MutAPK 2.0: A Tool for Reducing Mutation Testing Effort of Android Apps*

Camilo Escobar-Velásquez
Universidad de los Andes
Bogotá, Colombia
ca.escobar2434@uniandes.edu.co

Diego Riveros
Universidad de los Andes
Bogotá, Colombia
df.riveros@uniandes.edu.co

Mario Linares-Vásquez
Universidad de los Andes
Bogotá, Colombia
m.linaresv@uniandes.edu.co

ABSTRACT

Mutation testing is a time consuming process because large sets of fault-injected-versions of an original app are generated and executed with the purpose of evaluating the quality of a given test suite. In the case of Android apps, recent studies even suggest that mutant generation and mutation testing effort could be greater when the mutants are generated at the APK level. To reduce that effort, useless (*e.g.,* equivalent) mutants should be avoided and mutant selection techniques could be used to reduce the set of mutants used with mutation testing. However, despite the existence of mutation testing tools, none of those tools provides features for removing useless mutants and sampling mutant sets. In this paper, we present **`MutAPK 2.0`**, an improved version of our open source mutant generation tool (**`MutAPK`**) for Android apps at APK level. To the best of our knowledge, **`MutAPK 2.0`** is the first tool that enables the removal of dead-code mutants, provides a set of mutant selection strategies, and removes automatically equivalent and duplicate mutants. **`MutAPK 2.0`** is publicly available at **GitHub**: https://bit.ly/2KYvgP9 **VIDEO:** https://bit.ly/2WOjiyy

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Software evolution**; • **Human-centered computing** → **Mobile computing**.

KEYWORDS

Mutation Testing, Dead code, Mutant Selection, Equivalent, Duplicate

ACM Reference Format:
Camilo Escobar-Velásquez, Diego Riveros, and Mario Linares-Vásquez. 2020. MutAPK 2.0: A Tool for Reducing Mutation Testing Effort of Android Apps. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3368089.3417942

*Escobar-Velásquez and Linares-Vásquez are partially supported by a Google Latin American Research Award 2018-2020.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417942

1 INTRODUCTION

Mutation testing is a cumbersome and time consuming technique aimed at assessing test suites quality [10]. Therefore, a plethora of previous works have investigated different ways to reduce the effort required by mutation testing [23]. In addition, a significant number of approaches have been proposed to support mutant generation and mutation testing for different languages and types of apps [3, 15, 19, 20]. The case of Android apps is not the exception; developers and researchers can use a diverse set of tools for mutation testing of Android apps [6, 7, 11, 13, 17, 22], and there are also specific challenges that impact the cost of mutation testing. For instance, enabling mutation at APK level [8] increases signifcantly the amount of Android-specific mutants that can be generated, when compared to mutation at source-code level.

In general, removing useless mutants (*i.e.,* equivalent, duplicate, and dead-code mutants) is a widely used technique for reducing testing efforts. The latter type, dead-code mutants[1], are less common because compiler optimizations take care of removing dead-code when building executable files/packages. However, this is not the case for Android native apps written in Java and Kotlin, because dead code is not removed (by default) when building APK files. In addition, mutant selection techniques [24] are a desirable feature during mutation testing to reduce the amount of analyzed mutants while preserving the quality of the mutant population.

Unfortunately, despite the availability of a diverse set of mutation tools for both source-code and bytecode levels, none of the existing tools provide developers with capabilities for removing dead-code, equivalent and duplicate mutants. Moreover, practitioners and researchers lack publicly available tools for mutant selection and sampling. Therefore, in this paper we present **`MutAPK 2.0`**, an improved version of our original mutant generation tool (**`MutAPK`**) for android apps at APK level. **`MutAPK 2.0`** enhances mutant generation process by removing dead-code, equivalent and duplicated mutants. Additionally, it provides users with three mutant selection techniques based on random and representative subset selection.

`MutAPK 2.0`, to the best of our knowledge, is the first tool enabling techniques for reducing mutation testing effort of Android apps. **`MutAPK 2.0`** is open sourced and publicly available at **GitHub**: https://bit.ly/2KYvgP9. The **`MutAPK 2.0`** repository also includes an online appendix containing tutorials, evaluation data, and usage examples.

2 RELATED WORK

Mutation testing is a well know testing technique used to evaluate the quality of a given test suite. Several mutation operators and

[1]By dead-code mutants we mean mutants that are generated by applying mutation operators on dead-code.

tools have been proposed for different languages and app types such as web applications [19], NodeJS packages [20], JS apps [15] and data-intensive applications [3]. In the case of Android apps, several approaches and tools have been proposed to address different perspectives of mutation testing [6–9, 11, 13, 14, 16, 17, 22]. Most of the existing tools allow for mutant generation, and some of the tools provide mutation testing capabilities as in the case of Pit[5] and Major[12] for Java systems, and Stryker [21] and Mutode for JS/Node apps [20]. Surprisingly, none of the tools provide developers and researchers with features for reducing useless mutants (*i.e.,* dead-code, duplicate, and equivalent mutants) and sampling the set of mutants to be used during mutation testing.

3 THE MUTAPK 2.0 TOOL

In this section, we describe the main features implemented in **MutAPK 2.0** to support (i) mutant selection, (ii) identification and removal of duplicated and equivalent mutants, and (iii) dead code analysis. Fig. 1 presents the workflow **MutAPK 2.0** follows to generate and reduce APK mutants for Android apps. Note that our first release of **MutAPK** [8] focused on mutants generation but without having mechanisms for reducing the generated mutants. In this paper we describe the components added and modified on top of **MutAPK** (*i.e.,* sections framed in red in Fig. 1). In order to understand the original **MutAPK** architecture we refer the interested reader to our preliminary publication [8] and online appendix [1]. As the reader can note, we have improved three stages of the mutant generation process: (i) prior to the Potential Faults Profile (PFP) definition[2], (ii) after the PFP definition and before mutant generation, and (iii) during mutant building process.

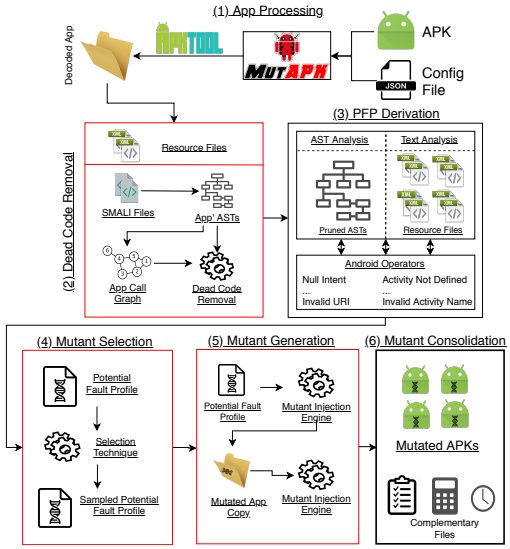

Figure 1: MutAPK 2.0 architecture and workflow

3.1 Avoiding Mutants from Dead Code

The Android compilers for Java and Kotlin languages do not remove dead code automatically, which means that by default, APKs of native apps include dead code. Thus, **MutAPK 2.0** allows users to reduce the mutants set by not injecting mutation operations on dead code. In order to identify the dead code within the app' SMALI representation[2], our tool builds the call graph of an app under analysis and identifies the methods that are not called by others.

Before explaining the dead code detection details, we provide some context about how a method call is represented in SMALI. As it can be seen in Fig. 2, a method call has 4 sections that are relevant for our purpose: (i) ***unitName***, (*i.e.,* the class a method belongs to), (ii) ***methodName***, (iii) ***parameters***, represented as a concatenation of the types of the method arguments, and (iv) ***returnType***. Having this in mind we define the `methodId` of a method as the concatenation of ***methodName***, ***parameters*** and ***returnType***; and the `methodCompoundId` as the concatenation of ***unitName*** and ***methodId***. This decision is based on the fact that it might exist more than one method with the same name but that differs by the parameters count and types.

```
invoke-virtual {p0, v0},
      Lcom/evancharlton/mileage/charts/LineChart; ← unitName
    -> publishProgress([Ljava/lang/Object;)Landroid/content/ContentResolver;
         methodName      parameters          returnType
```

Figure 2: SMALI representation of method call

Concerning the dead code detection, **MutAPK 2.0** splits the AST computation and PFP derivation [13] into two steps. First, it creates a dictionary that maps the ***unitName*** of a SMALI file to an object containing the AST and the qualified path. Second, using the previously generated ASTs, **MutAPK 2.0** visits all the methods within the different ASTs to identify the method calls.

Afterwards, **MutAPK 2.0** extracts the *Call Graph* of the app using a two levels HashMap, being the second level a mapping from the `methodId` to a *CallGraphNode (cGN)* and the first level a mapping from the ***unitName*** to the aforementioned second level HashMap of methods. Now, regarding the content of a *CallGraph node*, we store the `methodId`, the ***unitName***, the AST node that represents the method, and two sets containing (i) the *cGNs* of the methods that call the current method (*callers*), and the *cGNs* of the methods that are called by the current method (*callees*). As the reader might know, there are some methods that are called through the code that do not belong to the app's developed code (*i.e.,* libraries' methods), thus, we excluded API calls from the call graph creation[3].

Once **MutAPK 2.0** has built the call graph, the methods that do not have a caller are removed from the original AST, therefore, during the mutant generation stage, no mutation is applied on those methods. Note that there are some callback and lifecycle methods in the Android programming model that are automatically invoked by the framework, so, those methods usually do not have callers in the call graph; therefore, we have manually created a *greenList* of those methods that should not be removed from the AST [4]. Finally, when all ASTs were pruned conserving the methods in the *greenList*, the PFP derivation and mutant generations process are executed.

[2] A PFP is the set of code locations where each mutation operator can be applied [13].

[3] A file containing a list of ***methodCompoundId*** for API calls is generated at the end of mutant generation process in case a user wants to analyze them.

[4] The list of methods can be found in our online appendix[1]

3.2 Mutant Selection Techniques

After the PFP derivation is completed, a mapping between the mutation operators and the locations within the code identified as mutable is obtained. Using this map as input, along with the user selected technique, **MutAPK 2.0** generates a subset of the PFP to be used during the mutant generation process. If no selection technique is defined by the user, then, the whole set of mutants is generated. We describe each selection technique as follows.

3.2.1 Random selection (randSel). In this case, a user must provide a desired amount of mutants to be selected. First, **MutAPK 2.0** checks that the user's required amount of mutants is less than the available PFP locations. Second, in order to ensure all mutant operators that have PFP locations, are being represented in the subset, **MutAPK 2.0** uses a *round robin* exploration technique to go through all possible key values in the PFP map. Once the algorithm has retrieved the list of locations associated to a mutant operator, it randomly selects one element, adds it to the final set of PFP locations and remove it from the mutation operator's list. If there is at least one location per identified mutation operator, **MutAPK 2.0** creates a list with all the available locations within the PFP and randomly selects the remaining amount of mutants.

3.2.2 Representative Subset. The second available technique is based on the selection of an amount of mutants that fits the size of a representative sample of the PFP locations set. In order to use this technique, a user is required to provide three values: (i) ***confidenceLevel (c)***, (ii) ***marginError (e)***, and (iii) ***selectionScope***. **MutAPK 2.0** uses the equation shown in Fig. 3 to calculate the size of the subset. Until this point there are two values that are not being provided by the user, the *z-score* and the base' population size. Nevertheless, *z-score* is associated to the confidence level and the size of the population size is calculated based on the value of the latter parameter.

$$sampleSize(N, e, z) = \frac{\frac{{z_c}^2 p(1-p)}{e^2}}{1 + \left(\frac{{z_c}^2 p(1-p)}{e^2 N}\right)}$$

Figure 3: Sample Size equation. N = population Size, e = margin of error, z_c = z-score, and p = expected proportion

The population size (N) is computed based on the value of the selectionScope argument provided by the user:

Per Mutation operator (rSPerOperator). Represented as a *positive* boolean value in the config file, this option generates the resulting subset as a concatenation of the subsets of the locations belonging to each mutant operator. Which means, if a user selects this option, **MutAPK 2.0** would go through all mutant operators in the PFP, calculating and randomly selecting a sample of the locations associated to the mutant operator. Therefore, the resulting set would be a concatenation of the subsets for each mutant operator. For example, let us say we have a PFP with 1000 locations that are distributed between two mutant operators: *InvalidURI (N=600)* and *NullInputStream (N=400)*, and let us use a ***confidenceLevel*** of 95% and ***marginError*** of 5%. Once **MutAPK 2.0** is executed it would calculate the sample size for *InvalidURI* using the previous equation and will randomly select 235 mutants from the 600 mutants. Afterwards, it will continue by calculating the amount for *NullInputStream* resulting in a set of size 197. Finally, it would return a set of 432 *(235+197)* mutants.

Whole PFP set (rSWholePFPSet). On the other hand, if a user provides a *negative* boolean value, **MutAPK 2.0** uses the whole PFP set size as the *N* parameter of the equation. Nevertheless, it would first select at least one mutant per mutation operator to ensure the representativeness of the sample. Using the same example previously defined, **MutAPK 2.0** would use 1000 as *N*, however, it would first randomly select a mutant from each mutant operator.

3.3 Duplicate and Equivalent Mutants

To enhance the quality of the generated mutants set, we have included a step for removing duplicate and equivalent mutants. This process relies on the *Trivial Compiler Equivalence (TCE)* proposed by Papadakis *et al.* [18]. Given two apps, TCE calculates their similarity via the hashcode of its files. Specifically, in order to identify equivalent mutants (*i.e.,* mutants that are syntactically equivalent to the original app) we compare each mutant to the base app, and for duplicate mutants (*i.e.,* mutants that are syntactically equivalent to other mutants) we compare each pair of mutants by using TCE.

Since one of the most time consuming processes in the generation of mutants is the compilation/building of the mutants, we decided to compute TCE using three hashcodes for each mutant [9]: (i) *smali folder hash*, containing all the source code representations, (ii) *res folder hash*, containing all the resources of the app, and (iii) *AndroidManifest.xml hash*. By relying on the three hashes, we compute a ***compoundHashcode*** for each mutant, with the procedure depicted in Listing 1. This approach allows us to generate one Hashcode that represents the mutant and to use a hashmap structure to map ***compoundHashcode*** values to mutants . Afterwards, if there is no difference between the ***compoundHashcode*** of original app and a mutant or between two mutants, then we mark the mutants as equivalent or duplicate (respectively).

Snippet 1: Compound Hash code computation algorithm

```
int hash = 7;
hash = 31 * hash + hashManifest.hashCode();
hash = 31 * hash + hashSmali.hashCode();
hash = 31 * hash + hashResource.hashCode();
return hash;
```

The preference for a hashmap to map ***compoundHashcodes*** and mutants, is based on the fact that its search complexity is $O(1)$. Therefore, it allows **MutAPK 2.0** to easily identify if a mutant ***compoundHashcodes*** already exists. **MutAPK 2.0** computes the ***compoundHashcode*** of the base application and stores it in the hashmap. By this, **MutAPK 2.0** is able to identify, in case a hashcode is duplicated, which mutant is colliding to. **MutAPK 2.0** also computes the original app's ***compoundHashcode*** and adds it to the hashmap. Therefore, if there is a collision with the original app, then the mutant is identified as equivalent; if there is a collision but between mutants, the mutant is then tagged as a duplicate.

4 MUTAPK 2.0 IN ACTION

In order to evaluate the **MutAPK 2.0** features, we used our tool with 10 open source android apps belonging to the androtest [4] dataset.

In particular, we aimed at measuring the reduction of the dataset when removing dead-code mutants, as well as equivalent and duplicate ones. In addition, we evaluated whether the implementation of the selection techniques produced balanced sets (in terms of the mutation operators). Since our approach analyzes dead code within the SMALI representation, we selected the top 10 apps in terms of the smali folder size of their corresponding APK files. Note that (i) the original version of our tool [8] does not detect dead code before applying the mutation operators, and (ii) both (1.0 and 2.0) versions implement the same list of mutation operators. The list of analyzed apps is with our online appendix[1].

4.1 Dead-code Mutants

To measure the number of dead-code mutants we executed both versions of our tool (**`MutAPK 1.0`** [8] and **`MutAPK 2.0`**) on the ten selected apps and without using the equivalent/duplicate removal feature from **`MutAPK 2.0`**. By computing the difference in the number of mutants generated by each tool, we were able to identify the number of mutants created by mutations on dead code.

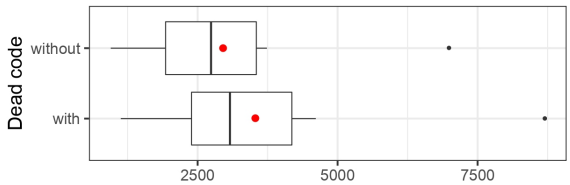

Figure 4: Amount of mutants generated with and without dead code on the 10 analyzed apps.

Fig. 4 presents the amount of mutants generated on the 10 apps. As expected, the set of mutants is larger when dead code is not removed. Specifically, there is a difference (on average) of 15.5% when removing dead-code mutants. Since mutation testing requires to execute a test suite on the generated mutants, by avoiding mutant generation over dead code, a user might reduce its execution time by 15%. In our dataset, *com.eleybourn.bookcatalogue* is the app with the largest number of dead-code mutants, 1711 cases (19.9%). Nevertheless, the *com.bwx.bequick* app is the one having the largest percentage of dead code mutants: 20.9% (*i.e.,* 462 out of 2210).

4.2 Mutant Selection Techniques

In order to evaluate the behavior of the selection techniques, we calculated the difference between the estimated amount of mutants[5] and the number of selected mutants by operator and app. Therefore, we extracted from the **`MutAPK 2.0`** logs the number of mutants generated for each mutant operator per app after applying a given selection strategy. Additionally, we estimated the expected amount of mutants that should be generated per mutant operator in order to preserve the distribution of the original mutant dataset (*i.e.,* without applying any selection technique). Once we had those values, we computed the average difference for the three selection techniques available in **`MutAPK 2.0`**.

[5] Amount of mutants required to preserve the proportion between mutant operators when selecting a sample set

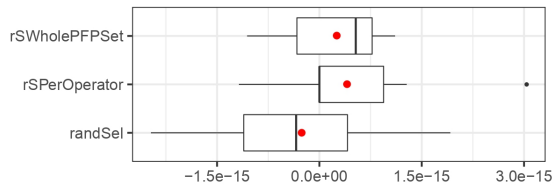

Figure 5: Average difference between expected amount of mutants and amount of generated mutants per mutant operator for each selection technique. randSel = Random Selection, rSPerOperator = representative subset per operator, rSWholePFPSet = representative subset whole PFP set

Fig. 5 depicts the average differences computed previously. The obtained values show that the three selection techniques generated a small difference when compared to the expected amount. Nevertheless, we could notice, by analyzing the standard deviation, that even when the average difference is close to 0, the distance to the mean per mutant operator is higher when using a fully random technique (*i.e., amountMutants*) than when using a technique based on representative subset.

4.3 Duplicate and Equivalent Mutants

Regarding the duplicate mutants, we processed the output of **`MutAPK 2.0`** and found that for our apps dataset there was on average 16 duplicate mutants per app, being *com.nloko.android.syncmypix* the top-1 app with 44 mutants (~2% of the mutants in the app).

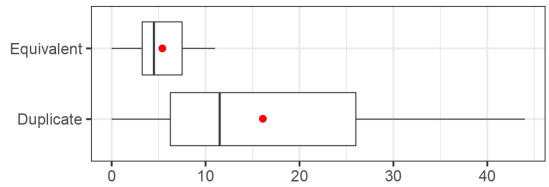

Figure 6: Average amount of mutants tagged as equivalent and duplicate per app

In terms of equivalent mutants, we found ~5.4 cases per app, being *com.eleybourn.bookcatalogue* the one with more equivalent mutants (*i.e.,* 11 out of 6990).

5 CONCLUSION & FUTURE WORK

Mutation testing is a time consuming process, therefore, reducing dead-code, equivalent and duplicate mutants contributes to reduce the number of mutants to be tested. Previous studies have depicted the large amount of mutants generated for android apps [9, 13] and in particular when mutation is done at APK level. Thus, to reduce the mutation testing effort required with APKs, in this paper we presented **`MutAPK 2.0`**, which is an extension of the original **`MutAPK`** tool. **`MutAPK 2.0`** removes dead-code mutants, and duplicate and equivalent mutants by following the TCE approach by Papadakis *et al.* [18]. In addition, **`MutAPK 2.0`** allows for mutant selection using three sampling techniques. To the best of our knowledge **`MutAPK 2.0`** is the only tool having these capabilities.

Future work will be focused on implementing more mutant selection techniques (*e.g.,* regression-based mutation), extending the scope of dead code removal to resources such as strings and colors, and extending the available mutant operators set.

REFERENCES

[1] [n.d.]. MutAPK. https://github.com/TheSoftwareDesignLab/MutAPK.

[2] [n.d.]. SMALI. https://github.com/JesusFreke/smali.

[3] Dennis Appelt, Cu Duy Nguyen, Lionel C Briand, and Nadia Alshahwan. 2014. Automated testing for SQL injection vulnerabilities: an input mutation approach. In *ISSTA 2014*. ACM.

[4] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *ASE'15*. 429–440. https://doi.org/10.1109/ASE.2015.89

[5] Henry Coles. 2017. PIT. http://pitest.org/.

[6] Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt. 2015. Towards mutation analysis of android apps. In *ICSTW 2015*. IEEE.

[7] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. 2017. Mutation operators for testing Android apps. *Information and Software Technology* 81 (2017), 154–168.

[8] C. Escobar-Velásquez, M. Osorio-Riaño, and M. Linares-Vásquez. 2019. MutAPK: Source-Codeless Mutant Generation for Android Apps. In *2019 IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

[9] C. Escobar-Velásquez, M. Linares-Vásquez, G. Bavota, M. Tufano, K. P. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk. 2020. Enabling Mutant Generation for Open- and Closed-Source Android Apps. *IEEE Transactions on Software Engineering* (2020), 1–1.

[10] R. G. Hamlet. 1977. Testing Programs with the Aid of a Compiler. *IEEE Trans. Software Eng.* 3, 4 (July 1977).

[11] Reyhaneh Jabbarvand and Sam Malek. 2017. miuDroid: An Energy-aware Mutation Testing Framework for Android *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 208–219. https://doi.org/10.1145/3106237.3106244

[12] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *ASE 2011*.

[13] 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 *ESEC/FSE'17*. 233–244. https://doi.org/10.1145/3106237.3106275

[14] Eduardo Luna and Omar El Ariss. 2018. Edroid: A Mutation Tool for Android Apps. In *CONISOFT 2018*. IEEE.

[15] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. [n.d.]. Efficient JavaScript Mutation Testing *(ICST 2013)*. https://doi.org/10.1109/ICST.2013.23

[16] 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 *(ICSE '18)*. 4. https://doi.org/10.1145/3183440.3183492

[17] Ana CR Paiva, João MEP Gouveia, Jean-David Elizabeth, and Márcio E Delamaro. 2019. Testing When Mobile Apps Go to Background and Come Back to Foreground. In *ICSTW 2019*. IEEE.

[18] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique *(ICSE 2015)*, Vol. 1. https://doi.org/10.1109/ICSE.2015.103

[19] Upsorn Praphamontripong, Jeff Offutt, Lin Deng, and JingJing Gu. 2016. An experimental evaluation of web mutation operators. In *ICSTW 2016*. IEEE.

[20] Diego Rodríguez-Baquero and Mario Linares-Vásquez. [n.d.]. Mutode: generic JavaScript and Node.js mutation testing tool. In *ISSTA 2018*. ACM.

[21] Info Support. 2020. Stryker. https://stryker-mutator.io/.

[22] Yuan-W. 2017. muDroid project at GitHub. (2017). https://goo.gl/sQo6EL.

[23] Jie Zhang, Muyao Zhu, Dan Hao, and Lu Zhang. 2014. An empirical study on the scalability of selective mutation testing. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 277–287.

[24] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. 2010. Is operator-based mutant selection superior to random mutant selection?. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 435–444.