Systematically Applying High-Level Mutations for Fuzz Testing Big Data Applications

Lars van Koetsveld van Ankeren , Burcu Kulahcioglu Ozkan
Software Engineering Research Group
Technical University Delft
L.vanKoetsveldvanAnkeren@student.tudelft.nl, B.Ozkan@tudelft.nl

Abstract

As the amount of data worldwide continues to grow the big data field is becoming increasingly important. Fuzz testing has shown to be an effective testing tool, and recent work has applied fuzz testing to big data applications. This study aims to contribute to knowledge on fuzz testing big data applications by extending on BigFuzz, a state-of-the-art fuzzing framework for big data applications. Our study offers an alternative mutation approach by systematically applying combinations of seven high-level mutation types, instead of selecting mutations randomly. Our findings show that 1) for three out of five benchmarks, systematic exploration finds a higher number of failures; 2) the amount of trials needed to find an equal number of failures is not increased by testing systematically for the majority of the benchmarks; 3) our configuration returns the best results when it explores with increased exhaustiveness; Thus, we show that systematically applying high-level mutations can find a higher number of unique failures in an equal number of trials.

1 Introduction

Data-intensive scalable computing (DISC) applications such as Apache Spark [1] are becoming increasingly important as the amount of data continues to grow worldwide. The importance of testing these applications is rising accordingly, emphasizing the relevancy of this field of study.

A testing method that can be applied to DISC systems is fuzz testing. Fuzzing frameworks have emerged as a popular testing tool, demonstrated by the success of AFL [2] and the discovery of the Heartbleed bug by fuzz testing [3]. Fuzz testing frameworks test applications by repeatedly mutating - altering - an input seed and consequently running the application with the mutated input.

For this purpose BigFuzz was implemented, a fuzz testing framework for Spark applications [4]. The six mutation types introduced by BigFuzz are error-type guided based on six common types of errors for Spark applications. BigFuzz applies one random high-level mutation on the seed input every trial run. An alternative approach is to combine mutations by applying them consecutively over separate runs. This a form of higher order mutation (HOM) testing, defined in 2009 by Jia and Harman, referring to mutants with multiple faults injected instead of one [5]. For a small number of possible mutations (e.g. six for BigFuzz), systematically applying high-level mutations may find additional unique failures in the program under test (PUT).

However, there is a lack of research on the possible benefits of this approach. HOM testing generally applies multiple mutations in the same run, instead of applying mutations in consecutive runs. Additionally, related research on HOM testing usually restrict the search space, due to the significant amount of possible mutants [6, 7]. In contrast, search techniques are not required for small amounts of possible mutations.

Furthermore, few research has been conducted on exploring alternative approaches to fuzz testing big data applications [8–10]. Out of these studies the focus of Olston et al. [8] lied on the generating of input data for dataflow applications, and the research conducted by Li et al. [10] and Gulzar et al. [9] aimed to test dataflow applications symbolically, which is a manner of testing that can not be applied to all data centric applications [4]. Therefore this study aims to fill the gap in the literature by providing knowledge of the performance of fuzzing DISC applications with a systematic testing approach.

The main question that this study will aim to answer is the following: 'How does systematic exploration of high-level mutations affect the performance of a fuzz testing framework?'. The main research question can be further subdivided into the following sub-questions:

1. How can high-level mutations be applied systematically?
2. How does systematic exploration perform when compared to random selection of mutations?
    - What is the difference in numbers of errors found?
    - What is the difference in number of runs needed?
    - How does the configuration of the exploration influence the performance?
3. Which program properties determine the performance of systematic exploration?

The following sections of this paper give an overview of the performance of systematic exploration by answering these research questions. The first section below will extend on necessary background information for this study. Following it,

sections four and five line out the methodology and contribution of this research. Section five contains the results and experimental set-up for replicating this study. Encountered limitations and ethical aspects related to the performed research are included in section six. Finally, the last section seven outlines conclusions on the conducted research.

2 Background

The following three section provide the background information on the most relevant parts of our study.

2.1 Fuzzing

Fuzzing is a test generation technique. It is a form of random testing, creating different inputs with the goal of finding unexpected behaviour. In a survey on fuzzing conducted by Manes et al. fuzzing is defined as the following: "the execution of the program under test using input(s) sampled from an input space ... that protrudes the expected input space of the program under test" [11]. Different versions of fuzzing software exist, distinguished mainly by the amount of analysis used to guide the fuzzing framework. Blackbox fuzzing refers to frameworks that perform fuzzing without first analysing the program and thus having only random or unguided fuzzing. See this paper by Woo et al. for a mathematical definition of blackbox fuzz testing [12] and the research of Kim et al. for a scenario in which blackbox fuzzing was chosen over whitebox [13]. Contrarily, whitebox fuzzing frameworks perform an in-depth analysis of the PUT and are in this manner able to explore the PUT systematically. The research of Bounimova et al. is an example of whitebox fuzzing in production [14]. Greybox fuzzing lies between blackbox and greybox, analysing only a limited amount. AFL is an example of a greybox fuzzer [15], as well as the studies extending on it [16]. Since this study does not analyse the PUT, our fuzzer is a form of blackbox testing. However, some whitebox approaches also use a systematic search.

2.2 BigFuzz

This study extends on the BigFuzz study from 2020 [4]. BigFuzz was introduced as a fuzzing framework for Spark applications. The fuzzing framework is built upon JQF, "a platform for performing coverage-guided fuzz testing in Java" [17]. BigFuzz utilizes the event tracking of JQF to abstract coverage guidance of dataflow operators. This is the first of three components that enable BigFuzz to make coverage-guided fuzz testing possible for big data applications. The second component consists of rewriting the Spark application to a Java application, hereby reducing the latency of running the tests. This is very effective since initiating Spark requires over 10 seconds per test. The third component is the manner in which mutations are generated in BigFuzz. It attempts to generate valid inputs and mutates schema-aware and error-type guided, which is how it can resemble real-world errors. Using such a schema improves performance by avoiding crashes early on. To be able to represent these errors Zhang et al. investigated the most common types of errors and how these can be replicated in their framework. These three components have together enabled BigFuzz to lay the the foundation for coverage-guided fuzz testing for Big Data analytics, which this study builds upon.

3 Methodology

This paper has the following methodologies defined per sub-question.

The first research question, how can high level mutations by applied systematically, can be answered by performing a literature search for means to systematically apply all combinations of mutations. This search consists of two parts: first, what research has been conducted on HOM testing; second, which algorithms exist to explore the mutation space. By combining the findings to these two questions an approach to systematic mutation with seven mutations can be formulated.

For the second research question, how does systematic exploration perform in comparison to random selection, several metrics can be measured for an indication of performance differences. In the literature the most used measure for evaluating performance is the number of unique crashes found per run [18]. This amount can be misleading however, since it depends on the ability of the fuzzing tool, in this case the BigFuzz application, to remove crashes with equal causes. Despite this issue, finding bugs is the purpose of a fuzzing tool, and is therefore the main evaluation metric for this study. Another metric that could be evaluated is the coverage of the fuzzing framework, however this is difficult to evaluate for the BigFuzz framework, since it currently does not include an analysis of the total amount of paths or branches of the PUT.

To answer the third and last research question, on which program properties determine the performance of systematic exploration, the benchmarks that were used by the authors of the BigFuzz paper are analysed and compared. These benchmarks differ in several aspects that can influence performance, such as the amount of columns in the input file and the input specification. By linking the performance per benchmark to these properties, conclusions can be drawn on when the fuzzing framework performs best.

4 Systematic Application of Mutants

Our approach to fuzzing DISC applications aims to explore combinations of high level mutations systematically, contrary to applying mutations randomly on the seed input every run. Exploring combinations in this manner can be achieved only for a small amount of mutation types, since the number of combinations increases exponentially with the amount of times mutations are combined. Section 4.1 shows related research which our approach is based on.

Our implementation is an extension of the BigFuzz functionality that was available in the repository of the authors of the BigFuzz paper [19]. Similar to the BigFuzz framework we apply mutations for a set number of trials, however instead of applying mutation to one initial input we also use mutants as input. Our implementation applies seven mutations of which six were based on the BigFuzz mutations, listed in Table 1. The seventh mutation M7 was added to replicate null errors similar to M5. The first level of mutations are applied to an input seed of the application, which generally consist

ID	Mutation	Description	Column specific
M1	Data value	Change data to arbitrary integer.	Yes
M2	Data type	Change data type from float to string, integer to float or string to integer.	Yes
M3	Data format	Change delimiter to "~" or if "~" to ",".	No
M4	Data column	Insert arbitrary character in data.	Yes
M5	Null data	Remove column from a row of data.	Yes
M6	Empty data	Mutate data to empty string.	Yes
M7	Add data	Add a column to row of data.	No

Table 1: Systematically applied mutation types

of multiple columns of data. Following mutations are then applied on the previously mutated input. Mutations are applied either on all columns for M3 and M7 or one specific column for the other mutation types. The number of explored combinations of mutations depend on the configuration of our fuzzing tool, which Section 4.2 will elaborate on. Our fuzzing tool runs until a set number of trials is reached.

4.1 High-Level Mutation Algorithms

To answer our first research question, how can high-level mutations be applied systematically, we searched for literature answering the following questions: what research has been conducted on HOM testing; which algorithms exist to explore the mutation space. The findings for these searches are the foundation our approach is built upon.

The first question concerns research conducted on HOM testing, since it is similar to our research in that multiple mutations are applied. However, it is different in that most studies apply multiple mutations in one run, instead of combining mutations over multiple runs as can be done for a small number of mutation types.

The question is answered by two studies published in 2017 and 2019 listing higher order mutation studies [6, 20]. The study by Ghiduk et al. is the first literature review conducted on HOM testing, listing research objectives, techniques and outcomes. Two of their findings are strategies for handling the issue of the great number of HOMs that could be generated by combining mutations. The first strategy is the use of search based techniques, for instance the genetic algorithm; the second strategy is to reduce the number of HOMs. This first strategy is not applicable to our research, since our aim is to search the mutation space systematically. For the second strategy the study found three ways researches had reduced the amount of HOMs: (1) reduce the number of mutation operators; (2) select a subset of HOMs based on an attribute; (3) reduce the number of locations where the original program can be mutated. For our study the number of mutation operators can not be reduced further, since it is already a minimum number of high-level mutation types representing real-world errors. Additionally, our approach does not mutate at a location in the program since it is a fuzz testing tool, instead applying mutations to a seed input. However, we can reduce the amount of combinations of mutation types by excluding certain combinations that do not seem beneficial to explore. These exclusion rules are explained in section 4.2. The research performed by Lima et al. is similar to the literature review from Ghiduk et al., although it is more recent and focusses on HOM generation and selection techniques. The goal of HOM generation in most studies is to cover multiple first order mutations (FOMs) in less test runs, one of the most successful methods is described by Polo et al. [21]. However, for this study our aim is to cover more edge cases, which is why we choose not to generate only a subset of mutants.

The second aim of our search was to find algorithms for exploring combinations of high-level mutations. Since our exploration is not search based, our approach needed a traversal for the entire set of possible mutants. We choose to employ a pre-order depth-first algorithm, since it can be used for a bounded exhaustive search of the mutation space [22].

4.2 Systematic Exploration

To explore the mutation space, our approach uses a data tree combined with a depth-first pre-order search. An overview of this search is shown in Figure 1, however in practice some nodes may be excluded, which the next two parts describe. On every test run the tree is traversed for the next mutation type to be applied for generating the next input and is subsequently tested on the PUT. The top node represents the columns of the input seed file. The children of a node represent the next mutations to be applied, and mutate upon the data of their parent. The nodes shown contain fields for the previously applied mutations, their level in the tree and the column to apply the mutation to. The mutated data is not stored in the nodes of the tree, instead all columns are saved once for every level in the tree to be more space efficient. For this reason the tree is traversed with a depth-first search, instead of a breadth-first exploration. Since any child mutates upon the data of its parent, a child node needs only access to its parent's mutation data. In the case of a breadth-first search, the data of the entire level in the tree would need to be stored. This number increases as the number of nodes per level grows exponentially. When the entire tree is explored, the tree restarts from the first mutation again, until the maximum amount of trials is reached.

Tree construction. The tree structure is not generated in its entirety when it is constructed. Instead, only the root node is added and children are generated lazily by constructing and adding children to a node when the nodes are visited for the first time. The root node does not contain a mutation type but exists only to traverse the rest of the tree. Children are removed from their parents when they are traversed, to avoid applying the same combinations of mutations repeatedly. Furthermore, a child is only generated for combinations of mutation type and columns that have not been ap-

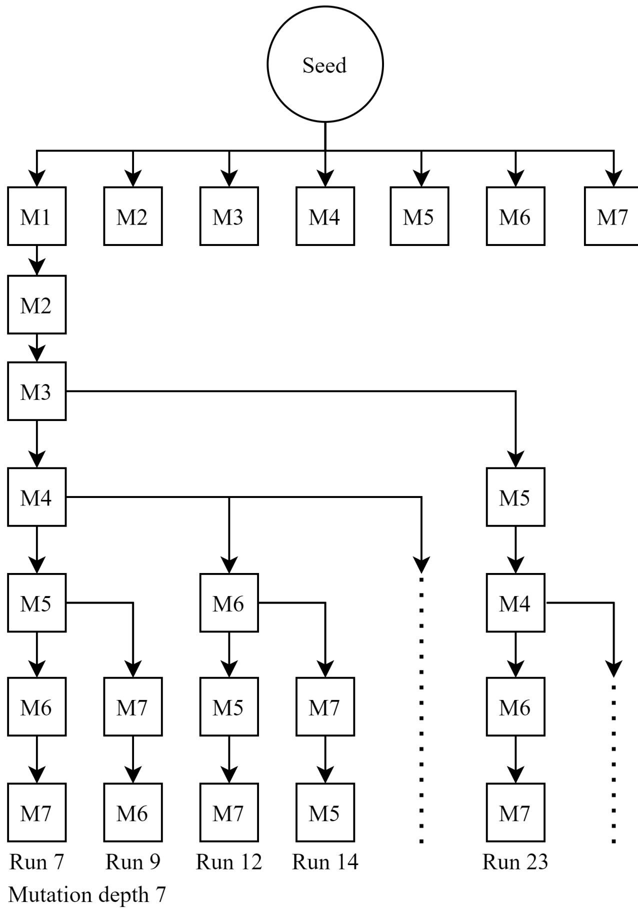

Figure 1: Mutation tree with column exploration turned off, therefore each mutation is only applied to one column of the input seed file. Some nodes could be excluded when applied to the same column.

plied by either its parents or itself. In addition, children are excluded for sequences of combinations without added exploration value. For instance: when a value is changed for a column, and continuing the column is set to an empty string, the effect of the changed value is nullified. The exclusion rules are listed in Table 2. As can be seen in the table, mutations exclude themselves when applied to the same column. Note that some mutations are not column specific: M3 and M7. These mutations are always excluded if they occurred previously, however the remainder of the mutations are only excluded when applied to the same column as previous mutations. To implement this behaviour child nodes that fit the exclude rules are removed from the parent node. The exclusion rules bound the exhaustiveness of the systematic search, however also improve efficiency and increases the number of times the exploration of the tree is restarted.

Configuration. The systematic exploration can be configured by two parameters to control the size of the tree and by one parameter to set the maximum number of testing trials.

The first parameter is a boolean to indicate whether to explore each mutation per column, for the mutations that apply to all columns. This parameter determines how children are

Excludes	M1	M2	M3	M4	M5	M6	M7
M1	■				■	■	
M2	■	■			■	■	
M3			■				
M4	■	■		■	■	■	
M5	■	■		■	■	■	■
M6	■	■		■	■	■	■
M7					■		■

Table 2: Exclusion rules

generated. If column exploration is turned off, children are generated for a random column and with mutation types that have not been applied on any column by the parents of the child node. The exclusion rules still apply however, which can lead to mutations being discarded. If column exploration is turned on, a child node is generated for each column. Instead of checking which mutation types have been applied to any column by the parents of the node, only the previous mutation types for the column of the child node are used. Thus the amount of nodes generated is increased by the amount of columns of the input file.

The second parameter is the maximal depth of the tree. The tree does not generate any additional children for a mutation at the level of the maximal depth. When column exploration is turned of, the maximum depth of the tree becomes seven, since this is the amount of possible mutations.

The last parameter determines the amount of times mutants are tested on the PUT.

5 Empirical Evaluation Results

Our evaluation aims to compare the performance of systematic exploration to random mutation in the following aspects:

1. What is the difference in numbers of errors found?
2. What is the difference in number of runs needed?
3. How does the configuration of the exploration influence the performance?

The random selection approach consists of applying one arbitrary mutation type to the seed input for every test run. The difference in number of errors found is the difference in number of errors found at the end of program. The number of runs refers to the amount of trials before the amount of errors starts to converge. Lastly, the evaluation of the configuration assesses how the column exploration and mutation depth, detailed in Sec 4.2, impact the results.

Besides providing answers to these questions, we aim to answer our third research question, on which program properties systematic exploration performs different compared to random mutation.

5.1 Benchmarks

Experimental setup. To evaluate the performance we applied five benchmarks compiled from various sources, listed in Table 3. Several of these benchmarks are from authors of the BigFuzz study and others originated from public applications or the BigTest repository [4]. We choose to use

ID	Program	Output
P1	IncomeAggregation	Average income per age range in a district
P2	StudentGrade	List of classes with more than 5 failing students
P3	MovieRating	Total number of movies with rating ≥4
P4	FindSalary	Total income of individuals earning ≤$300 weekly
P5	InsideCircle	Check whether the point (x,y) is in a circle

Table 3: BigFuzz benchmarks

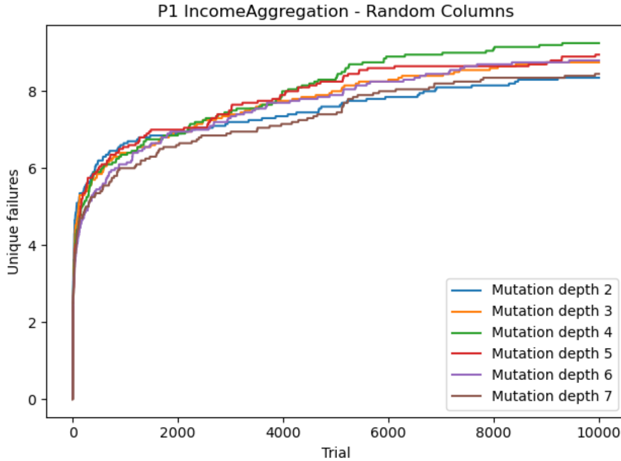

Figure 2: Benchmark P1 without column exploration and varying tree depths.

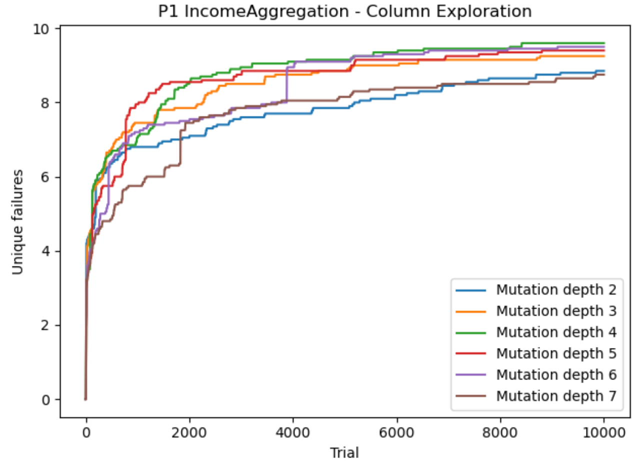

Figure 3: Benchmark P1 with column exploration and varying tree depths.

the BigFuzz benchmarks for comparison purposes, evaluating the benchmarks present in the BigFuzz repository [19]. The setup used was twenty iterations per benchmark with a trial length of 10000. The evaluation metric is the number of failures, which are unique unexpected exceptions thrown by the PUT. To compare our performance with a random mutation selection approach, our implementation contains an alternative method applying one arbitrary mutation type to the seed input for every test run.

Benchmark P1. The input of benchmark P1 contains three columns and has been tested extensively to show the difference for varying mutation depth settings. Figures 2 and 3 show the amount of unique failures found for this benchmark on average with and without column exploration. The graph without column exploration is more smooth than with column exploration, since randomly selected columns lead to different mutations between iterations. The figures show that column exploration finds a higher amount of unique failures for this benchmark overall. Furthermore a tree depth of 4 leads to the best results for both column exploration and random column selection. The performance with a tree depth of 3, 5 and 6 are similar to 4, a tree depth of 2 or 7 finds less unique failures. The following benchmarks are evaluated with a tree depth of 4 based on this result.

Figure 4 compares these results with random selection. On average systematic mutation is able to find 50% more unique failures than random mutation with column exploration. Systematic mutation without column exploration also reaches a higher number of failures compared to random mutation. The number of trials needed before converging is about 5000 trials for systematic exploration in comparison to 1000 for random mutation.

Benchmark P2. The seed for this benchmarks contains two columns. Figure 5 shows the evaluation results for this benchmark. As can be seen in the figure the difference between the number of errors found after 10000 trials is small. However, random selection finds failures in less trials when compared to systematic exploration overall. For this benchmark column exploration performs similar to the random column configuration.

Benchmark P3. The MovieRating benchmark P3 is a benchmark with two columns, and three delimiters. This made it a difficult benchmark to test for our implementation, since it stores only one delimiter. As Figure 6 shows, the difference in results is again less than one unique failure on average. Random mutation finds a 16% higher number of failures in 10000 trials compared to systematic exploration. In contrast to other benchmarks the amount of errors does not converge before 10000 trials for random mutation, however it does for systematic exploration - around 6000 trials.

Benchmark P4. The FindSalary benchmark P4 is a benchmark containing only one column. Therefore there is no difference between mutating with or without column exploration. Figure 7 provides the number of unique failures found for this benchmark. The results show that systematic exploration finds 38% more unique failures on average compared

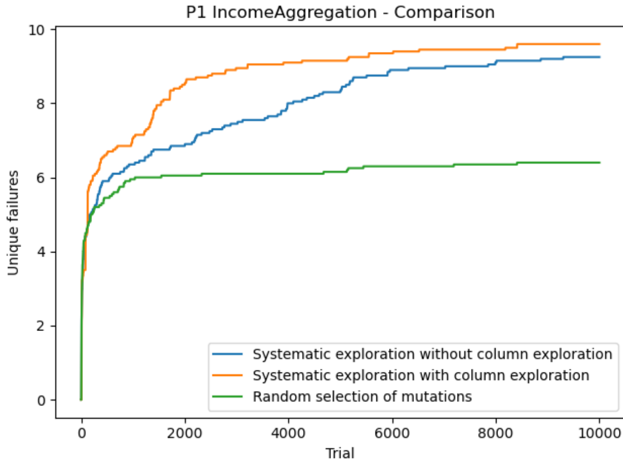

Figure 4: Benchmark P1 with systematic mutation compared to random mutation. The systematic exploration tree depth is 4.

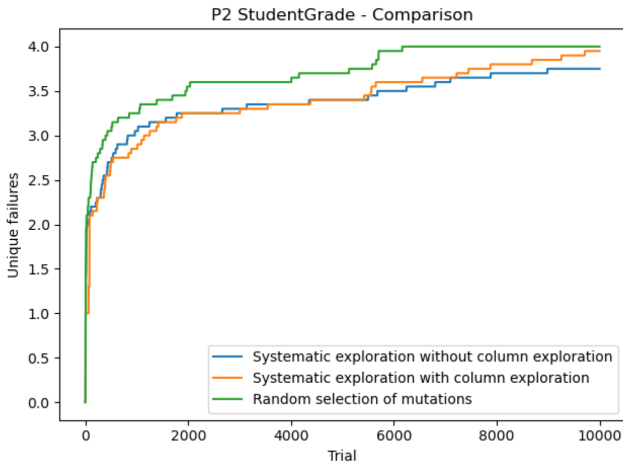

Figure 5: Benchmark P2 with systematic mutation compared to random mutation. The systematic exploration tree depth is 4.

to random mutation. Both results converge at a similar number of trials.

Benchmark P5. Figure 8 shows the results for benchmark P5. For this benchmark systematic exploration finds 30% increased unique failures compared to random mutation testing. Systematic exploration finds the same number of failures after 10000 trials for both configurations. The number of tests needed to find the maximum amount of bugs is furthermore comparable for all results. Interestingly, random mutation converges within only 500 trials for this benchmark, while our systematic approach converges at 5000 trials with random columns and 1000 trials with column exploration.

5.2 Findings

To assess the performance of our implementation we aim to answer the following questions.

1. What is the difference in numbers of errors found?

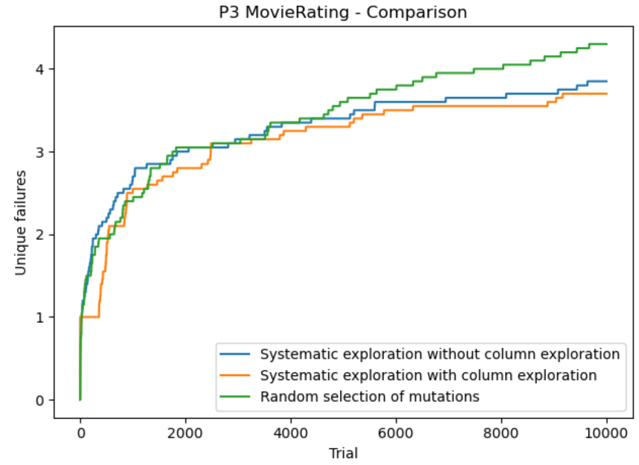

Figure 6: Benchmark P3 with systematic mutation compared to random mutation. The systematic exploration tree depth is 4.

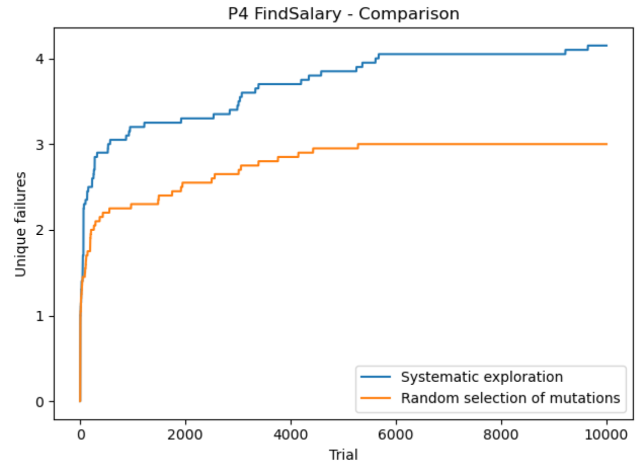

Figure 7: Benchmark P4 with systematic mutation compared to random mutation. The systematic exploration tree depth is 4.

2. What is the difference in number of runs needed?
3. How does the configuration of the exploration influence the performance?

Comparing the benchmarks results leads to the following findings: (1) Out of the five benchmarks evaluated systematic fuzzing finds more unique failures in three benchmarks, P1; P4; P5, performs similar to random fuzzing for benchmarks P2 and is slightly outperformed for benchmark P3. (2) The number of runs before the amount of failures starts to converge is similar between random and systematic mutation testing for two out of five benchmarks. For benchmarks P3 random mutation testing does not converge, while systematic mutation converges at a higher number of trials for benchmarks P1 and P5. Because of these conflicting results no conclusion can be drawn on systematic exploration in general. Future research could verify whether the amount of runs needed is larger when using systematic exploration for test-

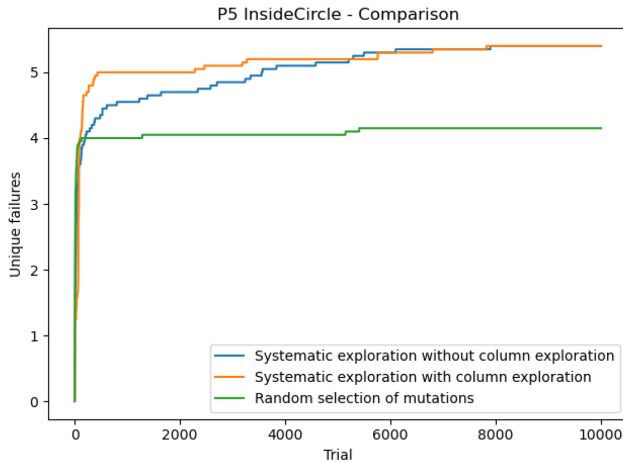

Figure 8: Benchmark P5 with systematic mutation compared to random mutation. The systematic exploration tree depth is 4.

ing most applications. (3) For all benchmarks besides P3 the configuration without column exploration finds equal or less unique failures than with column exploration enabled. Additionally, Figures 2-3 show that a mutation depth between 3 and 6 result in a similar amount of unique failures. A mutation depth of four appears to give the best results.

The results found furthermore answer our third research question: which program properties determine the performance of systematic exploration? For three out of the five benchmarks, P1; P4; P5, systematic exploration finds a higher number of failures than random mutation. These three benchmarks have in common that they have only one delimiter. Our implementation uses a "," as delimiter, as listed in Table 1, for all benchmarks. For the benchmarks that have multiple delimiters this could decrease the usefulness of systematic exploration, since differing delimiters are not mutated. This hypothesis can be verified in future work by applying our implementation to additional applications. Other properties for these three benchmarks are different, such as the the number of columns and the data types per column.

These findings answer our main research question, how does systematic exploration of high-level mutations affect the performance of a fuzz testing framework. Systematic exploration can find a higher number of failures, for benchmark P1 50%. However, systematic exploration requires more trials to converge to a maximum number of failures for two out of five benchmarks.

6 Limitations and Responsible Research

This section offers perspective on possible ethical concerns for this research, as well as encountered limitations.

6.1 Limitations

Our evaluation faced several limitations, which should be kept in mind while considering the results. First, since this study is an extension of the BigFuzz framework it relies heavily upon previous work done by the BigFuzz authors [4]. However, the repository linked to in the paper [19] does not resemble the paper in its entirety. Several components were either missing or non-functional at the time of writing. For instance, only two out of six mutations were implemented on the repository. Thus, for this paper the mutations are not extended from BigFuzz. Instead the mutations are implemented as described in the BigFuzz paper, with the extension of M7 as described in Figure 1. Furthermore the implementation of BigFuzz contained classes specifically made for each benchmark, leading to tailored results. For this reason this study does not compare between the performance of the BigFuzz repository and this study. Additionally, our approach consists of one set of classes designed for all benchmarks instead of one version for each benchmark. An explanation for these problems could be that the public code was not the latest version of the BigFuzz framework.

Second, the evaluation of the research is reliant upon the BigFuzz framework in its ability to remove duplicate failures. The BigFuzz repository code contained a bug causing duplicate unique failures which we have since fixed, however there may be additional bugs such as these. This is a general limitation of evaluating fuzz testing frameworks as noted by Klees et al [18].

Third, out of the twelve benchmarks used in the BigFuzz paper only six error-seeded versions were present in the BigFuzz repository. One of those six benchmarks could also not be used due to missing input specification functionality. This benchmark applied a loop for a number of iterations based on a column in the input file. Because of the missing functionality our implementation can generate large numbers, which resulted in our implementation being practically inapplicable to test the benchmark. Another limitation concerning the benchmarks is that it is unclear exactly how many errors were present in the benchmarks found in the BigFuzz repository. We can therefore not remark on the best possible performance that could be achieved.

6.2 Responsible Research

To ensure this study has been conducted responsibly several measures were taken, listed in this section. (1) Our implementation, seeds and results are freely accessible for verification purposes: https://github.com/LvKvA/SysFuzz. Note that the repository also contains code from our research group from independent studies. This repository can be used to confirm that our results were not fabricated. However, our implementation does use random values at multiple points, for instance for choosing the character or value to be inserted into a column. Consequently, some results may be difficult to reproduce. Our results are best reproduced by using the same configurations with at least twenty iterations. (2) We aim to mention all obtained results in this paper. The evaluation data is thus not trimmed in other means than mentioned in this paper, for instance for benchmarks that were excluded. Therefore the number of unique failures may differ for configurations not mentioned in the paper. All tested configurations are either shown or mentioned in the results section.

The authors report no conflicts of interests. There is no connection with the authors of the BigFuzz paper, or other involved parties. The research was carried out in a research group for The Delft University of Technology, with no grants.

7 Conclusion

Testing DISC applications is becoming increasingly relevant. To extend knowledge on this topic, we extend a state-of-the-art fuzzing framework for DISC applications, called BigFuzz. Our approach explores combinations of high-level mutations by combining seven high-level mutation types with the goal of finding a higher number of failures. To this end our implementation consists of a depth-first traversal of a data tree structure, bounded by a set depth and pruned by removing illogical mutations. The obtained results answer our main research question: 'How does systematic exploration of high-level mutations affect the performance of a fuzz testing framework?'.

The results show that systematic exploration can outperform random testing, depending on both the configuration we apply and the type of benchmark. In three out of five benchmarks, the number of failures found is higher for systematic exploration than by applying mutations randomly without combining mutations. Furthermore, the number of runs required before the amount of failures starts to converge is not increased by testing systematically, for the majority of the benchmarks. Lastly, for the configuration we conclude that exhaustively exploring the mutation types for all columns leads to better results than applying mutations on one random column. The depth bound of the tree seems to have a lesser impact on the number of failures found compared to the type of column exploration.

Based on our results, we hypothesise that the input specification of the application determines the effectiveness of systematic mutation. Further research is needed to show the performance of systematic high-level mutation testing for other fields.

8 Acknowledgement

We would like to thank the members of our research group for their guidance and contributions: Melchior Oudemans, Martijn Smits, Lars Rhijnsburger en Bo van den Berg. The feedback received by anonymous reviewers was also greatly appreciated.

References

[1] Apache Spark™, "Apache Spark™ - Unified Analytics Engine for Big Data," 2005. [Online]. Available: https://spark.apache.org/

[2] "American fuzzy lop." [Online]. Available: https://lcamtuf.coredump.cx/afl/#bugs

[3] "Heartbleed Bug." [Online]. Available: https://heartbleed.com/

[4] Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, and M. Kim, "BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction," *Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*, pp. 722–733, 2020.

[5] Y. Jia and M. Harman, "Higher Order Mutation Testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2009.04.016

[6] A. S. Ghiduk, M. R. Girgis, and M. H. Shehata, "Higher order mutation testing: A Systematic Literature Review," *Computer Science Review*, vol. 25, pp. 29–48, 2017. [Online]. Available: www.elsevier.com/locate/cosrev

[7] R. R. A. Silva, S. S. d. R. Senger de Souza, and P. S. P. Lopes de Souza, "A systematic review on search based mutation testing," *Information and Software Technology*, vol. 81, pp. 19–35, jan 2017.

[8] C. Olston, S. Chopra, and U. Srivastava, "Generating Example Data for Dataflow Programs," 2009.

[9] M. A. Gulzar, S. Mardani, M. Musuvathi, and M. Kim, "White-box testing of big data analytics with complex user-defined functions," *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 290–301, 2019.

[10] K. Li, C. Reichenbach, Y. Smaragdakis, Y. Diao, and C. Csallner, "SEDGE: Symbolic example data generation for dataflow programs," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, 2013, pp. 235–245.

[11] V. J. Manès, H. S. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *arXiv*, pp. 1–21, 2018.

[12] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 511–522, 2013.

[13] S. J. Kim, J. Cho, C. Lee, and T. Shon, "Smart seed selection-based effective black box fuzzing for IIoT protocol," *Journal of Supercomputing*, vol. 76, no. 12, pp. 10 140–10 154, 2020. [Online]. Available: https://doi.org/10.1007/s11227-020-03245-7

[14] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," *Proceedings - International Conference on Software Engineering*, pp. 122–131, 2013.

[15] M. Zalewski, "American Fuzzy Lop." [Online]. Available: https://lcamtuf.coredump.cx/afl/

[16] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing Greybox fuzz testing coverage," in *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, Inc, sep 2018, pp. 475–485.

[17] R. Padhye, C. Lemieux, and K. Sen, "JQF: Coverage-guided property-based testing in Java," in *ISSTA 2019 - Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 398–401.

[18] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the ACM Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, oct 2018, pp. 2123–2138. [Online]. Available: https://dl.acm.org/doi/10.1145/3243734.3243804

[19] Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, and M. Kim, "BigFuzz repository," 2020. [Online]. Available: https://github.com/qianzhanghk/BigFuzz

[20] J. A. do Prado Lima and S. R. Vergilio, "A systematic mapping study on higher order mutation testing," *Journal of Systems and Software*, vol. 154, pp. 92–109, aug 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2019.04.031

[21] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the cost of mutation testing with second-order mutants," *Software Testing, Verification and Reliability*, vol. 19, no. 2, pp. 111–131, jun 2009. [Online]. Available: http://doi.wiley.com/10.1002/stvr.392

[22] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, jun 1972. [Online]. Available: http://epubs.siam.org/doi/10.1137/0201010