TabFuzz: High-level mutations for tabular data

Martijn Smits,

Computer Science and Technology,
TU Delft

June 25, 2021

Abstract

Big Data is an expanding industry, yet exhaustive and automated testing of Big Data applications is still in its early stages. In the last few years, testing framework for Big Data applications have started appearing. BigFuzz is a program that uses fuzz testing for Big Data applications. Fuzz testing means generating random, potentially invalid or erroneous, inputs in attempt to find exceptions. This paper introduces TabFuzz, a tool that improves and extends the BigFuzz solution. TabFuzz reproduces the BigFuzz implementation and extends on it, by improving the generation of random input files. TabFuzz can generate a valid input file based on an input specification. It then mutates this file using high-level mutations. These mutations generate new test inputs that mimic real-world problems. This is an improvement over bit or byte level mutations. These mutations are supposed to mimic real-world problem, which is an improvement over random bit or byte level mutations. Most fuzzing programs start from a user-defined initial input file, called a seed file. TabFuzz offers the possibility to generate such a file. This research shows that these generated files are just as effective as starting from a seed file.

1 Introduction

Big Data is currently a 56 billion dollar industry and is expected to grow to almost double that amount by the year 2027 [1]. Despite the fact that Big Data applications are becoming increasingly more popular, exhaustive and automated testing for these applications is still in early development. BigTest [2] is one of the first frameworks to address this problem. It is a white-box testing framework that takes an Data-Intensive Scalable Computing (DISC) program as input and automatically generates data for testing. However, BigTest is limited to dataflow operators where execution is supported, this means that developing a robust test generation tool for DISC applications is still an unanswered problem [3].

Fuzzing could be the solution to this problem. Fuzzing means automatically providing unexpected input to software and monitoring for exceptions [4]. This technique is considered one of the most effective methods to test the security of software [5]. It has been deployed by some of the largest companies in the world, such as Adobe, Cisco, Google and Microsoft [6].

Recently Zhang et al. released a paper about applying fuzz testing on DISC systems and they released a framework called BigFuzz [3]. However, fuzzing cannot be directly applied to big data systems. The authors of BigFuzz describe the following three problem: (1) the long latency of DISC systems, (2) conventional branch coverage is unlikely to scale to DISC applications because most binary code comes from the framework implementation; and (3) random bit or byte level mutations can hardly generate meaningful data. In their paper they present a solution for these problems.

The goal of this research is to address the following problem: random bit or byte level mutations can hardly generate meaningful data. In order to test DISC applications, it is important that all branches of the code get tested. In the case of bit or byte level mutations, there is a big chance that the structure of the data is affected. This would most likely throw exceptions early on in the test progress in nearly every iteration. This means that many branches are never tested. This paper specifically focuses on designing and implementing high-level mutations for tabular data and assessing the effectiveness of these mutations.

This paper answers the following main question: How can we provide users to enter input specifications and implement mutations in a generic way for all kinds of tabular data? This question has been divided in the following four subquestions: (1) What are the possible ways of programmers specifying input?, (2) How can random inputs be generated based on the specification provided by the user?, (3) How can these inputs be mutated based on the specified structure, such that these mutations mimic real-world errors of big data applications?; and (4) How does the proposed framework perform compared to BigFuzz in terms of effectiveness?

The paper is structured in the following way. Section 2 provides a short background on some of the subjects. Section 3 explains the methodology in detail. Section 4 presents the contribution to this project. Section 5 discusses the evaluation and results of the experiments. Section 6 discusses the ethical side of this project. The conclusion can be found in section 7.

2 Background

2.1 Fuzz Testing

Fuzzing is a tool for finding bugs using randomly generated invalid or erroneous inputs [7]. Fuzzing mostly falls within black box and gray box testing, which, respectively, means that there is no or limited information available about the inner workings of the program [4]. Automated testing, such as fuzzing, is really convenient for black and gray box testing, because it allows the programmer to easily test the program on large amounts of different inputs. Fuzz testing initially started off as a technique to find security-related bugs, but it is becoming more common to use it for non-security programs [6]. What is important to note is that fuzz testing is not used to check the validity of outputs, but rather focuses on finding unexpected behaviour and exceptions within a program. A drawback of fuzzing is, however, that it there is no guarantee that fuzzing finds all bugs, no matter how long it runs [5].

2.2 BigFuzz

BigFuzz [3] is a program that applies fuzz testing to DISC applications. According to the authors of BigFuzz, setting up a test environment takes 98% of the time, because of the high latency of DISC applications. BigFuzz can transform a DISC application into abstracted Java program in order to solve the latency issues. BigFuzz then starts applying fuzz testing on the program using the JQF [8] framework, a framework used for fuzz testing on Java applications. However, instead of applying random bit and byte level mutations, the BigFuzz framework applies high-level mutations in order to create more meaningful test data. BigFuzz then returns the files that either cause an exception or discover new branches. This action is performed for a user-defined number of trials.

3 Methodology

The contribution of this research can be divided into three big components: (1) the specification of test input, (2) the generation of valid input files and (3) the mutations on valid input files. The methodology for each of these components is described in this section.

3.1 Input Specification

In order to generate valid input files and perform mutations on these input files, it is essential to know what makes the data valid or invalid. The input specification has two requirements. The first requirement is that it collects all properties that are important for the program. The second requirement is that it is user-friendly.

The properties were derived using the following two techniques. The first one was inspecting the input specifications of the benchmarks provided by the BigFuzz authors. The second one is by adding properties that either improve the solution or solve unanswered problem that came up during the implementation of this solution.

The goal is to make the input specification user-friendly. The idea is to let the programmer specify the input it as basic or advanced as they prefer. This means that they can either specify certain properties, or leave them open and automatically apply the default values of these properties.

3.2 Input Generation

In current fuzzing solution, such as BigFuzz, the framework can often be executed either with a user-defined seed file or without any seed file at all. The TabFuzz solution will provide the option to generate such a seed file, instead of running the program without a seed file. The program will use the input specification and generates a seed file that is valid according to all properties of the specification. TabFuzz will also still provide the option to run the program using a user-defined seed file.

3.3 Mutations

Mutations are used to create more test inputs by modifying the seed file. The BigFuzz authors describe several high-level mutations, these mutations are used for TabFuzz. On top of that, TabFuzz will also be frequently run to analyze what sort of errors take noticeably longer to detect than others. In case a pattern can be discovered in what sort of inputs cause these errors, new mutations may be introduced in order to find these failures quicker. The idea of TabFuzz is that it is an extendable framework, which can always be improved and optimized.

4 TabFuzz

This section explains how the methods described in section 4 are being applied. Section 4.1 focuses on the important properties to represent the input and how the programmer is able to specify these properties. Section 4.2 describes a method to generate a file that is valid according to the input specification. In section 4.3 the method to mutate files is described. These mutations generate new test inputs that mimic real-world problems in order to find erroneous behaviour in big data programs. Section 4.4 discusses how the fuzz testing works.

4.1 Input Specification

In the input specification the programmer can enter several properties per column.

TabFuzz offers a way for program to define an input specification. In this input specification the programmer can create columns and add properties to these columns. Some of these properties determine whether some input is a valid or invalid input. Other properties are used to increase efficiency. The possible properties are as follows: Datatype, Range, Special values, Column name and Repeat.

Datatype is the only column-property that is essential. The supported datatypes for the program are all the primitive datatypes and Strings.

Range is an optional column-property, but determines together with the datatype whether something is valid or invalid data. In case the range is left open, the program assumes that all values within the boundaries of the datatype are valid data entries. However if the programmer wishes to define a range on a column then they can do so as follows: Ranges on Strings and chars can be defined using Regular Expressions (REGEX), ranges on numeric values can be defined using one or more intervals, ranges on booleans can be set to either true or false.

Special values are specifically introduced in this research, to allow the programmer to specify values that may be interesting to test the program with. Bringing the programmer inside the loop can increase the efficiency of testing, programmers often know details about the internals of the program that can help speed up the testing process. Specifying special values heavily increases the probability of testing with these values.

Column name is introduced in order to help the programmer keep an overview of the specification. It helps the programmer keep track of which column is which. The property is not important for the testing itself.

Repeat is introduced to support the possibility of a variable amount of columns. The programmer can add the repeat property to a certain column followed by the amount of times it should be repeated. Repeat: x-y means that it is repeated between x and y times. It uses the data specification of the certain column for each of the repeated columns.

On top of the properties per column, it is also possible for the programmer to alter the settings of the writer. These settings can also be defined in the same file as the rest of the input specification. The programmer can change: the separator character, which is the delimiter that separates the different columns; the quote character, which is used to enclose a column, in case it contains the separator character; and the escape character, which is used to escape the quote character. In case no writer settings were defined, the program uses the same format as was used for all the benchmarks of the BigFuzz research.

Besides the question of what information is important to specify, is the question of how the information should be specified. The definitive version of this format is shown in Figure 1.

```
Separator: $

Column: Zipcode
Datatype: String
Range: 900[0-9]{2}
Special: 90042

Column: Intervals
Datatype: int
Range: <-50, -10#15, 27#120, >1300
Special: 15, 27, 1301

Column: Multiple Choice
Datatype: char
Range: (a|b|c|d)

Datatype: String
Repeat: 4-7
```

Figure 1: Data specification format example

The Writer Settings can be optionally defined at the top of the file. Followed by the specification of all the columns, separated by an empty line. As mentioned earlier, the only property that must be defined is the datatype, the rest of the properties can simply be left out in case the programmer does not want to define these. The special values can be entered using comma separated values. The range can also be defined using comma separated intervals. The intervals can be defined as follows:

- $< x$ means x and all values smaller than x
- $> x$ means x and all values larger than x
- $x\#y$ means all values between x and y, including x and y

The full specification of the data is stored as Java objects and is used for the input generation and mutations.

4.2 Input Generation

In fuzzing solutions, such as BigFuzz, the program often uses a so-called seed file. A seed file should contain only valid inputs. Containing only valid inputs heavily increases the chance that the program does not not crash very early in the testing process. These files are usually provided by the programmer, and the ability to provide such a seed still remains in the TabFuzz solution. However, TabFuzz also gives the programmer the possibility to generate such a valid seed file. The program takes the data specification and generates a value per column of the appropriate datatype within a valid range. Using a fuzzing framework with a user-defined seed file almost always yields better results [3] than without such a file. Therefore in section 5 the effectiveness of user-defined seeds versus generated valid seeds is evaluated to see whether generated valid seeds can approach effectiveness of user defined seeds.

4.3 Input Mutations

The mutations generate new test inputs. The mutation is always applied to the seed file. It performs exactly one mutation per trial. TabFuzz consists of the following six mutations:
The **Data Distribution Mutation** replaces an element with another element of the same data type. This mutation either generates a valid or invalid input in terms of range. TabFuzz generates valid or invalid inputs using a REGEX generator called RgxGen [9]. This library allows the user to generate a string that is either within or not within a certain regular expression. The ranges on numeric values can be entered as one or multiple intervals. To generate a valid input TabFuzz randomly picks one interval and generates a value within that range. To generate an invalid numeric input, TabFuzz reverses all the intervals and randomly picks one of the reversed intervals and generates a value within that interval.
The **Data Type Mutation** modifies the data type of an element. BigFuzz always makes the element keep the same value (e.g. changing an integer with value 20 to 20.0). TabFuzz approaches this mutation slightly differently. Since each datatype is essentially stored as a String (it is an inputfile), it means that in order to make its data type invalid, it must become a value that is not within the bounds of its datatype (e.g. trying to change an

integer to a short does not make any difference, since any short can also be read as an integer). Therefore the following changes of datatypes are possible:

Numeric values: are increased to be outside of the range of the datatype. They also get appended by non-numeric characters in order to make them into Strings. In case of whole-number types, ".0" is being appended to the element in order to try and make it a floating point value. While analyzing the results of the algorithm a discovery has been made, replacing a numeric value by an arithmetic operator would often cause an unique failure. Since the implementation at that time would only rarely generate these values, it would often take insanely high number of cycles (over tens of thousands in a program with only 3 columns) to detect these bugs. Therefore, the data type mutation for numeric values also replaces the element by an arithmetic operator every now and then.

Booleans: are replaced by any value that is not true or false.

Chars: are made invalid by replacing the element by a value that has more than one character.

Strings: are impossible to change its datatype by changing characters, since every sequence of characters is a valid String. Although there technically is a way to make a String invalid, which is done by entering more characters than the maximum amount of characters, this action is not performed due to performance issues.

The **Data Column Mutation** adds an extra column to a random row.

The **Null Data Mutation** removes an column from a random row.

The **Empty Data Mutation** picks a random column and replaces its value by the empty String. After analyzing the results of the program, it was discovered that one empty character would cause a unique failure. Therefore the empty data mutation picks either an empty string

The **Special Values Mutation** was introduced in TabFuzz. As explained earlier, the programmer can provide the program with special values. This mutation picks a random element and replaces it by one of its special values.

There are two scenarios in which the mutation does not perform anything meaningful. As explained earlier, in the case of Data Type Mutations, it does not perform anything interesting on Strings. In the case of Special Values Mutation, it does not perform any action in case the programmer did not specify any special values for the certain column. Therefore in the case one of these mutations are picked, it only chooses from the columns that can actually receive a meaningful change from the mutation. In case no such column exists, the program chooses another mutation to perform. This addition performs less meaningless mutations and the overall efficiency goes up.

The **Data Format Mutation** changes a delimiter by another character. This mutation is also one of the mutations described by BigFuzz. Since this character can theoretically be any character, this mutation would essentially just merge the two columsn into one. Moreover, the program splitting data on non-delimiter characters is problematic behaviour itself. This behaviour is already caught by basically any other sort of data input and/or mutation. Therefore this mutation is omitted from this research.

4.4 Fuzzing

TabFuzz start the fuzz testing by creating a JQF [8] fuzz testing environment. It generates a seed file if necessary. Then the fuzzing loop runs for the pre-defined amount of trials. Each trial, one random mutation is applied to the seed file and the program is tested using this randomly mutated file. This mutated file can either run the program successfully or cause an (unique) exception. An exception is considered unique if its stack trace is unique. Once the fuzzing is completed, the results are saved to the output folder. This output folder contains: the initial seed file, all the mutated files, a list of all the unique failures (including their stack trace) and the random-seed that was used for the test run. This random-seed can be used for reproducing the results.

5 Evaluation

In this section the effectiveness of TabFuzz is evaluated. The effectiveness is expressed in amount of unique failures found within a certain amount of trials. Firstly, the TabFuzz solution is compared to the TabFuzz-basic[1] solution. Secondly,

[1]TabFuzz could not be compared to the BigFuzz solution, more on that can be found in Responsible Research. TabFuzz-basic is the TabFuzz solution, without any of the additional functionality proposed in this research and represents a rough version of the BigFuzz framework.

the effectiveness of seed generation is compared to running the program with pre-defined seed files. The solution is tested against 12 benchmarks (Table 1). Section 5.1 presents the findings. The rest of the sections explain the results.

5.1 Results

The results can be found in Figure2. Each of the presented results is the average of 50 independent runs of 5000 trials. The x-axis represents the amount of unique failures. The y-axis represents the number of trials. Four benchmarks did not return a single failure and therefore the corresponding graph is trivial. These graphs of these four benchmarks have been omitted from the paper.

5.2 Seed File Generation

TabFuzz shows huge improvement in fuzzing without a seed file. Current fuzzing tools [3] notice a considerable decline of performance when running the tool without a seed file. The performance of running TabFuzz with or without a seed file is comparable. This means that TabFuzz elmiminates the need to fuzz with a user-defined seed file. Interestingly, in StudentGrade, the runs without a seed file even outperform the runs with a seed file.

5.3 TabFuzz Effectiveness

The TabFuzz framework outperforms the TabFuzz-basic framework on several benchmarks. All of the benchmarks where it outperforms are benchmarks that accept numeric values as input. As mentioned earlier, arithmetic operators on places where numeric values are expected introduce unique failures, however it often took an enormous amount of trials to randomly replace a numeric value by one of these artihmetic operator. Therefore TabFuzz does this operation more frequently and the results suggest that this is the addition that boosts the performance of TabFuzz compared to TabFuzz-basic.

5.4 Zero Failures

As mentioned earlier, neither of the test frameworks can find a single failure on four of the benchmarks. These benchmarks are: WordCount, ExternalCall, MapString and IncomeAggregation. For the first three of the benchmarks it can be easily explained why there was not a single bug. Each of these three benchmarks expect a single column of datatype String. Since every character and combination of characters can be interpreted as a String, these programs accept every sort of input. The other benchmark, IncomeAggregation, does have a structured input, on which one would expect the program to crash. Inspecting the internals of the benchmark suggested that there are pieces of code that should throw exceptions on certain inputs. For example, one operation in IncomeAggregation tries to parse values of multiple columns to integers. In case there is a wrong number of columns it should throw an `ArrayIndexOutOfBoundsException`. In case the column does not contain an integer, it should throw an `NumberFormatException`. Both of these scenarios do occur in the test data and therefore it is unclear why the fuzzer cannot find a single failure in this benchmark.

6 Responsible Research

There are several aspects that should be taken into account when reading this research. Each of the following sections introduces an ethical aspect of the research.

6.1 Overrelying risk

Testing frameworks, such as TabFuzz, introduce a risk that programmers start relying on these programs. However, one of the problems with random fuzzing is that there is no guarantee that it finds all bugs within a certain amount of trials. There is not even a guarantee that some bugs are found at all. Fuzzing tools, such as TabFuzz, definitely speed up the process of finding corner cases and exceptions, however they should never fully replace all other methods of testing or verification.

6.2 Empty stacktrace bug

During the course of this project, one bug was found that influences the results of this research. The amount of unique failures are measured by the amount of unique stack traces found during the run of the program. However, sometimes these stack traces would be null. The error has been investigated and seems to occur somewhere in the JQF or JUnit framework. Therefore this issue has been considered out of the scope of this research.

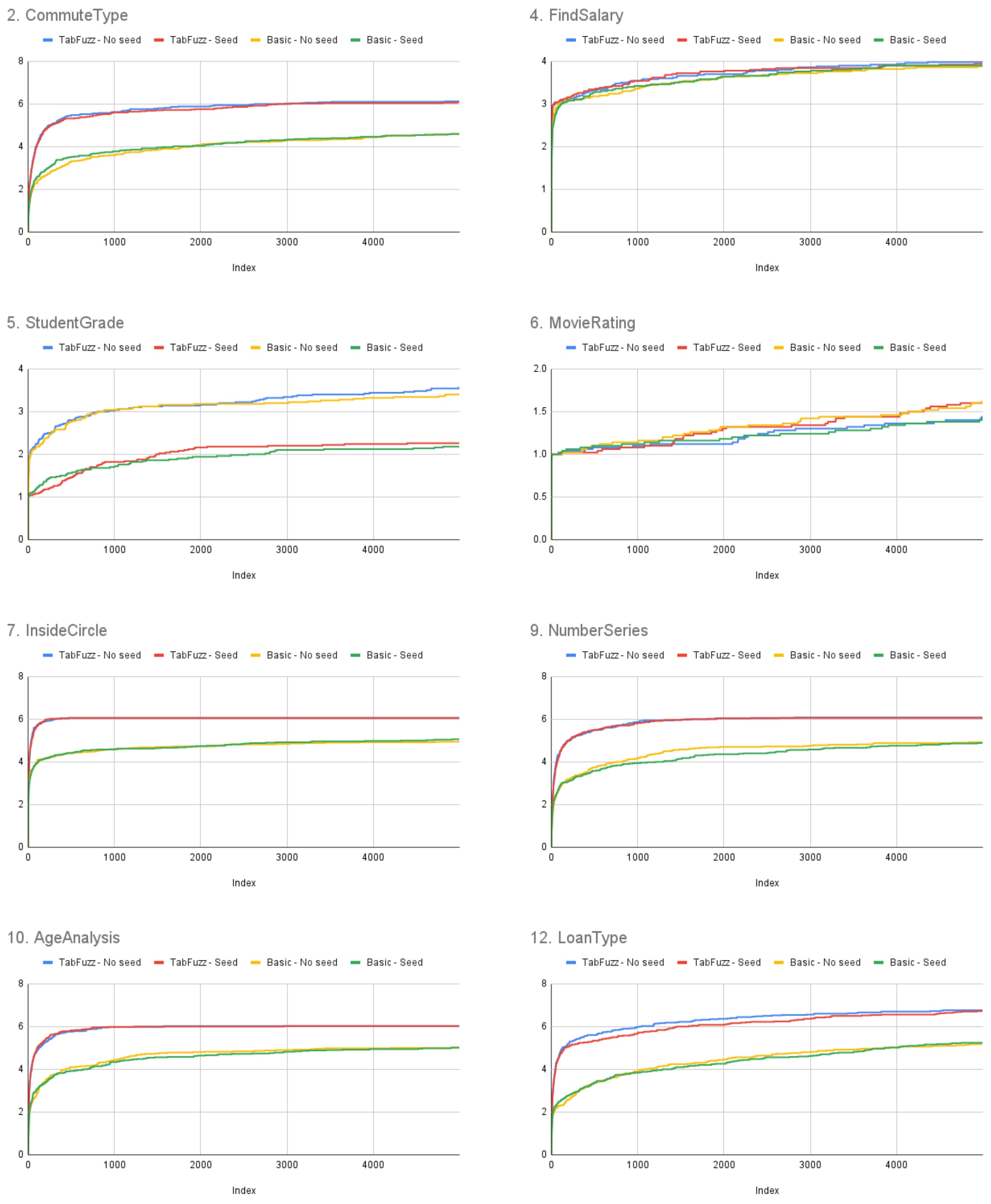

Figure 2: Results

ID	Subject Program	Description
1	WordCount	Find the frequency of words
2	CommuteType	People count using each form of transport for daily commute
3	ExternalCall	Find the frequency of words
4	FindSalary	Total income of individuals earning <= \$300 weekly
5	StudentGrade	List of classes with more than 5 failing students
6	MovieRating	Total number of mobies with rating >= 4
7	InsideCircle	Check whether the point (x,y) is in a circle
8	MapString	String mapping
9	NumberSeries	Find the numbers whose 3n+1 series' length is 25
10	AgeAnalysis	Total number of people with different age ranges
11	IncomeAggregation	Average income per age range in a district
12	LoanType	The frequency of each loan tpye within a metropolitan area

Table 1: Benchmarks [3]

The bug, however is not a massive problem for the following reasons. First, it can only occur once per run of the program, this is because an empty stack trace is also considered a unique stack trace. Secondly, the empty stack traces always seem to be duplicate failures, this was concluded after inspecting the input files on which it returned duplicate failures. These input files were often duplicates of files earlier in the run. This suggests that no unique failures were missed, because the empty stack trace was already in the unique failure list. Finally, the amount of unique failures found is not that important for the users, the users are mostly interested in what errors were found and what input caused the error. Both of these pieces of information are still available.

6.3 BigFuzz

In this research the plan was to replicate and extend the BigFuzz solution. However, unfortunately we did not manage to get the BigFuzz framework to work. The BigFuzz solution provides a tool to transform a scala program into a java version using a tool called the transformer. It then tests these java versions of the program using the JQF framework and the input files generated by their mutation class. Unfortunately the transformer did not work as intended, the programs generated by this class did not compile. Furthermore the mutation class seemed incomplete, it did not contain the mutations as described in the paper. The most likely explanation is that the BigFuzz repository that we used did not contain the final version of BigFuzz. Unfortunately this means that it was not possible to compare the results of TabFuzz to the results of BigFuzz.

6.4 Reproducibility

The results presented in this paper can be reproduced by using the same random seeds given in the repository. The repository also contains a file with instructions on how to replicate the results. The versions that were used to create the results are on the branches `TabFuzz-unaltered-final` and `Tabfuzz-basic-final`.

7 Conclusions and Future Work

TabFuzz provides programmers a framework to specify tabular inputs and mutating these inputs. A programmer can specify an input structure by defining the columns and specifying the properties of these columns. A column can be specified using the properties: Datatype, range, column name, special values and whether it should be repeated or not. The program can then generate inputs based on these specifications. For Strings and characters, this is done by generating inputs within a regular expression. For numeric values this is done by generating values within a certain interval. After this valid file is generated, the mutating can begin. These mutations are selected in attempt to mimic real-world problems.

The effectiveness of the framework was then tested. The TabFuzz solution has shown to be effective. Also the generated seed file has a negligible difference with using a user-defined seed file. This is a huge improvement compared to other fuzzing solutions, where the authors notice a decline of effectiveness in case there is no user-defined seed file.

As for future works, the solution presented in this research can always be optimized and expanded. Many improvements can be made to increase the effectiveness of the framework.

First of all, redundant test inputs can be eliminated. A duplicate file should in theory always cause the same errors. Removing this would significantly reduce the amount of trials required to find failures. Duplicate files are currently a common occurence, because not every mutation has an endless amount of possible new values. The null data mutations, for example, can only create a very limited amount of unique files, yet it is chosen 1/6th of the time.

Secondly, ranges on numeric values can be improved. Currently it can only accept one or more intervals. Allowing the programmer to specify a range based on a mathematical formula gives the programmer a lot more freedom in specifying ranges.

Finally, several functionalities can be improved. First of all, variable amount of columns only works on the last column as of now. Allowing the programmer to define this anywhere in the specification allows more types of input specifications. Secondly, the writer settings functionality does not work as intended. The CSV-reader library does not parse custom delimiters properly. Finally, the RgxGen generator is an impressive tool, however it does have its limitations. For example certain expressions contain serious issues with the distribution of values.

References

[1] Statista. *• Global Big Data market size 2011-2027 | Statista.* 2020. URL: https://www.statista.com/statistics/254266/global-big-data-market-forecast/ (visited on 05/06/2021).

[2] Muhammad Ali Gulzar, Madanlal Musuvathi, and Miryung Kim. “BigTest: A Symbolic Execution Based Systematic Test Generation Tool for Apache Spark”. In: *Proceedings - 2020 ACM/IEEE 42nd International Conference on Software Engineering: Companion, ICSE-Companion 2020.* ACM, 2020, pp. 61–64. ISBN: 9781450371223. DOI: 10.1145/3377812.3382145.

[3] Qian Zhang et al. “BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction”. In: *Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020.* Australia, 2020, pp. 722–733. ISBN: 9781450367684. DOI: 10.1145/3324884.3416641. URL: https://doi.org/10.1145/3324884.3416641.

[4] Michael Sutton, Adam Greene, and Perdam Amini. *Fuzzing - Brute Force Vulnerability Discovery.* 2007. URL: http://bxi.es/Reversing-Exploiting/Fuzzing%20Brute%20Force%20Vulnerability%20Discovery.pdf.

[5] Lei Zhang et al. “Improvement of the sample mutation strategy based on fuzzing framework peach”. In: *2018 International Conference on Artificial Intelligence and Big Data, ICAIBD 2018* (2018), pp. 33–37. DOI: 10.1109/ICAIBD.2018.8396162.

[6] Valentin J.M. Manès et al. *The Art, Science, and Engineering of Fuzzing: A Survey.* 2018. arXiv: 1812.00140.

[7] Gustavo Grieco et al. “QuickFuzz testing for fun and profit”. In: *Journal of Systems and Software* 134 (2017), pp. 340–354. ISSN: 01641212. DOI: 10.1016/j.jss.2017.09.018. URL: http://quickfuzz.org/..

[8] 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, pp. 398–401. ISBN: 9781450362245. DOI: 10.1145/3293882.3339002. URL: https://doi.org/10.1145/3293882.3339002..

[9] *GitHub - curious-odd-man/RgxGen: Regex: generate matching and non matching strings based on regex pattern.* URL: https://github.com/curious-odd-man/RgxGen (visited on 06/03/2021).