, and muDroid implements classic operators (which does not allow for a comparison to **MutAPK**).

VI. Conclusion & Future Work

With the purpose of enabling mutation testing of Android apps without having access to source code, we created **MutAPK**, an open source tool written in Java that is publicly available at GitHub. As part of the future work, we will extend **MutAPK** to be a complete mutation testing tool, *i.e.,* it also runs existing tests to compute mutation score and suggest developers/testers how to improve their test suites. Another desirable feature in **MutAPK** is to include different mutant selection techniques; current version of **MutAPK** only allow users to set the number of mutants to be generated, which is a trivial technique for mutant selection.

References

[1] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," ser. ICSE 2014, 2014, pp. 435–445.

[2] 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*, 2017, pp. 233–244.

[3] U. Praphamontripong, J. Offutt, L. Deng, and J. Gu, "An experimental evaluation of web mutation operators," in *ICSTW 2016*. IEEE, 2016.

[4] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: an input mutation approach," in *ISSTA 2014*. ACM, 2014.

[5] D. Rodríguez-Baquero and M. Linares-Vásquez, "Mutode: generic javascript and node.js mutation testing tool," in *ISSTA 2018*. ACM.

[6] H. Coles, "Pit. http://pitest.org/," 2017.

[7] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, M. Di Penta, and D. Poshyvanyk, "Mdroid+: A mutation testing framework for android," ser. ICSE '18, 2018.

[8] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of android apps," in *ICSTW 2015*. IEEE, 2015.

[9] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing android apps," *Information and Software Technology*, vol. 81, pp. 154–168, 2017.

[10] E. Luna and O. El Ariss, "Edroid: A mutation tool for android apps," in *CONISOFT 2018*. IEEE, 2018.

[11] A. C. Paiva, J. M. Gouveia, J.-D. Elizabeth, and M. E. Delamaro, "Testing when mobile apps go to background and come back to foreground," in *ICSTW 2019*. IEEE, 2019.

[12] R. Jabbarvand and S. Malek, "miudroid: An energy-aware mutation testing framework for android," ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 208–219.

[13] Yuan-W, "mudroid project at github," 2017, https://goo.gl/sQo6EL.

[14] "Mutapk. https://github.com/TheSoftwareDesignLab/MutAPK."

[15] "Uber apk signer. https://github.com/patrickfav/uber-apk-signer."

THESWDESIGNLAB
ASE'19 preprint