Towards differential fuzzing to reduce manual efforts to identify equivalent mutants: A preliminary study

Bruno E. R. Garcia
ICMC/USP
São Carlos, SP, Brazil
bruno.erg@usp.br

Marcio E. Delamaro
ICMC/USP
São Carlos, SP, Brazil
delamaro@icmc.usp.br

Simone R. S. Souza
ICMC/USP
São Carlos, SP, Brazil
srocio@icmc.usp.br

ABSTRACT

Mutation testing is a technique that assesses the effectiveness of a set of test cases by introducing changes to the source code and checking whether the test cases can detect them. However, mutation testing is costly, and many academic efforts have been directed to improve its effectiveness and reduce costs. One of the challenges related to mutation testing remains in the equivalent mutant problem. Fuzzing, as a search technique, can find test cases that the developers might not have addressed in unit testing, and it could be used to identify equivalent mutants. In this paper, we present a preliminary study that investigates the use of differential fuzzing to identify equivalent mutants. To identify equivalent mutants, one approach is to set a timeout period after which any surviving mutants are considered equivalent. In our experiment, a 3-minute timeout yielded an accuracy rate of 97%. In conclusion, differential fuzzing can be used to identify equivalent mutants accurately at a reasonable time, especially for projects that maintain a robust seed corpus for fuzzing.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Mutation testing, fuzzing, differential fuzzing, equivalent mutant problem

1 INTRODUCTION

Mutation testing is a technique that assesses the effectiveness of a set of test cases by introducing changes (mutations) to the source code and checking whether the test cases can detect them. This technique has been largely explored to improve the effectiveness of software applications [12] [14] [22] and has been adopted in industry [21].

However, mutation testing is costly, and many academic efforts have been directed to improve its effectiveness and reduce costs. One of the challenges related to mutation testing remains in the equivalent mutant problem. When a set of tests cannot detect a mutant because the behavior of the mutant may be the same as the original program for all test cases, we should evaluate it to determine if this mutant is equivalent to the original program. That is, the mutant kept the program behavior unchanged and thus cannot be detected by any test case [17]. The equivalent mutant problem has been discussed in academia, and researchers have shown that much work to identify them has been done manually [24]. Recently, many studies have emerged using Machine Learning (ML) to classify equivalent mutants [18] [3] [20]. However, as pointed by Chung and Yoo [5], it is a challenge to build a large enough dataset to train such a model, and most studies that use ML for this purpose have relied on small datasets for training.

Fuzzing (or fuzz testing) is a software testing technique that discovers program failures by checking their behavior against a large number of test cases and inputs, especially with malformed and unexpected inputs. The idea emerged in early 1980 when Professor Barton Miller was connected to his university computer via a telephone line during a storm. The thunderstorm caused noise on the line, and this noise, in turn, caused the UNIX commands to get bad inputs – and crash [23]. That crash made him question the reliability of the UNIX systems, and then he asked his students to develop a basic command-line fuzzer to test the reliability of Unix programs by bombarding them with random data and monitoring for any crashes.

One of the most used fuzzing approaches is called "coverage-guided fuzzing", commonly known as grey-box fuzzing [13]. As the fuzzing engine runs the target program with different inputs, it monitors the code coverage achieved by each input. Code coverage refers to which parts of the program's code were executed during the input's processing. The goal is to maximize code coverage, as unexplored code paths may hide bugs or vulnerabilities. The fuzzing engine iteratively mutates or evolves the input data based on the feedback from code coverage. Inputs that lead to increased code coverage or trigger new code paths are prioritized for further mutation or evolution. This process helps explore the program's logic deeper and potentially discover hidden bugs. In general, we can look at fuzzing from a genetic algorithm perspective, where:

- **Initial Population**: Generation of an initial set of test cases to start *fuzzing*, randomly done.
- **Fitness assessment**: Evaluation of each input based on predefined criteria such as code coverage, crash-inducing potential, or specific behavioral triggers.
- **Selection**: Choice of the most suitable individuals from the population to be subjected to additional mutation.
- **Mutation and crossover**: Introduce variations in selected inputs through operations such as mutation (changing individual bits or characters) and crossover (combining parts of two or more inputs).
- **Repetition**: Repeat the process iteratively, with more suitable inputs having a greater probability of surviving and contributing to the next generation.
- **Stop**: Stop the execution based on a defined termination condition, such as a time limit, desired coverage achievement, or bug detection.

To increase efficiency and save time, many projects maintain a *seed corpus*. The *seed corpus* is a collection of valid, well-formed test cases that are used as a starting point for the *fuzz* testing. Since

fuzzing is literally based on search, having a starting point avoids all the effort of executing it from scratch until finding the first test cases that will really be useful to test the application. Generally, projects keep a *seed corpus* that covers as much code as possible. Cheng et al. [4] shows that the success of a fuzzing campaign heavily depends on the quality of seed inputs used for test generation. In their study, they propose the usage of Machine Learning to improve the quality of seed inputs for fuzzing programs that take PDF files as input.

Fuzzing has been used in a lot of projects and uncovered many bugs and vulnerabilities [2] [26]. Due to its significant adoption, we should consider it in mutation analysis. However, previous work has pointed out that fuzzing cannot kill a wide variety of mutants [9]. Although this statement seems negative, this behavior is expected since fuzzing cannot detect errors that do not induce the system to crash or fail, since writing an oracle that can deal with all possibilities of test cases is impractical. In contrast, the authors point out that *differential fuzzing* can be promising in this context.

Differential fuzzing is a technique that applies fuzzing in two or more systems - or different versions of the same system - simultaneously using the same test cases, then compares the outputs to check for discrepancies between them. In conjunction with mutation testing, this technique has been used to evaluate fuzzing tools with promising results [7].

Since fuzzing is a technique that is used to find cases that uncover critical bugs in software, we can suppose that if a mutant is not equivalent, there is a high probability of fuzzing finding a test case that reveals it. However, since writing an oracle to cover all the possibilities is impractical, we could apply *differential fuzzing* between a mutant and its original code to reveal it by checking discrepancies in their outputs. For this reason, this paper aims to discuss using *differential fuzzing* to classify mutants, focusing on identifying equivalent mutants.

2 RELATED WORK

Groce et al. [9] described an effort to investigate and enhance the effectiveness of the Bitcoin Core implementation fuzzing effort. Their research started from a question about how to escape saturation in the Bitcoin Core's fuzzing effort. The authors explored different approaches to analyze it, such as ensemble fuzzing, swarm testing, and mutation analysis. By applying mutation testing, Groce et al. [9] pointed out fuzzing was able to detect only 12% of all mutants, which is a good result since fuzzing is not so effective in detecting non-crash-inducing bugs. Since Bitcoin Core is the reference implementation of the Bitcoin protocol, the authors pointed out that usage of *differential fuzzing* could be promising in that context. Groce et al. concludes the study showing that improvements to the *oracle* may be the best way to get more out of fuzzing.

Mutation testing has also been used to evaluate fuzzers. Görz et al. [7] show that eliminating coverage mutants using static seed files and using "super mutants" for the remaining evaluation can significantly reduce the computational expenditure necessary for mutation analysis and make mutation analysis feasible for fuzzing. In the same direction, Groce et al. [10] proposed that fuzzing programs "near" the System Under Test (SUT) can, in fact, improve the effectiveness of fuzzing. Their preliminary experiment shows that fuzzing mutants is trivial to implement and provides a better overview of a fuzzing harness's mutation score.

Vikram et al. [25] developed and evaluated a tool called Mu2, a Java-based framework for incorporating mutation analysis in the grey-box fuzzing loop. The goal is to produce a test-input corpus with a high mutation score. They implemented differential testing as an oracle for killing mutants and proposed optimizations to improve fuzzing throughput by dynamically pruning the number of mutants to be executed. Moreover, one of their intentions was to spread mutation analysis techniques in the fuzzing community.

Nourry et al. [19] investigated the obstacles encountered by developers in the realm of fuzzing activities. This study analyzed 829 randomly selected GitHub issues related to fuzzing, from which they derived insightful questions. Notably, practitioners frequently highlighted the issue of bad fuzzing targets, resulting in failures or inconsistencies, as a prominent concern in their findings. Furthermore, a substantial concern was raised about the excessive consumption of resources by the fuzzing process, highlighting it as a significant issue.

Jain et al. [15] proposed to use the difference between source code coverage and mutation score as a new metric to evaluate test adequacy. They advocate that it provides a way to identify source files where it is likely a weak oracle test important code.

Fernandes et al. [6] explores the reduction of manual effort in mutation testing. They introduced an approach to suggest equivalent mutants based on automated behavioral testing, which consists of test cases based on the behavior of the original program. Compared to manual analysis to identify equivalent mutants, their approach takes a third of the time to suggest equivalents and is 25 times faster to indicate non-equivalents.

Klooster et al. [16] studied the effectiveness and scalability of fuzzing in Continuous Integration/Continuous Delivery (CI/CD) pipelines. Their research was surrounded by the research question: *"What is a reasonable fuzzing campaign duration that is compatible with CI/CD testing timelines but is still effective in finding security vulnerabilities?"*. They conclude that campaigns of 10 minutes can still be almost as effective as ones that take multiple hours.

3 PRELIMINARY STUDY

This preliminary study focuses on utilizing differential fuzzing to identify equivalent mutants, guided by the following research questions:

RQ1 - What is the time required for differential fuzzing to effectively kill a mutant?

RQ2 - How differential fuzzing could identify equivalent mutants?

RQ3 - How does differential fuzzing compare to other testing methods, such as system or unit testing, in terms of efficiency and the detection of specific mutant operators?

3.1 Experimental setup

The experiments were conducted in Bitcoin Core, which is the reference implementation of the Bitcoin protocol, widely adopted by over 98% of nodes in the Bitcoin P2P network [1]. This project is notably robust, featuring over 35,000 stars and 900 contributors

[1]https://github.com/bitcoin/bitcoin

on GitHub, along with more than 40,000 commits, making it a substantial test subject for software testing research.

For mutation testing, our study focuses on the coin selection function of Bitcoin Core, specifically the "Branch and Bound" (BnB) algorithm [2]. This function was selected for its implementation of a widely utilized search algorithm that effectively breaks down optimization problems into smaller, manageable subproblems. It employs bounding functions to swiftly eliminate subproblems that cannot yield optimal solutions, thereby enhancing efficiency. Additionally, the BnB function is notable for its high test coverage across multiple testing methodologies, including fuzzing, unit tests, and functional tests. To conduct our mutation analysis, all available tests within these categories will be executed to evaluate the effectiveness of the function under various mutated conditions.

3.2 Mutation testing tool

To perform mutation testing we chose the *universalmutator*, a versatile, regex-based tool capable of generating and analyzing mutants across multiple programming languages [8]. This tool stands out for its flexibility in adapting to various languages and for its testing capabilities which include generating mutants and conducting mutation analysis through running existing tests. The mutant operators and the number of each operator used by the tool for the Bitcoin Core's BnB algorithm are summarized in Table 1. The three most applied rules were: Arithmetic Operator Replacement, which involves substituting arithmetic operators (e.g., replacing + with -), Conditional Expression Replacement, which modifies conditional logic (e.g., replacing && with ||), and Relational Operator Replacement, which alters relational operators (e.g., replacing == with !=),

Table 1: Mutant operators and the number of valid mutants generated for each operator by *universalmutator* for Bitcoin Core's Branch and Bound algorithm

Mutant operator	No. of valid mutants
Arithmetic Operator Replacement	100
Conditional Expression Replacement	76
Relational Operator Replacement	75
Break Statement Addition	37
Continue Statement Addition	37
Array Index Replacement	3
Math Function Replacement	3
Else Statement Deletion	2

3.3 Differential Fuzzing Application

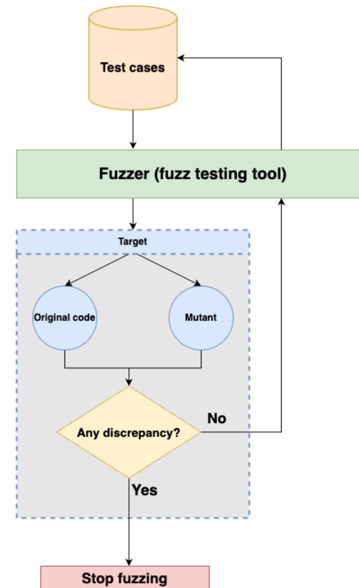
Differential fuzzing was applied between mutants and the original code to evaluate the first two research questions. This technique involves checking if same inputs in both the original and mutated code produce different outputs, an indication of behavioral discrepancies is useful for identifying non-equivalent mutants, as shown in Figure 1.

[2]https://github.com/bitcoin/bitcoin/blob/26.x/src/wallet/coinselection.cpp

The fuzzer used to perform it was *libfuzzer* [3], an in-process, coverage-guided, evolutionary fuzzing engine that utilizes LLVM's SanitizerCoverage instrumentation. This tool is tasked with generating initial test cases, mutating them to create new test cases, and analyzing the behavior of the system under test.

The "target", as shown in Figure 1, is a code that will receive the inputs from the fuzzing tool, send them to the applications under test, and, finally, act as an oracle that will compare the applications' outputs to check whether there are any discrepancies.

Figure 1: Differential fuzzing between a mutant and its original code



3.4 Experimental design

Experiments were designed to evaluate the efficiency of differential fuzzing under different conditions:

(1) Employing only the test cases from the system under test's seed corpus.
(2) Running differential fuzzing without the seed corpus until the mutant is terminated, with a timeout of 24 hours per mutant.
(3) Using the seed corpus until the mutant is killed, with a timeout of 24 hours per mutant.

All experiments were performed on a Macbook Pro M1 with 16GB RAM. All results that will be presented for fuzzing and differential fuzzing are derived from the average of 30 executions excluding outliers to ensure statistical consistency and reliability of the findings. Also, we compiled the code with the sanitizers: UndefinedBehaviorSanitizer (UBSan) and AddressSanitizer (ASan). UBSan identifies operations that lead to undefined behavior as per the C++ standard, such as integer overflows, null pointer dereferencing, and out-of-bounds array indexing, thus preventing unpredictable program behavior. ASan, on the other hand, focuses on memory errors,

[3]https://llvm.org/docs/LibFuzzer.html

including buffer overflows, use-after-free, and memory leaks, by instrumenting the code to monitor memory accesses and utilizing a shadow memory for tracking.

4 RESULTS

Figure 2 shows the results of our mutation analysis conducted on the Branch and Bound algorithm within Bitcoin Core's coin selection module. We tested the mutants using all unit tests and functional tests that reached the mutated code, the coin selection fuzz target, and differential fuzzing between a mutant and the original code. Out of 448 mutants, 30 were not killed by any test. Notably, only 11% of the mutants were killed by the fuzzing process alone, which is consistent with the findings reported by Groce et al. [9].

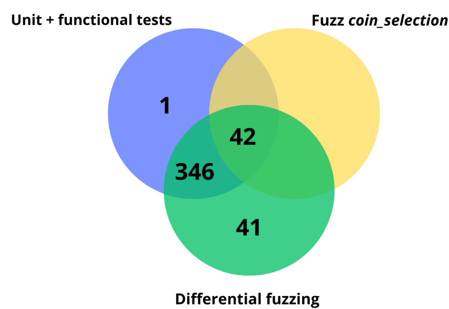


Figure 2: Number of killed mutants

4.1 Differential Fuzzing performance

Differential fuzzing, when enhanced by various configurations, showed significant results. As detailed in Table 2, executing only test cases from Bitcoin Core's seed corpus successfully killed 88% of the mutants, underscoring the quality of these test cases. Interestingly, even starting without the SUT's seed corpus, the mutation score remained robust.

Table 2: Differential fuzzing setups and mutation scores

Setup (per mutant)	Mutation score
Max total time of 24hs from SUT's seed corpus	99%
Max total time of 24hs	99%
One individual test run from SUT's seed corpus	88%

While the mutation score is a valuable metric, evaluating the time required to kill mutants is essential, with a maximum allowable timeframe of 24 hours. In this experiment, killing a mutant from the SUT's seed corpus took approximately 3 minutes on average, with around 13,936 executed units. Conversely, without the project's seed corpus and without composing a new one (i.e., without reusing test cases from previous runs), the process took 2.2 hours. When a new seed corpus was introduced, effectively reusing inputs from previous runs, the average time was reduced to 6.2 minutes. Also, it is important to note that parallelization may be employed to further improve efficiency in both scenarios.

RQ1: Differential fuzzing can kill mutants within 3 minutes. Starting with a good seed corpus it is crucial for a better performance.

4.2 Analysis of Unkilled Mutants

Analysis of the mutants that survived all tests revealed that they were all equivalent. This insight suggests that fuzzing can uncover test cases that are not addressed by conventional unit tests, thereby enhancing the testing suite's coverage. If mutants were killed by any method other than differential fuzzing, it could indicate a fault in the target or oracle used for fuzz testing.

To better assess the efficacy of differential fuzzing in detecting equivalent mutants, we ran the experiments again with a timeout of three minutes per mutant, after manually excluding all equivalent mutants. This approach achieved a 97% accuracy rate when running from the project's seed corpus and 92% when started from scratch. In comparison, Table 3 presents other approaches, their respective time to classify the mutants and their accuracy.

Table 3: Approaches to identify equivalent mutants, their average time to analyze them and the accuracy

Approach	Avg time	Accuracy
Differential fuzzing with SUT's seed corpus	3 minutes	97%
Differential fuzzing without SUT's seed corpus	3 minutes	92%
Automated behavioral testing [1]	5 minutes	57% ~ 100%.
Manual analysis [11]	15 minutes	N/A

RQ2: Differential fuzzing can effectively kill mutants in a short time. To identify equivalent mutants, one approach is to set a timeout period after which any surviving mutants are considered equivalent. In our experiment, a 3-minute timeout yielded an accuracy rate of 97%. It is important to note that projects utilizing fuzzing usually maintain a robust seed corpus, which is crucial for the performance of this approach.

4.3 Specific findings

By analyzing the mutants killed only by differential fuzzing, we could notice that most of them applies the following rules:

- Relational Operator Replacement
- Constant Replacement

In particular, mutants that substituted relational operators replaced <with $\leq$ or >with $\geq$. Additionally, constant replacement predominantly involved incrementing or decrementing the original

value by one. An illustrative instance is presented in Figure 3, where Bitcoin Core's branch and bound function conduct 100,000 attempts to find the best solution. By reducing the attempts to 99,999, we need a specific case where, in fact, the best solution was found precisely in the 100,000th try. Having a unit or functional test to capture this scenario appears challenging, yet fuzzing proves adept at discovering such nuanced cases, as demonstrated in this instance.

The same mutation was applied to another function within the coin selection codebase, named *ApproximateBestSubset* [4]. Despite decreasing the total attempts by one, our differential fuzzing approach, even after running for over 24 hours, failed to kill this mutant. It means there is no need for this number of tries.

```
-    for (size_t curr_try = 0, utxo_pool_index = 0;
 ↪  curr_try < TOTAL_TRIES; ++curr_try,
 ↪  ++utxo_pool_index) {
+    for (size_t curr_try = 1, utxo_pool_index = 0;
 ↪  curr_try < TOTAL_TRIES; ++curr_try,
 ↪  ++utxo_pool_index) {
```

Figure 3: Mutant killed only by differential fuzzing

RQ3: By decreasing or increasing a constant by one or changing, for example, <to ≤, few cases can reach these changes, and they may not be addressed in the unit or functional tests. In this case, fuzzing can find those cases, and differential fuzzing can kill them.

In Groce et al. study [9], they applied mutation testing for the *tx_verify.cpp* file. In their research, fuzzing could detect just under 12% of all the generated mutants and they pointed out that only 90 of the 430 compiling mutants survived all tests (unit and functional tests), for an overall mutation score of 79.07%. After a manual inspection, they realized that 29 of them are equivalent. Their detailed list of surviving, killable mutants is available in https://github.com/agroce/bitcorpus/blob/master/mutation/prioritized_full_inspect.txt.

We applied *differential fuzzing* with the mutants from their list with the same setup as we did for the *Branch and Bound* function, limiting the execution to three minutes per mutant and with the SUT's *seed corpus*. The goal is to evaluate our approach to identify equivalent mutants. Considering the list does not contain any equivalent mutant, the number of mutants killed by *differential fuzzing* reflects its accuracy to detect equivalent mutants.

Figure 4 shows that it killed 90.25% of the mutant, which means 90.25% of accuracy. However, this is not the reality. By analyzing the survived mutants we noticed that most of them are, in fact, equivalent.

[4] https://github.com/bitcoin/bitcoin/blob/26.x/src/wallet/coinselection.cpp#L260

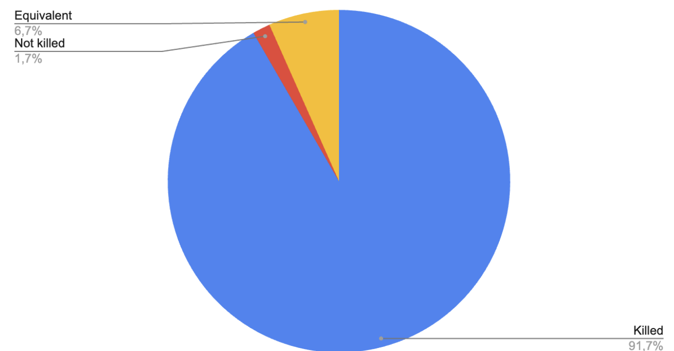


Figure 4: Mutation analysis with differential fuzzing for the mutants from Groce et al.'s list [9]

Figure 5 shows one of the mutants from the list. Although it seems a not-equivalent mutant, *nLockTime* can not be negative so this change does not have any effect. It shows that manual analysis of mutants also depends on good knowledge about the source code.. Excluding the equivalent mutants, the accuracy of our approach is 98.3% for this case, overcoming the results from our previous experiment.

```
- if (tx.nLockTime == 0)
+ if (tx.nLockTime <= 0)
```

Figure 5: Killable mutant from Groce et al.'s list [9]

5 THREATS TO VALIDITY

Threats to validity in this study include the following considerations: **Internal Validity**: The use of *universalmutator* may limit the comprehensiveness of mutants generated, as it is not specifically tailored for C++ and Rust. The experimental setup on a single hardware configuration (MacBook Pro M1 with 16GB RAM) and the manual exclusion of equivalent mutants could introduce biases and affect the results. **External Validity**: The study's focus on Bitcoin Core may limit generalizability to other projects, especially those in different domains. Results might vary with different fuzzing tools, and the specific characteristics of Bitcoin Core may not apply to other software systems. **Construct Validity**: The three-minute timeout for killing mutants is arbitrary and may not be optimal for all contexts. The effectiveness of differential fuzzing heavily relies on the quality of the seed corpus, which may not be representative across different projects. Further studies on diverse projects and using various fuzzing tools are needed to strengthen the validity of these findings.

6 CONCLUSION AND FUTURE WORK

This study explores the application of differential fuzzing for identifying equivalent mutants. Differential fuzzing is particularly adept at uncovering test cases that may be overlooked by traditional unit or functional tests. Given the prevalence of fuzz testing in many projects and the maintenance of robust seed corpuses, differential

fuzzing emerges as a highly effective and efficient method for identifying equivalent mutants, especially when utilizing the SUT's seed corpus. Additionally, we demonstrate the straightforward implementation of differential fuzzing when comparing the original code to its mutants.

This research is ongoing, and the preliminary findings are promising. Our future work will extend this research in several key areas:

(1) Expanding the replication of experiments across diverse projects and different contexts to validate the robustness and adaptability of our findings.
(2) Applying the experiments using various fuzzers to assess the impact of fuzzing tools on the outcomes.
(3) Investigating the use of differential fuzzing in conjunction with mutation testing to reveal fault-revealing mutants.
(4) Employing differential fuzzing and mutation testing to identify test cases that would enrich the existing unit test effort.

REFERENCES

[1] Samuel Amorim, Leo Fernandes, Márcio Ribeiro, Rohit Gheyi, Marcio Delamaro, Marcio Guimarães, and André Santos. 2024. Reducing Manual Efforts in Equivalence Analysis in Mutation Testing. *Journal of Software Engineering Research and Development* 12, 1 (Mar. 2024), 3:1 – 3:17. https://doi.org/10.5753/jserd.2024.3588

[2] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 351–364. https://doi.org/10.1145/3548606.3560624

[3] Claudinei Brito, Vinicius H. S. Durelli, Rafael S. Durelli, Simone R. S. de Souza, Auri M. R. Vincenzi, and Marcio Eduardo Delamaro. 2020. A Preliminary Investigation into Using Machine Learning Algorithms to Identify Minimal and Equivalent Mutants. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 304–313. https://doi.org/10.1109/ICSTW50294.2020.00056

[4] Liang Cheng, Yang Zhang, Yi Zhang, Chen Wu, Zhangtan Li, Yu Fu, and Haisheng Li. 2019. Optimizing Seed Inputs in Fuzzing with Machine Learning. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 244–245. https://doi.org/10.1109/ICSE-Companion.2019.00096

[5] Seungjoon Chung and Shin Yoo. 2022. Augmenting Equivalent Mutant Dataset Using Symbolic Execution. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 150–159.

[6] Leo Fernandes, Márcio Ribeiro, Rohit Gheyi, Marcio Delamaro, Márcio Guimarães, and André Santos. 2022. Put Your Hands In The Air! Reducing Manual Effort in Mutation Testing. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering* (<conf-loc>, <city>Virtual Event</city>, <country>Brazil</country>, </conf-loc>) *(SBES '22)*. Association for Computing Machinery, New York, NY, USA, 198–207. https://doi.org/10.1145/3555228.3555233

[7] Philipp Görz, Björn Mathis, Keno Hassler, Emre Güler, Thorsten Holz, Andreas Zeller, and Rahul Gopinath. 2023. Systematic Assessment of Fuzzers Using Mutation Analysis. In *Proceedings of the 32nd USENIX Conference on Security Symposium* (Anaheim, CA, USA) *(SEC '23)*. USENIX Association, USA, Article 254, 18 pages.

[8] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. 2018. An Extensible, Regular-Expression-Based Tool for Multi-Language Mutant Generation. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 25–28. https://doi.org/10.1145/3183440.3183485

[9] Alex Groce, Kush Jain, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. 2022. Looking for Lacunae in Bitcoin Core's Fuzzing Efforts. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 185–186. https://doi.org/10.1145/3510457.3513072

[10] Alex Groce, Goutamkumar Tulajappa Kalburgi, Claire Le Goues, Kush Jain, and Rahul Gopinath. 2022. Registered report: First, fuzz the mutants. In *International Fuzzing Workshop, ser. FUZZING*, Vol. 22.

[11] Bernhard J. M. Grün, David Schuler, and Andreas Zeller. 2009. The Impact of Equivalent Mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*. 192–199. https://doi.org/10.1109/ICSTW.2009.37

[12] Pieter Hartel and Richard Schumi. 2020. Mutation Testing of Smart Contracts at Scale. In *Tests and Proofs*, Wolfgang Ahrendt and Heike Wehrheim (Eds.). Springer International Publishing, Cham, 23–42.

[13] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 230–243. https://doi.org/10.1145/3460319.3464795

[14] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. 2019. DeepMutation++: A Mutation Testing Framework for Deep Learning Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1158–1161. https://doi.org/10.1109/ASE.2019.00126

[15] K. Jain, G. Kalburgi, C. Le Goues, and A. Groce. 2023. Mind the Gap: The Difference Between Coverage and Mutation Score Can Guide Testing Efforts. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Los Alamitos, CA, USA, 102–113. https://doi.org/10.1109/ISSRE59848.2023.00036

[16] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben Ten Hove, and Marcel Böhme. 2023. Continuous Fuzzing: A Study of the Effectiveness and Scalability of Fuzzing in CI/CD Pipelines. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. 25–32. https://doi.org/10.1109/SBFT59156.2023.00015

[17] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Józala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering* 40, 1 (2014), 23–42. https://doi.org/10.1109/TSE.2013.44

[18] Muhammad Rashid Naeem, Tao Lin, Hamad Naeem, and Hailu Liu. 2020. A machine learning approach for classification of equivalent mutants. *Journal of Software: Evolution and Process* 32, 5 (2020), e2238.

[19] Olivier Nourry, Yutaro Kashiwa, Bin Lin, Gabriele Bavota, Michele Lanza, and Yasutaka Kamei. 2023. The Human Side of Fuzzing: Challenges Faced by Developers during Fuzzing Activities. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 14 (nov 2023), 26 pages. https://doi.org/10.1145/3611668

[20] Samuel Peacock, Lin Deng, Josh Dehlinger, and Suranjan Chakraborty. 2021. Automatic Equivalent Mutants Classification Using Abstract Syntax Tree Neural Networks. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 13–18. https://doi.org/10.1109/ICSTW52544.2021.00016

[21] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2022. Practical Mutation Testing at Scale: A view from Google. *IEEE Transactions on Software Engineering* 48, 10 (2022), 3900–3912. https://doi.org/10.1109/TSE.2021.3107634

[22] Amol Saxena, Roheet Bhatnagar, and Devesh Kumar Srivastava. 2021. Improving Effectiveness of Spectrum-based Software Fault Localization using Mutation Testing. In *2021 2nd International Conference for Emerging Technology (INCET)*. 1–7. https://doi.org/10.1109/INCET51464.2021.9456109

[23] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. 2018. *Fuzzing for software security testing and quality assurance.* Artech House.

[24] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F Bissyandé, Yves Le Traon, and Koushik Sen. 2020. Selecting fault revealing mutants. *Empir. Softw. Eng.* 25, 1 (Jan. 2020), 434–487.

[25] Vasudev Vikram, Isabella Laybourn, Ao Li, Nicole Nair, Kelton OBrien, Rafaello Sanna, and Rohan Padhye. 2023. Guiding Greybox Fuzzing with Mutation Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (<conf-loc>, <city>Seattle</city>, <state>WA</state>, <country>USA</country>, </conf-loc>) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 929–941. https://doi.org/10.1145/3597926.3598107

[26] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. 2021. Finding Consensus Bugs in Ethereum via Multi-transaction Differential Fuzzing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 349–365. https://www.usenix.org/conference/osdi21/presentation/yang

Received 24 May 2024; revised 31 May 2024; accepted 5 July 2024