

# Stacking High-Level Fuzz Mutations in Big Data Applications

M.W.M. Oudemans, B.K. Özkan

TU Delft, Computer Science and Engineering

27 June 2021

## Abstract

The big data technology market size is expected to grow in the coming years. The advantages of having automated test tools for big data applications are becoming increasingly important. Fuzzing is an automated testing method which has been used in many different fields, but has not been frequently used in the big data domain as it poses several challenges. BigFuzz, a new method which was proposed by a recent study, solves these problems and shows promising results. One of the BigFuzz contributions are high-level mutations, which are error type guided and schema aware mutations. This paper is answering the question: How does stacking high-level fuzz mutations affect the test performance for big data applications? It does so by creating different stacking strategies and evaluating the effect compared to the BigFuzz method. As evaluation metrics the research looks at the amount of unique failures per trial and the distribution of unique failures found. The three stacking strategies that have been developed for this project are: Random Stack, Smart Stack and Single Stack. This research has shown that there appear to be benefits to stacking high-level mutations. The results show that stacking algorithms find on average more unique failures in less trials than a non-stacking approach. Furthermore, is Smart Stack able to find unique failures more frequently. Empirical results suggest that stacking high-level mutations can provide an advantage over only mutating once.

## 1 Introduction

The big data technology market size for 2027 is expected to grow up to 116 billion USD [1]. This increase in market share for big data applications attracts attention and it implies an expected increase in the amount of big data applications. If it would be possible to generate automatic tests for these applications it could potentially speed up development and increase software quality.

One way to automatically test software is called fuzz testing. The key of fuzz testing is that the software is tested

by automatically generating many different inputs for the program. The goal is to find as many bugs and cover as much of the code as possible. There are different ways to generate said inputs by either randomly creating bit-strings, using an example input or having a grammar describing the syntactical structure of the program. Even though fuzzing has been proven feasible and often useful in different fields [2; 3; 4; 5], it is not yet fully developed for big data applications as there are problems with applying fuzz testing on these systems [6; 7].

The work towards a fuzz testing tool for big data applications has only emerged in the past few years. In 2013 SEDGE was proposed as an example input generating tool for complex dataflow programs [8]. This generation tool was not yet applied on big data applications nor in the fuzz testing context. In 2019 BigTest surpassed SEDGE as it was a new input generation tool which could be applied to data-intensive scalable computing (DISC) systems. Moreover, compared to SEDGE, BigTest promised to reduce the amount of test data significantly while improving the user defined function path coverage [9]. However, BigTest has its limitations as it relies on the support of symbolic execution of data flow operators and the exploration capabilities of the symbolic execution engine [7].

BigFuzz has presented a new method of generating fuzz tests for DISC systems in 2020 [7]. It does this in three steps. First it creates a dataflow abstraction using source-to-source transformation with user-defined functions, then it creates an application specific coverage guidance and lastly it generates input by applying high-level mutations to an initially provided input. These are error type guided, schema-aware mutations, meaning these are designed to apply data type specific changes to program inputs to trigger common errors occurring in big data applications. BigFuzz has collected these common errors by collecting known bugs in existing programs, from GitHub and Stack Overflow. The BigFuzz paper has reported promising results compared to random fuzzing and BigTest.

A possible way to improve the BigFuzz algorithm is to stack the high-level mutations, meaning multiple high-level mutations are applied at once. This is different from BigFuzz

which always applies only one of the high-level mutations before passing it to the program that is to be tested. This new method could possibly find new, more complex bugs as the stacking of mutations can produce inputs which might never be reached by applying only single mutations. Additionally, the new method could speed up the bug finding process. Since more mutations are applied at once, more errors can be searched for at once. It can also backfire as the usage of multiple mutations can mask errors because other errors are found at the same time. This could eventually cause less errors to be found or requiring a longer amount of time to find the errors.

Whether stacking mutations can provide a benefit has led to the main question of this research: How does stacking high-level fuzz mutations affect the test performance for big data applications? To do so, the following three sub-questions have been constructed:

1. Which mutation methods are used in existing fuzzing algorithms?
2. What kind of high-level mutation strategies are used in existing fuzzing algorithms?
3. How do different kinds of stacking strategies affect the test performance?

The stacking strategies constructed during this research are Random Stack, Smart Stack, and Single Stack. These implementations stack mutations randomly, based on a set of rules, and one mutation per data point respectively.

This paper has been organized in the following way; in Section 2 the approach for each sub-question will be discussed. The algorithms developed in this research and essential parts of the algorithms will be explained in Section 3. In Section 4, the experimental setup, the literature found for the first two sub-questions and the results of the evaluation will be presented. Then in Section 5 the ethical side of the research will be discussed. Section 6 will reflect on the results found and Section 7 will be dedicated to the conclusion of the research including suggestions for possible future works.

## 2 Methodology

Below, for each sub-question the approach to answer said question is described.

### 1. WHICH MUTATIONS METHODS ARE USED IN EXISTING FUZZING ALGORITHMS?

The research started with finding existing literature and extracting the mutation methods. Collecting these methods provided a better understanding of how mutations work and provided inspiration for the implementation of new algorithms. In order to comment on the general usage of mutations and potentially the lack of literature, it was important to include all literature available in the field. A formal search query was developed and then the results of

said search query were used<sup>1</sup>.

### 2. WHAT KIND OF HIGH-LEVEL MUTATION STRATEGIES ARE USED IN EXISTING FUZZING ALGORITHMS?

For the second sub-question, again literature was collected to provide a better understanding of mutation techniques and to be an inspiration for the development of new algorithms. The same approach as for the first sub-question was used.

### 3. HOW DO DIFFERENT KIND OF STACKING STRATEGIES AFFECT THE TEST PERFORMANCE?

To answer this sub-question the research continued towards implementing an algorithm which would stack high-level mutations. Before a new algorithm was developed, the BigFuzz algorithm was implemented using the descriptions provided in the BigFuzz paper. The implementation is used as a baseline to compare with the new implementations.

We have developed three different algorithms, each implementation following a different approach but relied on the same concept of stacking high-level mutations. The implementations, Random Stack, Single Stack, and Smart Stack will be discussed in detail in section 3.3. The Smart Stack implementation requires a set of rules which describe which mutations can be stacked on top of each other. Defining the rules was done by creating a mutation matrix and reasoning about which mutations should not be stacked. To finalise the rules the Random Stack implementation was run against different benchmarks. Because this implementation generates random mutation combinations it can generate mutation combination which would not be generated by the Smart Stack. The rules were adjusted such that the random combinations that found faulty branches could still be created using the rules set. Single Stack was developed to give an insight in whether applying multiple mutations to the same column or mutating different columns was affecting the performance.

After implementation the algorithms are run on a set of benchmarks. This set of benchmarks is a sub-set of the benchmarks used in the BigFuzz paper. For the evaluation, the effectiveness and the reliability of the implementations was tested. For these characteristics the number of unique errors found per trial and the unique failure distribution are used as metrics respectively. If the distribution is low, that means there is more certainty the average amount of unique failures is found.

## 3 Stacking Mutations

This section will first describe what kind of high-level mutations have been used in this research. Then the BigFuzz algorithm is explained in more details to provide a better understanding of the stacking algorithms. Next, the new algorithms will be introduced and the contributions will be explained.

---

<sup>1</sup>If the University library did not have access to the paper or the paper was in a foreign language the result was omitted

Finally, a more detailed description of the set of rules for the Smart Stack algorithm will be presented.

### 3.1 High-Level Mutations

High-level mutations are error type guided and schema-aware mutations, meaning these are designed to apply data type specific changes to program inputs to trigger common errors occurring in big data applications. Most high-level mutations used in this research have been presented in the BigFuzz paper. Below the list of mutations and the description, as presented in the BigFuzz paper [7].

- **Data Distribution (M1):** mutate value to be either inside or outside the provided range<sup>2</sup>
- **Data Type (M2):** mutate the value such that it is the same value, but a different data type
- **Data Format (M3):** modify the column separating delimiter
- **Data Column (M4):** insert one or several characters in the value
- **Null Data (M5):** remove the value column
- **Empty Data (M6):** empty the value column

Additionally, we present a seventh mutation: adding a new column (M7). This can be seen as the counterpart of M5 and is designed such that it can discover potential new paths of the program where there are more data columns than expected. An example would be a program that tries to split a text file and then uses an IF or SWITCH statement to determine which path to take [10].

- **Data Add (M7):** add value column

### 3.2 BigFuzz Fuzzing Algorithm

The BigFuzz algorithm can be summarized in two steps. First it transforms a Spark program to a Java program, then it applies the transformed program to a fuzzing loop. A low-level description of these two processes is given below:

- The program transformation creates a dataflow abstraction to apply a source to source transformation. This transformation creates an executable class and a class for each operation performed in the Spark program. The executable class calls each operation class in the sequential order and can be run by the test framework. This process is only done once and is independent of the rest of the test process.
- When the BigFuzz fuzz process is started, a fuzzing loop is created and a the BigFuzz guidance class will be used to feed the inputs. This guidance class manages the mutation class and the evaluation of the test run. The mutation class will mutate the seed by applying a single high-level mutation to one of the input columns. The mutated value is run on the transformed Spark program and the result of the test is evaluated.

<sup>2</sup>Value is random when no input specification is provided

### 3.3 Mutation Stacking Algorithms

This work extends the fuzzy loop with stacked mutation methods. The implemented algorithms have similar functionality to each other, but differ in why and how multiple mutations are selected.

When one of the stacking algorithms is asked to apply the mutation to a provided input, it generates a list of all the mutations before applying it to the input. The number of mutations is randomly selected between one and a parameter provided by the user. Figure 1 provides an abstract visualisation of the a fuzz loop in the BigFuzz algorithm and the contribution of the new algorithms.

Per implementation, the list of mutations is generated as follows:

- **Random Stack:** the algorithm applies randomly selected high-level mutations on randomly selected input columns. There are no rules when stacking the high-level mutations. Mutations can nullify previously selected mutations.
- **Single Stack:** the algorithm applies randomly selected high-level mutations, but can only apply one mutation per input column. The number of stacked mutations is therefore limited by the amount of columns of the input.
- **Smart Stack:** the algorithm applies randomly selected high-level mutations following the defined set of rules. If an input column can not stack anymore mutations, other input columns are selected. The number of mutations being stacked is therefore limited by the combinations being made by the algorithm and the amount of columns the input has.

To nullify a mutation means the first mutation is removed by applying a new mutation (e.g. removing a value of a data point after changing the value of said data point).

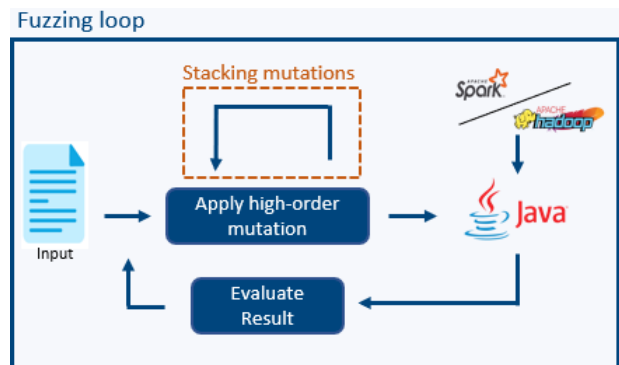


Figure 1: Visualisation of mutation stacking in a fuzz loop

### 3.4 Mutation Stacking Rule Set

The Smart Stack algorithm utilises a set of rules to determine which mutations can be stacked. Applying these rules prevents mutations nullifying each other and should decrease the

	M1	M2	M3	M4	M5	M6	M7
M1	Can be stacked	No benefit of stacking	Can be stacked	Can be stacked	Should not be stacked	Should not be stacked	Can be stacked
M2	Can be stacked	Can be stacked	Can be stacked	Can be stacked	Should not be stacked	Should not be stacked	Can be stacked
M3	Can be stacked	Can be stacked	Can be stacked	Can be stacked	Should not be stacked	Should not be stacked	Can be stacked
M4	Should not be stacked	Should not be stacked	Should not be stacked	Should not be stacked	Should not be stacked	Should not be stacked	Should not be stacked
M5	Should not be stacked	Should not be stacked	Should not be stacked	Should not be stacked	Should not be stacked	Should not be stacked	Should not be stacked
M6	Can be stacked	Can be stacked	Can be stacked	Should not be stacked	Should not be stacked	Should not be stacked	Can be stacked
M7	Can be stacked	Can be stacked	Can be stacked	Should not be stacked	Should not be stacked	Should not be stacked	Can be stacked

Table 1: Mutation rules matrix

amount of tests needed to find bugs. Reasons as to why a mutation should not be stacked on another mutation is described in Appendix A. Using the rules, a mutation matrix has been constructed which can be seen in Table 1. In the matrix, the coloured cells indicate whether the mutation on the left can be stacked on the mutations shown in the top row, e.g. the green cell next to "M2" on the left indicates "M2" can be applied to a column after "M1" has already been applied. Only the green cells are stacked. The Smart Stack algorithm looks at all mutations on the stack when excluding mutations from being stacked. e.g. The stack M5-M3-M7 is not allowed as M5 does not allow M7 to be stacked.

## 4 Evaluation

This section will first discuss how the data for the evaluation were collected. Then the findings from the literature for the first two sub-questions will be discussed. Lastly, the results for the benchmarks suite will be presented.

### 4.1 Experimental Setup

Evaluating new fuzzing implementations is not trivial. As shown by an empirical study from 2018, every fuzzing paper they considered had problems in their evaluation [11]. These problems had different causes. One of the problems had to do with the assumption that more code coverage correlates with finding more bugs. However, this might not always be the case as the correlation between the coverage of a test suite and the effectiveness of its error detection capabilities is generally low for controlled test suite sizes [12]. Other problems discovered were incorrectly counting the unique failures found, too small of a benchmark suite and using different benchmarks across papers [11]. To empirically demonstrate that a fuzzer has an advantage, one has to have a baseline fuzzer, a benchmark suite, a performance metric and a meaningful set of configuration parameters [11]. Moreover, an evaluation should be applied on multiple iterations as the results could differ due to the randomness of fuzz testing.

To avoid these common mistakes, the evaluation has been set up the following way. The new implementations have been evaluated using the BigFuzz method as a baseline. The benchmark suite that has been used was retrieved from the BigFuzz repository<sup>3</sup>. As performance metrics, the

<sup>3</sup><https://github.com/qianzhanghk/BigFuzz/>

amount of unique failures per trial and the distribution of the total unique failures was used. Each metric was collected over multiple independent runs. To prevent an incorrect count of unique failures, the stack trace of the error was used. This stack trace contained at which point the error originated and is extended until the last line of the program that is tested.

Each implementation was run for 25 independent executions, applying 5000 trials each execution. Because the process of fuzzing is dependent on randomization, each independent execution had a different number of unique failures per trial. Running the program for 25 independent runs and taking the average of all runs resulted in a relative smooth graph. Most unique failures were found within the first 1000 trials after which the number of unique failures found drops significantly. 5000 trials was used as the amount of times an input is mutated and tested. This number of trials was selected as the experiments showed that most algorithms were stabilizing in the amount of failures found per trial, while the time required to run each benchmark was feasible for this research project.

Data was collected for each of the stacking implementations and the BigFuzz implementation. For the evaluation the same stack size was used for the stacking algorithms per benchmark, but has changed per benchmark. Per benchmark a suitable stack size was selected proportional to the amount of input columns the benchmark required. The four implementations have been run on six different benchmarks.

### 4.2 Results

This section will first present the literature results for the first two sub-questions and then show the evaluation for the implementation of the third sub-question.

#### 4.2.1 Which mutation methods are used in existing fuzzing algorithms?

There are different mutation methods a fuzzer can use and it is dependent on the amount of given information about the program that is tested. When the grammar is known of the program, the fuzzer has a lot of information on what kind of inputs the program does and does not accept. From this information, the fuzzer can create a structure in which syntactical valid inputs have been derived from the grammar [13]. This is not possible when there is no grammar, as is the case in this research.

Without any information about the program, one of the most straightforward mutation methods is to begin with a random bit string and perform bit-wise mutations, i.e. changing one of the bits from 0 to 1 or vice versa. However, this method produces a lot of inputs that are syntactically invalid and can not cover a lot of the program as they are quickly rejected [14]. Instead of beginning with a random bit-string and single bit flips, one could pass an initial seed file and apply flip multiple bits. One of the earliest and most known fuzzers, AFL, uses this strategy [15]. Unsurprisingly, fuzzers that built on top of AFL and others apply this method [16; 17; 18; 19; 20; 21]. For many of these fuzzers, the differences

to the fuzzing process is mostly in what and why data is mutated, rather than how it is mutated. The benefit of these methods is that the user only has to provide an initial seed that is accepted by the program. The downside is that the bit mutations are sometimes not as effective as when the fuzzer would know what kind of datatype the program requires.

This leads to the next method, where the fuzzer applies specific mutations depending on the field or datatype the mutation is applied to. One possible method is to use a dynamic approach, where it first starts byte mutations during which it tries to discover the type, and then switches to a group specific mutation strategy [22]. This dynamic approach is not needed when the user provides the input specifications, like proposed in BigFuzz[7] and other projects [23; 24].

#### 4.2.2 What kind of high-level mutation strategies are used in existing fuzzing algorithms?

A possible strategy to apply high-level mutations is to define a set of mutation operators specific to the input domain [19; 25]. These operators are based on a special bit-wise mutation. AFLSmart, which targets programs that parse complex chunk based files, has three different operators: addition, deletion and splicing [19]. This method also allows their high-level mutations to be stacked, which is not trivial as mutations can interfere. Their solution is to keep a copy of the original and use indices which can point to a location for the mutation. If the indices overlap, no mutation is applied. However, this strategy requires the fuzzer to be custom designed for a target program.

A different approach would be to describe high-level mutation on a data type level. This approach would have a high-level description like change the data type, instead of a low level description of which bits need to be flipped. Examples for these kinds of mutations would be the mutations described in section 3.1 and the program FaFuzzer [13]. FaFuzzer would for example test for boundary values in integer values or would extend characters to a longer string.

Besides BigFuzz and FaFuzzer, no literature could be found on fuzzers that describe high-level mutation on a data type level. This is supported by a systematic literature review from 2017 which inspected more than 160 research articles and found that only 5.6 % of the research is centered on constructing more realistic high-level mutations and no research has been done on the relation between high-level mutations and real faults [26].

#### 4.2.3 How do different kinds of stacking strategies affect the test performance?

In Table 2 the benchmark names with the corresponding codes can be found which will be used in the interpretation of the results. In Figures 2 to 7 the number of unique failures per trial and the unique failures distribution at 5000 trials can be seen for the 6 different benchmarks. This section will first explain the results for the unique failures per trial and then the results for the unique failure distribution at 5000 trials.

Code	Name
B1	ExternalUDF
B2	FindSalary
B3	StudentGrades
B4	MovieRating
B5	SalaryAnalysis
B6	Property

Table 2: Benchmark names

#### Unique failures per trial

The total amount of unique failures found per trial is averaged over 25 independent runs. An implementation is considered to be outperforming another implementation if it finds on average more total unique failures in less amount of trials. The stacking implementations finds on average more bugs than the BigFuzz algorithm in 5 of the 6 benchmarks (B1, B2, B4, B5, B6). Below are the stacking algorithms compared with the BigFuzz algorithm:

- Random Stack
  - Performs better on B2, B5, B6
  - Performs similar on B1, B3, B4
- Smart Stack
  - Performs better on B2, B5, B6
  - Performs similar on B1, B3, B4
- Single Stack
  - Performs better on B2, B5, B6
  - Performs similar on B1, B3, B4

#### Unique failures distribution

The unique failures distribution is created from the total unique failures found at 5000 trials over 25 independent runs. An implementation is considered to be outperforming another implementation if the interquartile range of the boxplot is on average higher and the maximum unique failures found is at least equal. Below are the distributions of the stacking algorithms compared with the BigFuzz algorithm distribution at 5000 trials:

- Random Stack
  - Performs better on B5, B6
  - Performs similar on B1, B2
- Smart Stack
  - Performs better on B2, B5, B6
  - Performs similar on B1, B3, B4
- Single Stack
  - Performs better on B1, B6
  - Performs similar on B2, B4

### 4.3 Empirical Results Summary

There are two observable benefits of stacking mutations. First, more unique failures are found in less trials as each implementation performed better in three benchmarks, and the same in the other three benchmarks. Secondly, the stacking of mutations could find the unique failures at 5000 trials

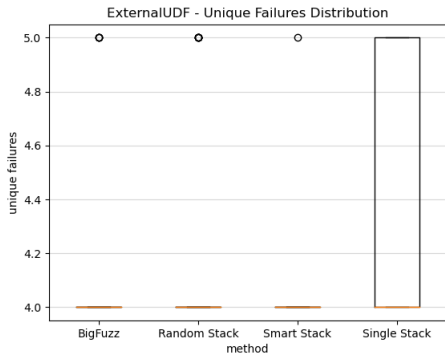
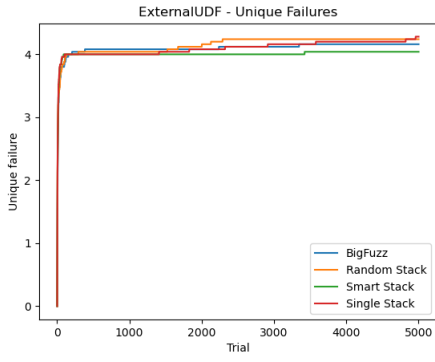


Figure 2: ExternalUDF benchmark results

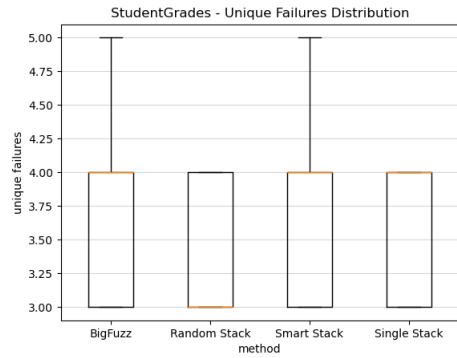
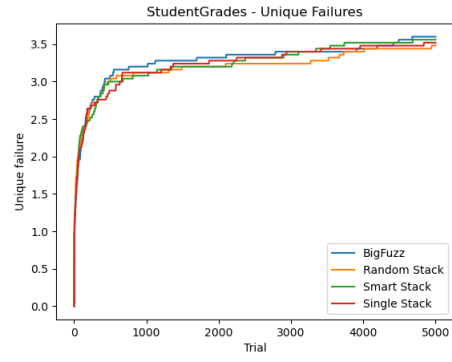


Figure 4: StudentGrades benchmark results

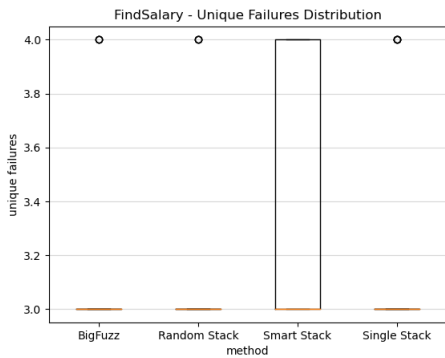
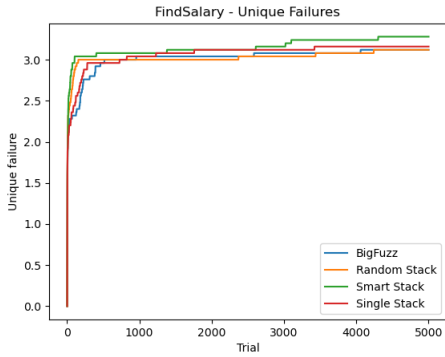


Figure 3: FindSalary benchmark results

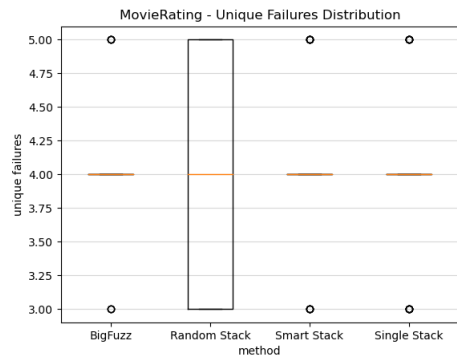
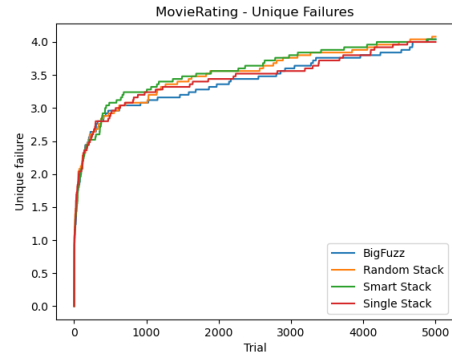


Figure 5: MovieRating benchmark results

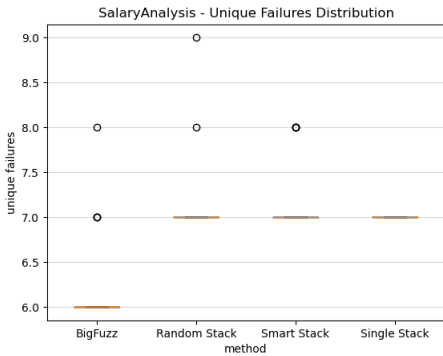
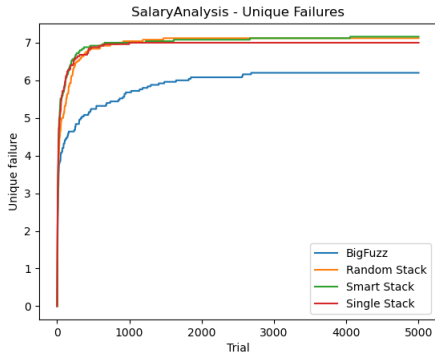


Figure 6: SalaryAnalysis benchmark results

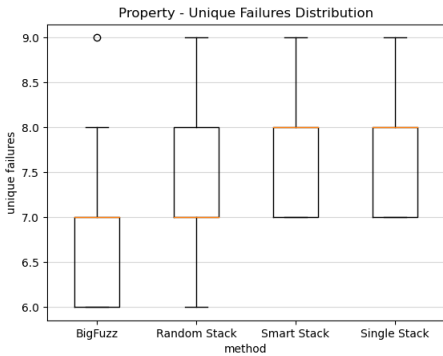
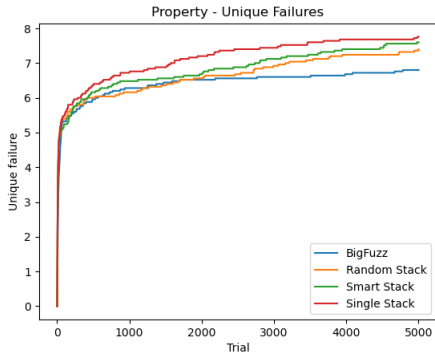


Figure 7: Property benchmark results

more reliably as the distribution of the Smart Stack performs better in three benchmarks, and the same in the other three benchmarks. Which stacking strategy performs best is unclear as the best algorithm differs per benchmark. In general, the Smart Stack algorithm performs better than the other implementations over all six benchmarks, but the differences between the stacking algorithms are negligible.

## 5 Responsible Research

In this section the ethical aspects of the use of the proposed test method will be expanded on first. Then, the reproducibility of BigFuzz will be discussed and finally the reproducibility of this project will be explained.

### 5.1 Ethical Reflection

A possible reaction to the reported results is to believe that this method removes the need for tests written by the developer. While these results might be promising in the field, it is not yet thoroughly tested. One can not assert that this method is finding every bug in a program and might actually be deceiving as described in section 6. Moreover, the new implementations have only been tested on a few benchmarks for a limited amount of high-level mutation focusing on a limited amount of error types. There is more work required to make a robust tool for developers which can produce exhaustive testing capabilities. Even when such a tool is created, the developer should never fully rely on an automated testing tool.

### 5.2 BigFuzz Reproducibility

As this research is built upon the BigFuzz paper, discussing the reputability of said research is important as it has had an impact on the reproducibility of this research paper. The BigFuzz paper has made a very good effort in collecting information from various sources and establishing a method for testing big data applications which looks promising. Their contribution is making a big step forward in this field. However, to reproduce and expand their prototype implementation was very challenging. The BigFuzz paper referred to their Github repository containing the artifacts that have been used in their implementation, which is incomplete and seems to be in development. After thoroughly investigating the code the following unresolved problems have been found:

- Mutation methods seemed to be benchmark specific and are not schema aware. While in the paper they present six different mutations operations, there are mutation implementations per benchmark where the code has been manually changed and optimized to fit the benchmark. Many times only a subset of the high-level mutations was used rather than all the high-level mutations.
- The source-to-source transformation tool did not work as it required a UDF generator tool which was not included in the repository. An attempt was to implement the UDF generator tool from the BigTest repository<sup>4</sup>, but without success. This means that the Spark benchmarks could not be generated and existing benchmarks from

<sup>4</sup><https://github.com/maligulzar/BigTest/>



the BigFuzz repository, which are not reproducible, had to be used.

### 5.3 Reproducibility

During this research, special attention was given to the reproducibility. Both the literature research, implementation and evaluation have been described in detail in this paper.

For the first two sub-questions, formal search plans have been created to collect the literature available. Sections 2 and 3 can be used to recreate the implementations created for this research. These implementations can also be found on a copy of the research repository<sup>5</sup>. This repository contains other contributions by the same research group who were working on other methods. The repository contains a Readme file which contains more technical details as to how to run the project. On the repository the data used to construct the results of this paper can be found. Additionally the seeds used to produce the data can be found to reproduce the results. Not all data produced by the algorithm is stored due to storage restrictions.

## 6 Discussion

This section will expand on some of the limitations and observations concerning this research.

The results that have been reported in the previous section have been collected by running the benchmarks provided by BigFuzz. It is not known whether these benchmarks are an accurate representation of the possible DISC systems that should be tested. Therefore, it is possible that this research has an incomplete sample of DISC programs and other programs could yield different results.

The stacking algorithms were given the best configurations per benchmark such that it was performing well. It has not been tested how the implementations behave with other configurations.

The implementations that have been made and tested do not have an input specification and are therefore not schema aware. The BigFuzz method had to be recreated from the methods described in the paper and due to time limitations of the project the input specifications could not be added.

One of the noteworthy observations made during evaluation is the uniqueness of the failures found. In some benchmarks the StackTrace of the found error was empty. This issue seemed to only appear for one certain kind of error and would only appear if the said error was already found in a previous iteration. During development the issue was investigated and seemed to originate from the frameworks on which the implementation was built. The issue could not be resolved during this research, because it was a too time consuming task due to the complexity of the framework. The issue can be reproduced using the code and instructions

<sup>5</sup><https://github.com/moudemans/StackedMutation>

provided on one of the branches of the repository<sup>6</sup>. Because the issue was occurring for every implementation the results for comparison were not effected. The amount of actual unique failures for each implementation might be incorrect and should not be used as a measure to the total amount of errors. The issue caused an incorrect unique failure count in the results in B5 for the Random Stack implementation. The 9th unique failure in one of the runs was not really a unique as it did not have a StackTrace.

## 7 Conclusions and Future Work

This section will first discuss the sub-questions. Then it answer the main research question and present the conclusion of this research. Finally, it will provide some recommendations for future work.

In this paper the different kinds of mutation strategies that are used in existing fuzzing algorithms was researched. The knowledge gained from the literature has helped in the development of new implementation strategies.

While there are examples of methods using high-level mutation strategies, these are often bit-wise and do not describe a high-level mutation on a data type level. These high-level of mutations are centered on constructing more realistic high level mutations and have a relation to real software faults. Only a small portion of the literature in this field discusses the high-level mutations and its relation to real software faults.

As for how the different kind of stacking strategies affect the performance, from the results can be concluded that stacking high-level mutations can provide an advantage over only mutating once. In almost all benchmarks one of the stacking algorithms outperformed the traditional one mutation method. The stacking of mutations shows two advantages. First, it performs better as more unique failures are found in less amount of trials. Second, the Smart Stack implementation is more reliable as the algorithm is able to find more unique failures more frequently.

This research has provided a potential benefit of stacking high-level mutations. A future study could apply the new proposed method on more benchmarks. Furthermore, one could improve the proposed stacking methods by further developing the stacking rules and using biased high-level mutations.

## References

- [1] A. Arora, "Big data technology market 2021 top manufacturers, industry size, global demand and covid 19 impact on revenue growth," May 2021.
- [2] A. Alsharif, G. Kapfhammer, and P. McMinn, "Domino: Fast and effective test data generation for relational database schemas," pp. 12–22, 2018.

<sup>6</sup>[https://github.com/moudemans/StackedMutation/tree/MO\\_bugReport.1](https://github.com/moudemans/StackedMutation/tree/MO_bugReport.1)



- [3] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” pp. 209–224, 2019.
- [4] B. Ghit, N. Poggi, J. Rosen, R. Xin, and P. Boncz, “Sparkfuzz: Searching correctness regressions in modern query engines,” 2020.
- [5] F. Langner and A. Andrzejak, “Detection and root cause analysis of memory-related software aging defects by automated tests,” pp. 365–369, 2013.
- [6] E. M. Fredericks and R. H. Hariri, “Extending search-based software testing techniques to big data applications,” pp. 41–42, Association for Computing Machinery, Inc, 5 2016.
- [7] Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, and M. Kim, “Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction,” in *Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*, pp. 722–733, 2020.
- [8] K. Li, C. Reichenbach, Y. Smaragdakis, Y. Diao, and C. Csallner, “Sedge: Symbolic example data generation for dataflow programs,” pp. 235–245, 2013.
- [9] M. Gulzar, S. Mardani, M. Musuvathi, and M. Kim, “White-box testing of big data analytics with complex user-defined functions,” pp. 290–301, 2019.
- [10] C. Fregly, “Arrayindex out of bound exception.” <https://forums.databricks.com/questions/585/how-do-i-get-around-an-arrayindexoutofboundsexcept.html>, Apr 2015.
- [11] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing,” pp. 2123–2138, 2018.
- [12] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness,” pp. 435–445, IEEE Computer Society, 5 2014.
- [13] J. Wang, B. Chen, L. Wei, and Y. Liu, “Superior: Grammar-aware greybox fuzzing,” vol. 2019-May, pp. 724–735, 2019.
- [14] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, “Breaking things with random inputs - the fuzzing book.” <https://www.fuzzingbook.org/html/Fuzzer.html>, 2019.
- [15] M. Zaleski, “Binary fuzzing strategies: what works, what doesn’t.” <https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html>, Aug 2014.
- [16] C. Lemieux and K. Sen, “Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage,” pp. 475–485, 2018.
- [17] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, “Saff: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing,” pp. 61–64, IEEE Computer Society, 2018.
- [18] S. Rawat and L. Mounier, “Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results,” pp. 531–533, 2011.
- [19] V. T. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, “Smart greybox fuzzing,” *IEEE Transactions on Software Engineering*, 2019.
- [20] M. Bohme, V.-T. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as markov chain,” *IEEE Transactions on Software Engineering*, vol. 45, pp. 489–506, 2019.
- [21] M. Böhme, V. T. Pham, M. D. Nguyen, and A. Roychoudhury, “Directed greybox fuzzing,” pp. 2329–2344, Association for Computing Machinery, 10 2017.
- [22] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, “Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery,” vol. 2019-May, pp. 769–786, Institute of Electrical and Electronics Engineers Inc., 2019.
- [23] L. Liu, X. Huang, A. Zhou, P. Jia, and L. Liu, “Fuzzing the android applications with http/https network data,” *IEEE Access*, vol. 7, pp. 59951–59962, 2019.
- [24] Y. Wang, Z. Wu, Q. Wei, and Q. Wang, “Field-aware evolutionary fuzzing based on input specifications and vulnerability metrics,” vol. 2019-October, pp. 226–232, IEEE Computer Society, 2019.
- [25] X. Wang, C. Hu, R. Ma, D. Tian, and J. He, “Cmfuzz: context-aware adaptive mutation for fuzzers,” *Empirical Software Engineering*, vol. 26, 2021.
- [26] A. Ghiduk, M. Girgis, and M. Shehata, “Higher order mutation testing: A systematic literature review,” *Computer Science Review*, vol. 25, pp. 29–48, 2017.

## A Mutation rules reasoning

First mutation	Second mutation	Reason
M1	M1	Changing the value twice does not provide a benefit because the value is random.
M1	M2	No reason to not allow it.
M1	M3	No reason to not allow it.
M1	M4	No reason to not allow it.
M1	M5	Removing the column after the value has been changed nullifies the first mutation
M1	M6	Emptying the column after the value has been changed nullifies the first mutation
M1	M7	No reason to not allow it.
M2	M1	No reason to not allow it.
M2	M2	Changing the datatype twice does not provide a benefit as the first datatype change appears to have never happened
M2	M3	No reason to not allow it.
M2	M4	No reason to not allow it.
M2	M5	Removing the column after the datatype has been changed nullifies the first mutation
M2	M6	Emptying the column after the datatype has been changed nullifies the first mutation
M2	M7	No reason to not allow it.
M3	M1	No reason to not allow it.
M3	M2	No reason to not allow it.
M3	M3	Changing the delimiter twice does not provide a benefit as the first delimiter change appears to have never happened
M3	M4	No reason to not allow it.
M3	M5	No reason to not allow it.
M3	M6	No reason to not allow it.
M3	M7	No reason to not allow it.
M4	M1	Changing the value after a random insert nullifies the first mutation
M4	M2	Changing the datatype after a random insert does not make a difference as the column was already changed to a string with the first mutation
M4	M3	No reason to not allow it.
M4	M4	No reason to not allow it.
M4	M5	Removing the column after a random insert has been done nullifies the first mutation
M4	M6	Emptying the column after a random insert has been done nullifies the first mutation
M4	M7	No reason to not allow it.
M5	M1	Changing the value after the column has been removed is not possible
M5	M2	Changing the value after the column has been removed nullifies the first mutation
M5	M3	No reason to not allow it.
M5	M4	Adding a random character after the column has been removed is not possible
M5	M5	Removing the column after the column has been removed is not possible
M5	M6	Emptying the column after the column has been removed is not possible
M5	M7	Adding a new column after removing one nullifies the first mutation
M6	M1	Changing the value after the column has been emptied nullifies the first mutation
M6	M2	Changing the datatype after the column has been emptied is not possible
M6	M3	No reason to not allow it.
M6	M4	Adding a random character after the column has been emptied nullifies the first mutation
M6	M5	Removing the column after the column has been emptied nullifies the first mutation
M6	M6	Emptying a column that has already been emptied does nothing
M6	M7	No reason to not allow it.
M7	M1	No reason to not allow it.
M7	M2	No reason to not allow it.
M7	M3	No reason to not allow it.
M7	M4	No reason to not allow it.
M7	M5	Adding a column after a column has been added the first mutation
M7	M6	No reason to not allow it.
M7	M7	Adding a column after a column has been added targets the same error and does not provide a known benefit




	Mutations do not interfere
	Mutations can interfere if the mutation is performed on the same column
	No benefit of again mutating

Table 3: Mutation rules reasoning