Guiding Big Data Fuzz Testing with (Boosted) Coverage-Based Input Selection

Bo van den Berg , Burcu Özkan
TU Delft

Abstract

Big data applications are becoming increasingly popular. The importance of testing these applications increases with it. A recently proposed work called BigFuzz applies automated testing. The big data fuzzing tool shows very promising results.
The aim of this research is to inspect how coverage guidance affects the performance of big data fuzzing. The current coverage usage is first described, then an extension is proposed, which is compared to the original.
This work extends the BigFuzz tool with branch coverage guidance. The existing black-box fuzzer is substituted for a grey-box fuzzer, which is then extended to a boosted grey-box fuzzer.
The two extensions both allow branch discovery. Boosted grey-box fuzzing shows to be the most efficient branch exploration mechanic. Furthermore, both extensions outperform the original tool regarding error detection.

Keywords

fuzz testing, software testing, test generation, branch coverage, big data analysis, DISC systems

1 Introduction

In the last 10 years, big data applications have become more and more popular, and this popularity increase does not seem to halt anytime soon either. The Big Data industry has been estimated at a worth of $56 billion USD in 2020 and its value is expected to be doubled by 2027 [12].
These big data applications often use Data-Intensive Scalable Computing (DISC) systems to handle such large data. A few examples of such systems are Apache Spark [20], Apache Hadoop [17] and Google's MapReduce [5].
At the moment, the common testing practise for such applications is to run it locally on randomly sampled inputs [23]. However, due to the magnitude of such big applications, it is rather difficult (and time-consuming) to find all bugs using this manual testing method. Rare and buggy corner cases are regularly only encountered when these data-intensive applications are used in practise, which could lead to crashes or corrupted output [16].
Another proposed testing method is fuzz testing [22][10]. Fuzz testing or fuzzing is an automated software testing technique. The essence of fuzzing is to automatically and repeatedly generate input data that may be malformed, and see if it breaks the program or creates corrupted data. The simplest fuzzing technique is random fuzzing [1], which is not applicable to huge applications because they require a lot of start-up latency. Another variant of fuzz testing is mutation-based fuzzing [25], which is more efficient at finding valid inputs as it creates new inputs based on mutating known valid inputs.
A specialization of the above is Coverage-Guided mutation-based Fuzzing (CGF) [14][9][18][21]. This variant is recently emerging as a majorly successful test generation technique for large software systems [10]. Coverage-Guided Fuzzing (CGF) systems like AFL [21] have proven to be highly successful in finding bugs by aiming to maximize code coverage. However, applying conventional coverage-based fuzzing to big data applications comes with some challenges because "1) DISC systems have a long latency; 2) Conventional branch coverage is unlikely to scale to DISC applications because most binary code comes from the DISC framework itself; and 3) random bit or byte level mutations hardly generate meaningful data, and often fail to reveal real-world application bugs" [24].
Recently, the open-source tool BigFuzz [24] is proposed, which is a fuzzing tool for big data applications. It first abstracts the DISC framework using UDF specifications and applies automated transformations to create an equal but smaller DISC application that (in contrast to the DISC application itself) is suitable for fast test generation. The tool outperforms the state-of-the-art big data testing tool significantly in many ways such as fuzzing time, code coverage, error detection and applicability. BigFuzz is a very promising tool, but it is still at an early stage and there is definitely some room left for improvement.
This work is focused on extending the BigFuzz tool by guiding the fuzzer with coverage information. The tool itself lays the framework for coverage information, but does not utilize the information to generate new inputs. The main aim of this research is to inspect how input selection based on coverage information affects the performance of fuzz testing big data applications. To effectively do so, it is useful to answer three sub-questions: 1) How is the coverage information currently

used in big data fuzzing?; 2) How can big data input selection be improved based on coverage information?; and 3) How does the extended fuzzer compare to the current fuzzer? This comparison is done with regards to coverage, number of tests, and detected errors.
The main contributions made by this research are:

- Grey-box fuzzing - Inputs that find new branch coverage are saved and used as inputs for further mutations.
- Boosted grey-box fuzzing - Inputs that found "unusual" paths are exercised more often than other frequent paths.

On the BigFuzz benchmarks, grey-box fuzzing finds equal (sometimes even more) errors as the black-box implementation. Boosted grey-box fuzzing consistently slightly outperforms black-box error detection on all BigFuzz benchmarks. Branch discovery speed is similar on all three methods.

However, these benchmarks are very small and only contain two branches to discover at most. In order to allow proper evaluation, another benchmark containing 40 branches is introduced.
Boosted grey-box fuzzing is shown to be the most efficient at finding high branch coverage. After 1500 iterations, it has discovered double the amount branches compared to traditional greybox/coverage-guided fuzzing. The original black-box BigFuzz implementation quickly stops discovering new branches as expected.
This paper is organized as follows. Section 2 provides more background information on the relevant subjects. Section 3 contains a detailed description of the approach used for the implementation. Section 4 discusses the extension's results and how these results have been gathered. Section 5 shows how to replicate these results and touches on the ethical aspects. Section 6 provides a summary of related works, and Section 7 includes a conclusion and touches on interesting subjects for future work.
The complete extended implementation can be found on our GitHub repository [3].

2 Background

Apache Spark. Apache Spark [20] is one of the most commonly used DISC frameworks. In Spark, datasets are loaded as Resilient Distributed Datasets (RDDs) [19]. These RDDs can then be transformed in parallel using dataflow operations (e.g. filter, join, map). These operations are higher-order functions which take a User-Defined Function (UDF) as argument. RDDs are evaluated in a lazy manner, meaning that the computation is postponed until its value is actually needed. A series of RDD transformations is rewritten as a Directed Acyclic Graph (DAG) in which the vertices represent the operations. Next, each stage is executed in a topological order by Spark's scheduler. This process is also called the pipeline.
Random fuzzing Fuzz testing or fuzzing is an automated software testing technique. The essence of fuzzing is to automatically and repeatedly generate input data that may be malformed, and see if it breaks the program or creates corrupted data. The simplest fuzzing technique is random fuzzing, in which inputs are generated at random [1]. The random fuzzing method is not at all sufficient at finding valid inputs. This insufficiency is usually compensated by the fact that thousands of random inputs can be generated and used as inputs each second. However, DISC applications sometimes spend 98% of the total run-time just setting up the environment [24]. The high start-up latency defeats the purpose of random fuzzing.
Mutation-based black-box fuzzing A second variant of fuzz testing is mutation-based fuzzing [25], in which new inputs are generated based on (small) mutations to known valid inputs. Most programs only allow specific inputs, and mutation-based fuzzing is efficient at finding more valid inputs because it uses valid inputs to the base mutation on. Regular mutation-based fuzzing is considered a black-box testing method [13], because it does not require any knowledge about the program internal structure.
Coverage-Guided Fuzzing. A specialization of the above is coverage-guided mutation-based fuzzing [14][9][18][21]. Inputs to mutate on are chosen based on coverage information, rather than at random. CFG tools like [21] have shown great success at discovering memory-corruption bugs, which are introduced by incorrect handling of unexpected inputs. However, applying conventional coverage-based fuzzing to big data applications comes with some challenges because "1) DISC systems have a long latency: naive fuzzing would spend 98% of the time setting up a test environment; 2) Conventional branch coverage is unlikely to scale to DISC applications because most binary code comes from the DISC framework itself; and 3) random bit or byte level mutations hardly generate meaningful data, and often fail to reveal real-world application bugs" [24].
The CGF algorithm works as follows. it utilizes a set S of inputs, on which future mutations will be performed. S is first initialized with a set of user-provided initial inputs, declared in a configuration file. The baseline CGF method is to cycles through each element from S and to generate a mutation M by applying at least one mutation operation on the input. The 6 used mutations used are described in more detail in chapter 3.3 from the BigFuzz paper [24], and are summarized in Appendix A. The application is then executed with M as the next test input. "The key to the CGF algorithm is that it instruments the test program with dynamic feedback in the form of code coverage for each run" [14]. All valid inputs that explore new code branches (or a new combination of branches) are saved in the Coverage set C for future mutation. The new-found branches are also added to the total coverage. After a cycle is completed, S will be filled with all coverage-finding inputs from C and the next cycle starts. The program stops when the stop condition is met. This condition can either be a maximum number of tests or a maximum duration.

3 Approach

The key idea of the performed research is to guide fuzz testing of big data applications with coverage-guided mutation input selection. As stated before, the BigFuzz tool lays the foundation for CGF by tracking coverage information. Initially, mutations are always performed on the initially provided input file(s). It can be classified as black-box testing, because it does not require any information on the program's internals.

This chapter is split into three subsections. The first shows what the original program was like. The second subsection explains the implementation of the baseline input selection method. The thirds explains how the input selection is extended with boosted grey-box fuzzing.

3.1 Black-box Fuzzing

The general process used in all fuzzing methods is described as a fuzzing loop: First, an input is selected from the set of interesting inputs. This seed input is then mutated with a random mutation. The program is then run with the mutated input. If the run results in an interesting discovery (for CGF: finds new coverage), it is added to the set of interesting inputs. Otherwise, it is discarded.
The mutations used for this research are direct copies of the mutations from chapter 3.3 of the BigFuzz paper [24], which are summarized in Appendix A.
The given implementation does not act on its feedback. It does not use any knowledge about the application's internal framework and does not have any intention of learning about it either, thus it can be considered as black box testing. Only the input and its produced output are known. This black-box fuzz loop is visualized in figure 1.

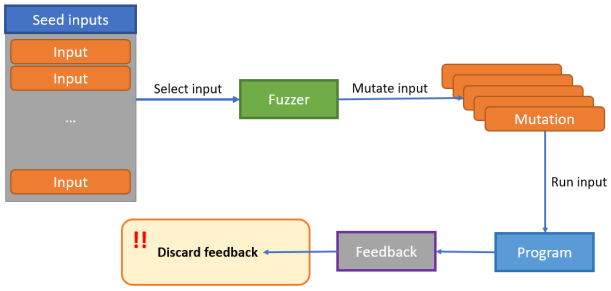

Figure 1: A visual representation of the black-box fuzz loop

3.2 Grey-box Fuzzing

Black-box fuzzing performs some exploration, but does not utilize coverage feedback/information. It suffices for small benchmarks, but hardly explores bigger benchmarks. The aim is to use coverage information in order to increase the seed inputs pool with inputs that found previously unexplored coverage.
The first needed addition was to make the input files and mutations more manageable. All created mutations are saved in a sub-directory *all_inputs* and files referencing the inputs are saved to their relevant sub-directories. The most important sub-directory for this research is *new_coverage_inputs*, which contains only the inputs that found a new combination of discovered branches.
The next step is to transform the program from a regular mutation-based fuzzer into a coverage-guided mutation-based fuzzer. The needed extension is to make the program run through each coverage-discovering input in cycles. At the beginning of each cycle, all known inputs from *new_coverage_inputs* are stored in a list. The program then iterates through this list and uses each of those files as input to generate a mutation from. If the mutation discovers new branches, it is added to the seed inputs. This cycle loop is used as the baseline method used for input selection.
The program can now be considered as a grey-box fuzzer: It roughly "discovers" what the original application looked like by exploring it based on inputs that found new paths. A visualization is provided in Figure 2, in which the differences with black-box fuzzing loop are indicated by the selection with exclamation points.

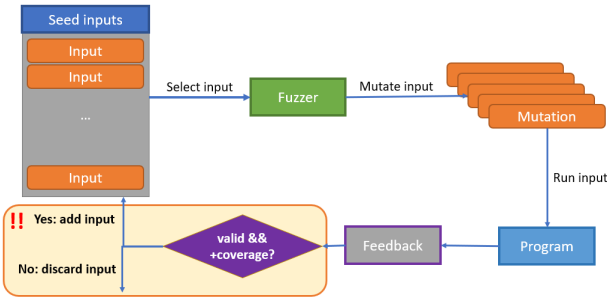

Figure 2: A visual representation of the grey-box fuzz loop

3.3 Boosted Grey-box Fuzzing

The baseline input selection method allows for branch exploration, but the mutations often result in finding the same few branches again and again. To further increase the rate at which new paths are discovered, it makes sense to select inputs that found "unusual" paths more often. This idea is referred to as boosted grey-box fuzzing.
The idea is to redistribute fuzzing time among seed inputs, by pulling some executions from the most-often explored paths and add them to the least-often explored paths instead. The new distribution allows more exploration around the newer and more refined paths.
We extend the baseline method to include favored input selection. First, a counter is added which tracks how often each branch combination is explored. The favored selection method utilizes the inverted value to determine the chance at which the input is selected again.
Again, a visual representation is provided, found in Figure 3.

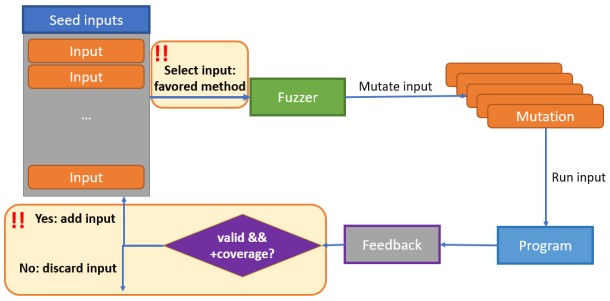

Figure 3: A visual representation of the boosted grey-box fuzz loop

ID	Subject Program	Reason
P1	WordCount	Accepts any String (always valid)
P3	ExternalFunction	Accepts any String (always valid)
P8	StringSelf	Accepts any String (always valid)
P9	NumberSeries	Results in an infinite loop
P11	IncomeAggregation	Does not contain any failures

Table 1: Omitted benchmarks and reasoning

4 Evaluation

This section presents an empirical evaluation of the extensions' effectiveness, compared to the original application.
The possible program arguments are described in more detail in Section 5.1.
This research's evaluation is set up as follows. We run 10 iterations using 2500 trials as the stopping condition. The maximum duration is set to null, meaning that each trial should execute (which should take about 1 minute per iteration). Furthermore, we run the program with StackedMutation as mutation method, and maximum mutation stack set to 0. Setting the maximum mutation stack to 0 disables StackedMutation, resulting in mutation application equal to BigFuzz.
This configuration is used to run the 3 different selection methods covered in section 3: black-box, grey-box, and boosted grey-box selection.
The extended application is first compared to the original using its existing 12 benchmarks, to ensure that it does not lead to decreased error detection or coverage. Comparison is be done with regards to coverage, number of tests, and detected errors. Because the original BigFuzz benchmarks are insufficiently small, a second sub-section is added in the configurations are run on a newly created benchmark, which contains 40 branches.

4.1 BigFuzz Benchmarks

The existing BigFuzz benchmarks are used as test classes and test methods. A detailed description of these benchmarks is provided in Section 4 and shown in Table 4 of The BigFuzz paper [24]. A copy can be found in Appendix B, including a column indicating the used input. It is important to know that these benchmarks are based on the results of a research analyzing frequent DISC errors.
Five out of twelve benchmarks are omitted from the unique failure results for various reasons, as displayed in Table 1. The unique failure results of all other benchmarks are found in Appendix C.
Regarding error detection, the results show that boosted- and traditional grey-box fuzzing perform at least as good as black-box testing.
Regarding code coverage, BigFuzz benchmarks contain at most 2 branches. The 3 benchmarks with multiple branches are FindSalary (P4), InsideCircle (P7), and IncomeAggregation (P11). All other benchmarks have only one branch, which does not allow exploration.
The branch discovery speed of each selection method on these 3 benchmarks has been combined, and the calculated mean is depicted in Figure 4. The two extensions are a little slower, but still discover all branches within 80 trials.

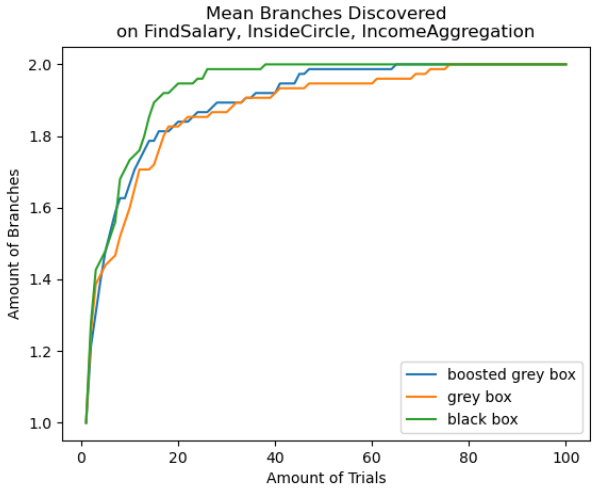

Figure 4: Average branch discovery on the 3 BigFuzz benchmarks containing multiple branches

4.2 Additional Benchmark for Coverage

Our results show that the extended application produces equal, if not improved results on the original benchmarks. However, the small branch size of 2 does not allow much coverage exploration.
Therefore, the extra benchmark "BranchMark" has been created, which contains 40 branches to explore. The new benchmark does branch exploration based on the amount of columns existing in the first line of the input file, with a limit of 40 (e.g. inputs "1" and "1, 2" lead to different branches). The benchmark allows 3 out of 6 mutations to possibly discover new branches:

- Data Format Mutation (M3) - Mutates a column-separating delimiter mentioned in the schema (e.g. replacing delimiter "," to " ").
- Data Column Mutation (M4) - Inserts one or several characters (e.g. could add a delimiter character).
- Null Data Mutation (M5) - Mutates the input row by removing one or more columns.

BranchMark does not contain any injected errors. It is created with the single purpose of comparing branch discovery. Creating invalid inputs only slows down the process.
The initial input file contains only the line "1, 2". The exact used program arguments are "BranchMarkDriver testBranchMark StackedMutation 2500 0 2 conf_branchmark".
To provide some more detail to the impact of favored input selection compared to the baseline input selection, a half boosted grey-box selection method has been evaluated. This algorithm selects baseline input selection half of the time, while utilizing the favored method for the other half.
This benchmark is evaluated with 25 iterations (of 2500 trials). The average results for each selection method are shown in Figure 5.
We immediately see that the black-box implementation stops discovering new branches at an early stage. This is expected, as it can only generate 1 extra column using M4, or find less

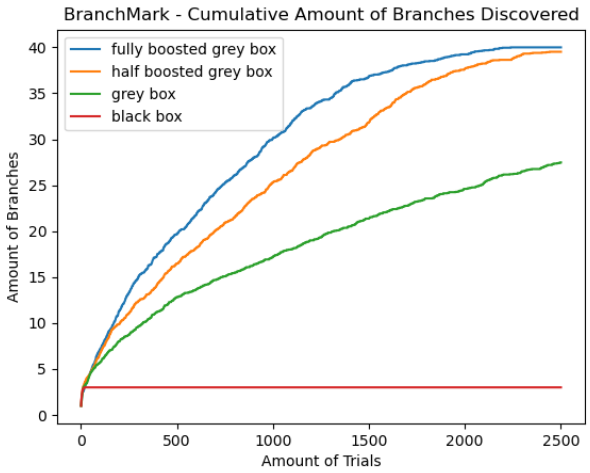

Figure 5: Coverage improvement of boosted greybox fuzzing over traditional greybox- and blackbox fuzzing

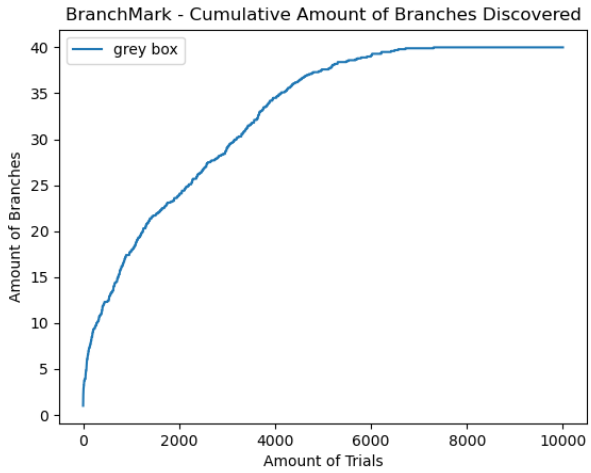

Figure 6: BranchMark coverage of grey-box fuzzing

columns using M3 or M5. Each black-box fuzzing iteration finds exactly 3 branches.

The results show that fully boosted grey-box fuzzing is the most efficient algorithm for finding new branches. It discovers almost double the amount of branches as traditional grey-box fuzzing. These results are explained by its eager exploration. The newly produced (and thus less-frequently discovered) branches are frequently selected by the favored selection method, which leads to quick exploration. Each of the 25 iterations discover all 40 branches. The last branch is found after 2000 trials on average.

Even though traditional grey-box testing is not as efficient as its boosted alternative, the results show that it still steadily exploring new branches until the stopping condition is reached. It finds 27.48 branches on average. None of the iterations reached 40. A smaller side experiment (see Figure 6) shows that it takes approximately 7500 trials for grey-box fuzzing to find all 40 branches.

The discoveries made by half boosted grey-box fuzzing lie perfectly in between the two algorithms for the most part. In the end, we see that it starts to discover the last few branches even quicker than fully boosted grey-box fuzzing. At that point, most branches are discovered by the boosted part. On average, this variant finds 39.52 branches in 2500 trials. Three iterations end up at 37 branches and three others result in 39. The others find all branches.

Figure 7 shows only the first 30 trials of this experiment. We can see that the coverage results are a lot more similar during these first few trials. However, black-box fuzzing quickly shows to be the least successful.

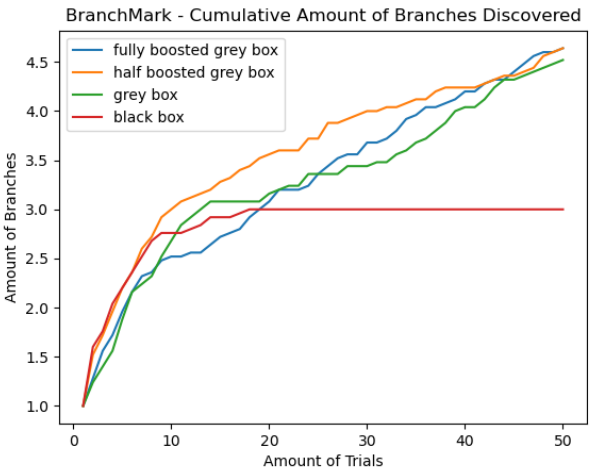

Figure 7: Only the first 50 trials of figure 5's experiment

5 Responsible Research

The purpose of this section is to guide the reader on reproducibility, and to reflect on any ethical aspects regarding the research.

5.1 Reproducibility

The complete extended version of this implementation is uploaded to this GitHub page [3]. Any drawn conclusions are based on careful inspection of this repository.

In order to properly recreate Section 4's results, checkout the CovEvaluation branch and execute the edu.tud.cs.jqf.bigfuzzplus.BigFuzzPlusDriver class. For example by using the run configurations: `FindSalaryDriver testFindSalary StackedMutation 1000 0 2`.

The program arguments can be configured as follows:

0 Test class - Use any benchmark defined in the *Benchmarks* folder (accessed by module *bigfuzz-benchmarks*)

1 Test method - Use any method from the test Class

2 Mutation method - Use StackedMutation as default

3 Max trials - Set to how many trials should be performed

4 Stacked mutation method - Use 0 as default

5 Max mutation stack - Use 2 as default

Furthermore, any of the fields in BigFuzzPlusDriver class can be enabled (e.g. set to true) in order to print the relevant run information to the terminal. Use either Black Box, Grey Box, Half Boosted Grey Box, Fully Boosted Grey Box testing, or a combination as the selection method. The NUMBER_OF_ITERATIONS and max duration variables can be adjusted as well to fit your preferences.
If anything regarding the reproducibility remains unclear, my contact information can be found in the header of this paper.

5.2 Ethical Aspects

First of, this research heavily relies on the BigFuzz tool [24], which in turn extends the JQF [14] implementation and targets Apache Spark [20]. Some errors may have been introduced in deeper layers, and end up in this research as well.
Secondly, automated testing relies on randomness. It can not guarantee whether each bug will be detected, or whether bugs can be found within a certain amount of trials. Our GitHub repository [3] can be used to produce *similar* results. For proper approximation, we recommend using a significant amount of iterations, at least 20.
Thirdly, while fuzzing tools are useful for detecting bugs and corner cases, it should be used merely as an assisting tool. Manual testing or other tools might still be needed. Trivial edge cases (such as swapping 0 for 1) might go undetected by the tool's randomness, but are simple for humans.
And lastly, I would like to mention that BigFuzz only claims to propose a "novel coverage-guided fuzz testing tool for big data analytics" [24] in the paper's abstract. The groundwork did not actually act upon the coverage information yet. Due to the wording, their statement could be misinterpreted as if the BigFuzz tool already contained coverage-guidance, but actually it is part of this research's extension.

6 Related Work

Currently, fuzz testing can invoke thousands of random test inputs each seconds. However, fuzzing has not always been as efficient as it is now. In this chapter, the origin and evolution of fuzz testing DISC applications are discussed in detail.
Fuzz Testing. Fuzz(y) testing uses randomized testing to test the targeted program's input handling. The essence is to automatically and repeatedly generate (malformed) input data, which are run against the systems to invoke crashes, hangs or corrupted data. The data is not necessarily random, but inputs which are meaningless to the software are more likely to crash it (e.g. providing a random String instead when a HTTP request is expected).
The first fuzz-like experiment is Macintosh's the Monkey, which is considered random fuzzing. "Macintosh seemed to be operated by an incredibly fast, somewhat angry monkey, banging away at the mouse and keyboard, generating clicks and drags at random positions with wild abandon" [6]. The implementation is based on the infinite monkey theorem, which states that a monkey hitting keys at random for an infinite amount of time will almost surely type a given text.
Before public networks and the internet become more available, "attackers" did not exist yet. Software security and reliability were often neglected. Syntax testing and fuzz testing were only really introduced at 1990 [2][11].
The interactive Fuzzing Book [22] provides excellent and detailed information about most fuzzing methods and attributes. This survey on Fuzz Testing [10] summarizes prior fuzzing works, and the paper Evaluating Fuzz Testing [7] introduces guidelines for fuzz evaluation to help preserving coherency in the field.
Coverage-Guided Fuzzing. AFL [21] is the most-often used CGF tool worldwide. It adds inputs that found previously undiscovered coverage information to the seed inputs pool. AFL is responsible for detecting a ton of errors. In 2019, FairFuzz [9] introduces an extension to the traditional CGF algorithm. It identifies rarely executed branches and exercises them more often. Custom mutations are used which gravitate toward exploration around the rare branches, resulting in higher and quicker coverage discovery.
The open-source JQF platform [14] instruments a program's bytecode with coverage information, and implements a fuzzing loop. Its Guidance interface can be extended to guide custom fuzzing algorithms.
Other Fuzzing Approaches. Another approach is to guide mutation and selection by utilizing symbolic execution in order to find new program paths [4]. For example, Zest [15] creates inputs by applying small bit-level mutations to valid inputs. Yet a different approach is to only generate valid inputs for a program. This is implemented by manually declaring a set of legal inputs and boundaries. For example, Saffron [8] accepts a set of user-defined grammar. If a program failure is encountered during fuzzing, then the tool adjusts the grammar accordingly, evolving the grammar specifications.

7 Conclusions and Future Work

The main aim of this research is to inspect how input selection based on coverage information affects the performance of fuzz testing big data applications. At the time of writing, BigFuzz is the only tool allowing to fuzz big data applications, so this research is only aimed towards their implementation.
The original BigFuzz application contained some design choices regarding coverage-guidance. It keeps track of the run's total coverage, and any inputs that explored new branch coverage are "saved" (i.e. renamed to include +cov in the file name). However, this information was not further used. All generated inputs are discarded after exercising the program.
In order to speed up and expand coverage exploration, this work covers the two main contributions:

- Grey-box fuzzing - The default input selection is changed to cycle through all inputs which discovered new branches, rather than only using the user-defined initial files.
- Boosted Grey-box Fuzzing - A favored input selection method is implemented which favors selecting inputs that exercise less frequent paths.

Both extensions outperform black-box fuzzing consistently in finding unique failures. Regarding coverage exploration, black-box fuzzing shows to be insufficient for benchmarks

containing a vast amount of branches. Grey-box fuzzing does perform sufficient exploration, but the boosted variant clearly shows to be the most efficient.

Acknowledgements

We thank the creators of BigFuzz for allowing us to use their GitHub repository and for their great research. They have shown to be a major contribution to the subject of fuzz testing DISC applications.
Furthermore, this research has been performed as a research project for the bachelor Computer Science and Engineering at TU Delft. Together with 4 other students, we have created five different extensions to improve the BigFuzz tool and each of these extensions is included in [our shared GitHub repository [3]. The other researched extensions are focused on answering the following questions: 1) How can we provide users to enter input specifications and implement mutations in a generic way for all kinds of tabular data?; 2) How can we provide users to enter input specifications and implement mutations in a generic way for all kinds of JSON typed data?; 3) The current fuzzing algorithm applies one mutation on the input at a time. How does applying multiple high-level mutations at a time affect the performance?; and 4) How does systematic exploration of high-level mutation types affect the performance of the fuzz testing framework?
And most of all, we want to thank our supervisor Burcu Özkan for her great advice and support during the project.

References

[1] C. Miller A. Takanen, J.D. Demott. Fuzzing for software security testing and quality assurance. Artech House, 2008.

[2] B. Beizer. Software testing techniques. 1990.

[3] B. Berg, L. Rhijnsburger, L. Koetsveld van Ankeren, M. Oudemans, and M. Smits. site: github.com/bovdberg/bigfuzzcovguidance, Jun 2021.

[4] S.K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. 2015.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. 2008.

[6] A. Hertzfeld. Macintosh monkey lives, Oct 1983.

[7] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. 2018.

[8] Xuan-Bach D. Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. Saffron: Adaptive grammar-based fuzzing for worst-case analysis. *SIGSOFT Softw. Eng. Notes*, 44(4):14, December 2019.

[9] C. Lemieux and K. Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. 2018.

[10] V.J.M. Manes, H. Han, C. Han, S.K. cha, M. Egele, E.J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019. cited By 32.

[11] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[12] Kimberly Mlitz. Global big data market size 2011-2027, Jan 2021.

[13] Srinivas Nidhra and Jagruthi Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[14] Rohan Padhye, Caroline Lemieux, and Koushik 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.

[15] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with ZEST. In *ISSTA 2019 - Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[16] N. Pillay, Vikash Jugoo, and Mangosuthu Technikon. An analysis of the errors made by novice programmers in a first course in procedural programming in java. 01 2006.

[17] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. 2010.

[18] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See. Deephunter: A coverage-guided fuzz testing framework for deep neural networks. 2019.

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. 2012.

[20] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. Spark: Cluser computing with working sets. 2010.

[21] Zalewski. American fuzzy lop, 2013.

[22] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. The fuzzing book. Saarland University, 2019. Retrieved 2019-09-09 16:42:54+02:00.

[23] L. Zhang, J. Liang, L. Liu, Z. Jiang, and J. Liu. Improvement of the sample mutation strategy based on fuzzing framework peach. 2018.

[24] Q. Zhang, J. Wang, M.A. Gulzar, R. Padhye, and M. Kim. Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. 2020.

[25] Z. Zhang, Q.-Y. Wen, and W. Tang. An efficient mutation-based fuzz testing approach for detecting flaws of network protocol. 2012.

A BigFuzz's Mutation Table

Mutations, as found in section 3.3 (page 6) of the BigFuzz paper [24]:

- **Data Distribution Mutation (M1)** mutates a record to be either valid or invalid in terms of the allowed range based on the data distribution given in the schema (e.g., an integer value 10, corresponding to the range integer[0-30] mutated to 25 or -1).
- **Data Type Mutation (M2)** modifies the data type of a selected column, while keeping the same value (e.g., 20 corresponding to integer[0-30] can be mutated to 20.0, leading to NumberFormatException in line 5 Figure 2a).
- **Data Format Mutation (M3)** mutates a column-separating delimiter mentioned in the schema (e.g., replacing delimiter "," to " ").
- **Data Column Mutation (M4)** inserts one or several characters (e.g., replicating ArrayIndexOutOfBoundsException in StackOverflow post No.45962453 [15] when a random ' ' is inserted to data that is "Ctrl+A" separated).
- **Null Data Mutation (M5)** mutates the input row by removing one or more columns (e.g., replicating NullPointerException in Stack Overflow post No.36015704 [7] by accessing positions that do not exist).
- **Empty Data Mutation (M6)** mutates a random column to an empty string, leading to StringIndexOutOfBoundException caused by incorrect string operations.

B BigFuzz Benchmarks

The 12 benchmarks found in table 4 (page 7) of the BigFuzz paper [24]:

ID	Subject Program	Output	Initial Input Used
P1	WordCount	Find the frequency of words	"test"
P2	CommuteType	People count using each form of transport for daily commute	file 1: "1,10000,10000,1,15", file 2: "91000,A"
P3	ExternalFunction	Find the frequency of words	"test"
P4	FindSalary	Total income of individuals earning $\leq$ \$300 weekly	"1000"
P5	StudentGrade	List of classes with more than 5 failing students	"CS186:1"
P6	MovieRating	Total number of movies with rating ≥ 4	"ASTAR:_2,_3"
P7	InsideCircle	Check whether the point (x, y) is in a circle	"5,3,4,6"
P8	StringSelf	String Mapping	"test"
P9	NumberSeries	Find the numbers whose 3n+1 series' length is 25	"123,234"
P10	AgeAnalysis	Total number of people with different age ranges	"90001,28,10990"
P11	IncomeAggregation	Average Income per age range in a district	"90001,28,10990"
P12	PropertyInvestment	The frequency of each loan type within a metropolitan area	"a0,a1,2.0,15,0.01,a5,2"

C Unique Failures on BigFuzz Benchmarks

Amount of unique failures detected by running the BigFuzz Benchmarks (see Appendix B) against 3 selection methods: boosted grey-box, traditional grey-box, and black-box.

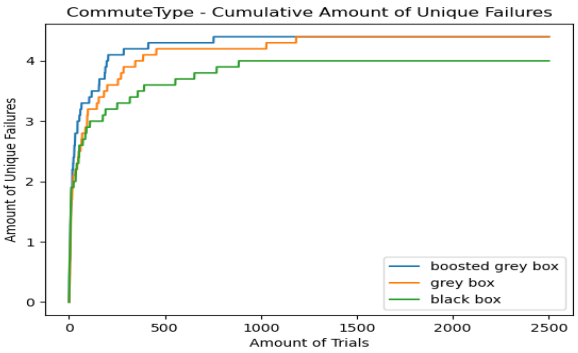

Figure 8: P2 Commute Type Results

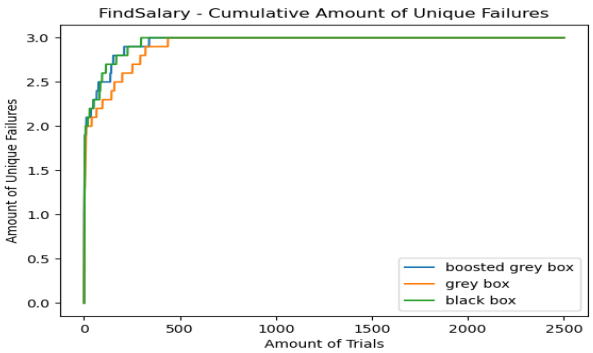

Figure 9: P4 Find Salary Results

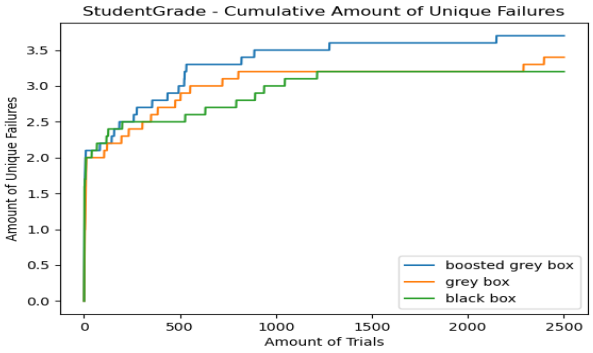

Figure 10: P5 Student Grade Results

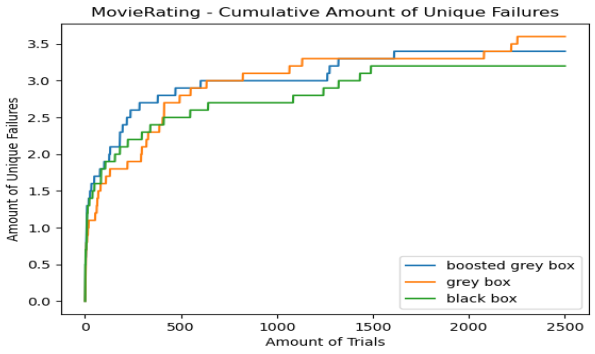

Figure 11: P6 Movie Rating Results

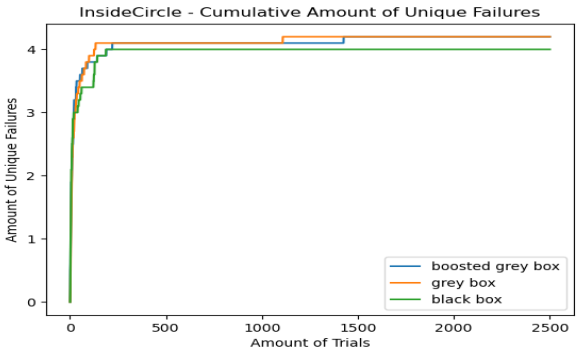

Figure 12: P7 Inside Circle Results

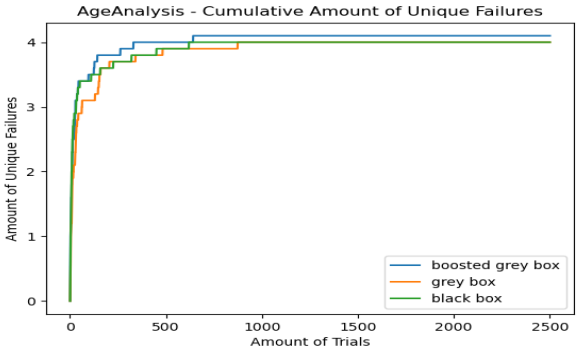

Figure 13: P10 Age Analysis Results

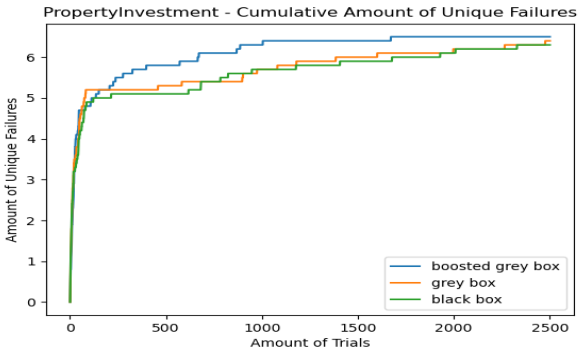

Figure 14: P12 Property Investment Results