CS293C: SOFTWARE FUZZING FOR CORRECTNESS AND SECURITY - WINTER 2022

FINAL PROJECT REPORT

VYFuzz

GitHub - https://github.com/saikumarysk/cs293c_final

Team Members:
Sharath Chandra Vemula
Saikumar Yadugiri

Instructor:
Prof. Benjamin Hardekopf

March 11, 2022

Contents

1 Introduction

Fuzzing or fuzz testing is an automatic software testing paradigm in which random inputs are generated and are fed to a software in the hopes of finding issues in it. In a typical fuzz testing scenario, the software which is generating the random inputs is called a ***Fuzzer*** and the software which is being tested is appropriately called ***Software Under Test(SUT)***. The phrase "random inputs" can have diverse meaning. It could mean that they are valid inputs that the SUT is expecting. In this case, SUT is expected to provide valid results for the input. It could be invalid format in which case SUT is expected to behave unpredictably with the least amount of information about it's internal state as possible. The degree of invalidity of the inputs can vary widely. It could be as simple as providing an empty string when the SUT is expecting a non-empty string or providing an integer instead of a string or providing Java code instead of rust code.

In all of these scenarios, the main aim of executing the fuzz testing task is to make sure that the SUT is robust under any likely real-world scenario. Providing invalid inputs is also a real-world scenario as people who are using the SUT might not be aware of its correct usage or it could be a malicious user who is trying to extract information about SUT's inner workings. In any case, a software should be robust enough to handle not a 100% of all attacks, but at least close to it. With fuzzing we can reveal the inherent flaws present in SUT and maybe provide a way of fixing them as well.

Since the inception of fuzzing in 1988, it has been widely used in industry for testing developer branches to production-ready code. In academia, it has been a well-studied research area with several hundreds of papers written each year to multiple conferences. In both academia and industry, the major idea of fuzzing is the same as we described. Use/write a custom fuzzer useful for testing a specific SUT and feed the random inputs from the fuzzer to the SUT to generate various metrics. Using these, we can find the robustness of the SUT under various scenarios. In all of the input generation paradigms, the interesting ones are those which seem valid and random for the initial layers of the software but can produce a failure in the deeper layers. Much like a Trojan horse, a cunning input should be able to pass the initial validation so that we can explore the flaws present in the deeper layers of SUT. This is where Grammar-Based Fuzzing is quite useful.

1.1 Definitions

1.1.1 Grammar

In the context of computer science, grammar is a set of generation rules that describes how a language can be generated from a set of alphabets. Grammar does not specify what can be done with the language, but how it can be generated. In the context of our current project, grammar can be though of as a set of rules that can be used to generate an input from a set of alphabets. The set of abstractions used for generation are called non-terminals and the alphabets are called terminals.

In our project, we use a more specific form of grammar in the Chomsky hierarchy called Context-Free Grammars. In a context-free grammar, each production rule has a mapping from a single non-terminal to any combination of non-terminals and terminals. This mapping can also map to an empty terminal. This restriction allows efficient recognition and generation algorithms as any terminal in the language, regardless of what symbols surround it(the context of the symbol), can be replaced by the non-terminal from which it is generated. Although this replacement might not be unique, context-free grammars provide rich mathematical constructs to work with them. Languages which can be recognised by context-free grammars are called context-free languages. A few examples of context-free languages are data interchange formats such as JSON, XML, modern programming and markup languages such as C, Python, HTML etc.

1.1.2 Grey-Box Testing

Whenever we consider an SUT, we also have to think about how it could be used. In the real world, it could be argued that a majority of software is used without considering it's explicit implementation details. But while using a software, certain information about it can be extracted as well. For example, in the security domain, side-channel information such as the time it took for the software to execute for a certain input, its memory usage, power consumption, network packet information can be extracted without worrying about the SUT's implementation aspects. It could also be the case that a software is open-source and anyone can check its code and derive useful information from it in the hopes of either improving it, destroying it or anything in between.

So, while using a software, especially for testing, the usage can be roughly categorized into three types -

1. White-Box Testing
2. Black-Box Testing
3. Grey-Box Testing

As the names suggest, white-box testing occurs when we have the source code of the SUT and we can derive useful information from it by pre-processing it or on-the-fly during execution. In black-box testing, we only check the input-output relation of the SUT and nothing more. Grey-box testing is somewhere in the middle of these. Although you might not have the source code, you can find interesting aspects of SUT's execution and use this to steer your approach accordingly. These definitions are not completely spot-on but they are enough for defining validity for our project.

1.1.3 Code Coverage

Code coverage is one of the various metrics we can extract from a program's execution to see how well the program responds to our input. Code Coverage can be be of various types. A few of them are -

1. Line Coverage
2. Function/Method Coverage
3. Branch Coverage
4. Path Coverage

As the names suggest, each of these take into account, the number of valid lines or functions or branches or paths in the program the execution covers. Of all these coverages, line and function coverage are easy to define and extract as line coverage is simply the number of lines the execution covers. Any external tool or the SUT compiler itself can instrument the SUT to find this information. Similarly, as functions are well defined in each software, it is also easy to extract function coverage information. In our project, we look at function coverage and line coverage as our metrics.

2 Structure

The rest of the paper is organised as follows. We briefly mention our work in section 3. For our project, we wrote a lot of Python programming files. We provide a brief code walkthrough for all our files and also explain the salient features and briefly touch upon the algorithms we used in section 4. In section 5 we dive into our experiment setting and results. We produce a few valid improvements in section 6 and conclude in section 7.

3 Our Work

The main aim of our project is to test the robustness of open-source JSON parsers. In order to do that, we wrote a new fuzzer ***VYFuzz***(S. **V**emula and S. **Y**adugiri **Fuzz**er) which uses JSON grammar from [1] and generates new JSON files. We use the Backus-Naur Grammar available for JSON and generate various JSON files which are guaranteed to be valid JSONs. To make sure that the generation engine for JSONs is working correctly, we use the command line tool `jsonlint` and check if the generated files are valid JSON files.

Using the generated JSON files, we test two open-source state of the art JSON parsers, `nlohmann/json` and Google's `gson`. While the former is a C++ parser for JSON files which can be used in CLI, the latter is a Java language parser for JSON files which, as all the documentations suggests, used as a Maven plugin for many Java projects. We use the latest version of the `nlohmann/json` source code but we went back a few versions for Google's `gson`. For the exact details on the version we used, please refer to Section 5. From here on out, we denote `nlohmann/json` as `json` and Google's `gson` as `gson`.

While testing these parsers, we collected the line coverage and function coverage metrics to see how well our inputs are able to cover the parser's code. Our assumption is that the more number of lines an input covers, the higher is the chance that the input will be able to cover hidden bugs. For finding the line coverage and function coverage for `json`, we used C++'s inherent `gcov` and `lcov`. For specific details on how we did this, refer to Section 4.1.5. For finding the line coverage and function coverage for `gson`, we used a third party library called `jacoco`. For specific details on how we used `jacoco`, please refer to Section 4.1.7 Note that in the case of `gson`, function coverage is technically method coverage as it is written entirely in Java.

Furthermore, we annotate the grammar production rules with float numbers denoting probabilities which are useful to steer the JSON production engine in a specific way. This technique is also called Probabilistic Grammar Fuzzing. Using the coverage metrics we gathered as described, we adjusted the probabilities of the grammar production rules to steer our input generation to produce JSON files which cover more of the parser's code. As we are only using the parsers in a grey-box manner where we are only collecting the coverage information of the execution, our testing is a coverage-guided grey-box fuzzing. This way, to summarise our project in a sentence, we developed a coverage-guided grey-box probabilistic probabilistic grammar fuzzer that generates JSON inputs to test `json` and `gson`.

To dive a bit deeper into our project, we generate JSON inputs using the probabilistic grammar fuzzer we wrote and execute these JSON files on `json` and `gson` libraries. Before executing the inputs, we instrument the libraries in order to get the line and function coverage information. We also check if the execution resulted in a failure using the return code of the execution. Using these three metrics, we sort all the inputs in the priority order of error, line coverage and function coverage. We chose to prioritise line coverage over function coverage as in case of `json`(which is a C++ parser), there could be lines which are not part of any function. And to be fair in our evaluation, we chose the same for `gson`(Java parser). This sorting acts as a power schedule for our fuzzer and in the next iteration of the testing, we take the top 10% of the inputs which produce errors, higher coverage and learn new probabilities for the grammar production. This way, in the next iteration of the process, we are trying to check if this new probabilistic grammar is able to produce higher coverage and/or more errors.

Initially, we use a probabilistic grammar which has uniform probability across all expansion rules. Using this, we generate 50,000 random JSON files and test `json` and `gson` libraries. From the resulting metrics, we extract the top 10% in both the cases and learn new probabilistic grammar and repeat the process 2 times for each SUT. In this process, we were able to find a bug in the `json` library. Although we were not able to find a bug in the `gson` library, we were able to produce high coverage yielding inputs in the latter iterations.

4 Code Walkthrough

As mentioned earlier, we wrote a few files which use a probabilistic grammar to generate random JSON files, and also files which take these inputs and test `json` and `gson` libraries. With the exception of 2 files, all the files are written in Python programming language. In this section, we provide a brief description of these files.

4.1 Fuzzing Files

4.1.1 Fuzzer.py

In this file, we define a class `Fuzzer` which uses a `DerivationTree` object to produce random JSON files using random weighted expansions(weighted on the probability of the expansion rules in the grammar). We also provide a way to reset the grammar used in the derivation tree so that when we update the probabilistic grammar, we can keep on using the same fuzzer object to save memory and time. When this file is run, it produces, on an average $\sim$ 17 kB of JSON string for uniformly random probabilistic grammar.

4.1.2 DerivationTree.py

In this file, we define a class `DerivationTree` which is the core part of our JSON file generation process. `DerivationTree` is an n-ary tree in which the root node is the start symbol of the probabilistic grammar and the leaves of the tree form the JSON string. Beginning from the root node, we randomly expand each node using the strategy provided in [2] and produce a valid JSON file. We also made a lot of optimizations in terms of caching and parsing the tree. In a typical tree produced, the number of levels will be more than 1500. So, we parse the tree iteratively to extract the JSON string instead of recursively as Python will produce a `RecursionError`. Similarly we cache many expansions and leaves to speed up the expansion process.

4.1.3 Grammar.py

In this file, we define a class `Grammar` which contains all the necessary methods required to handle the JSON grammar. We used a Backus-Naur form JSON grammar available at [1] and initially we assign each expansion uniform probability. One notable feature in this file is that the alphabet set include all the valid UTF-8 characters except the ones in the range \ud800 to \udfff. This is because these characters are reserved as UTF-16 surrogates in Python. We have also optimised most of the code present using caching before any operation on the the JSON grammar starts.

4.1.4 test.cpp

This is a C++ file we have written which uses the `json` library to parse a JSON file and convert it into a JSON object in C++. It takes a command line argument for the path to the JSON file.

4.1.5 CppJSONRunner.py

This file contains the code for instrumenting `json` library using `gcov` and `lcov`. The steps to instrument the library are as follows -

```
# Compiling test.cpp to include coverage information
$ g++ -I. -fprofile-arcs -ftest-coverage -lgcov
path_to_file/test.cpp
```

We then run the compiled C++ file with all the JSON files generated and extract the line and function coverage information.

```
# Execute the code to parse the JSON file (this step also
# generates gcda files which have information about coverage)
$ path_to_file/a.out path_to_json/json_file.json

# Generate the coverage report using gcov
$ gcov path_to_file/test.cpp

# Generating coverage.info data file using lcov
$ lcov --capture --directory path_to_output_dir --output-file
path_to_output_dir/main_coverage.info

# Generating visual code coverage HTML report from data file
$ genhtml path_to_output_dir/main_coverage.info
--output-directory path_to_output_dir/out
```

The stdout of the `genhtml` command produces the line and function coverage information and we extract that using Python regex. We repeat the same for all the JSON files in a directory and finally produce a CSV file which includes the name of the JSON file, error information, line and function coverages. For versions of all the packages, refer to Section 5.

4.1.6 Test.java

This file is similar to `test.cpp` except it is written in Java and parses JSON files and converts them into Java JSON objects.

4.1.7 JavaGSONRunner.py

This file is similar to `CppJSONRunner.py`. For instrumentation, we use a third party coverage generation `jar` file, `jacoco`. To be more precise, we use the command line version of `jacoco`, `jacococli.jar` The steps to instrument `gson` are as follows -

```
# Use jacococli to instrument gson
$ java -jar path_to_jacoco_lib/jacococli.jar instrument
path_to_gson/gson-2.8.3.jar --dest path_to_jacoco_lib

# Compile Test.java to instrument jacoco information
$ javac -cp '.:gson-2.8.3.jar:jacocoagent.jar' Test.java
```

We then run all the JSON files using these commands -

```
# Execute Test.class on JSON file
$ java -cp '.:gson-2.8.3.jar:jacocoagent.jar' Test
path_to_json/json_file.json

# Generate jacoco CSV report
$ java -jar jacococli.jar report jacoco.exec --classfiles
path_to_gson/gson-2.8.3.jar --csv test.csv
```

We then process the generated `test.csv` file to extract the line and function(method) coverage metrics and remove unnecessary files and repeat the process for all JSON files in a folder. For the versions of all libraries and packages, refer to Section 5.

4.2 Input Generation Files

4.2.1 InputGen.py

In this file, we use the `Fuzzer` object to generate random JSON strings and store them in various files ordered according the generation sequence. We use a strict 30 second timeout for generating any JSON file as we need to generate files in the order of tens of thousands.

4.2.2 EnsureCorrectJSON.py

As mentioned before, to make sure that we are generating valid JSON files, we use the command line tool `jsonlint` available as a Debian package to check all the files we generated are valid JSONs. The command to check the validity of a JSON file is

```
$ jsonlint path_to_json/filename.json
```

which succeeds with return code 0 and the stdout contains “filename.json: ok” if “filename.json” is a valid JSON file. In this file, we run this command on all the files in a folder and check how many fail. All the JSON files we generated as part of this project produce a valid result using `jsonlint`. So, we are assuming that the JSON files generated by our fuzzer are valid JSON files. For the version of the `jsonlint` package, check Section 5.

4.2.3 GenerateGrammar.py

In this file, we take a csv file and the number of rows to consider as command line arguments. We parse the csv file for the number of rows provided and for each input file in the row, we check the key and expansion counts derived from the derivation tree of the JSON file, to generate a new probabilistic grammar using the formula -

The probability p_i for each alternative a_i of a symbol $S \to a_1|a_2|\ldots|a_n$ is given by

$$p_i = \frac{\text{Expansions of } S \to a_i}{\text{Expansions of } S}$$

The numerator is the expansion counts we are storing and the denominator is the key counts. Note that some of the expansions will have the number of counts in the derivation tree to be zero. And in that case, the probability of such expansions will be set to zero. As we are starting with uniform random probability and as the above formula mathematically guarantees valid probabilities for all expansions, we are are not performing to check to see if probabilities add up to 1 in our `Grammar` class's `is_valid_grammar` method.

Moreover, to get the key and expansion counts, we need to parse the JSON file using the JSON grammar. But as we are generating the JSON file using a derivation tree, we are storing this information in the Counts directory whenever we generate the JSON file. The code for this is present in `InputGen.py` and `DerivationTree.py` files. Thus, we are skipping the parsing step entirely and improving the time of our fuzz testing.

4.2.4 GenerateNewInputs.py

This file is similar to `InputGen.py` except that we have a step to produce a new probabilistic grammar and feed it to the fuzzer before starting the generation of inputs. We use `GenerateGrammar.py` for this purpose.

4.3 Auxilary Files

4.3.1 CombineCSV.py

As we are dealing with JSON files in the size of kilobytes, the generation and testing takes a while. Hence, we generated and tested the JSON files in batches and produced multiple CSV files. As all executions are independent of each other, we can combine these CSV files and sort them accordingly to produce the same result as we would if we executed them together. In order to combine several CSV files together and generate a new file, we use this file.

4.3.2 TransferFiles.py

As mentioned above, we generated JSON files in batches and to transfer the files to various VMs, we used this file.

4.3.3 CppStatistics.py & JavaStatistics.py

We used these files to analyze all the final CSV files generated after testing to generate various metrics.

4.3.4 test.py

A common test file to unit test out Python implementations.

5 Experiments

Before proceeding to the details of our experiments and our final results, the following table provides the details of the versions of all the software packages and libraries we used.

All the testing was done in Ubuntu 20.04.3 LTS on a Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz quad core processor with 16 GB memory. A rough sketch of out experimentation process if given below -

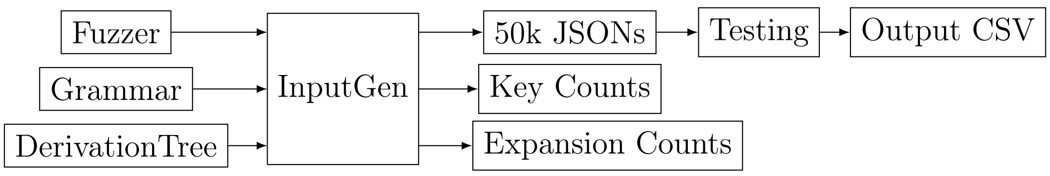


Figure 1: Iteration 0

Package	**Version**
Python	3.8.10
Java	1.8.0_312
gcc	9.4.0
gcov	9.4.0
lcov	1.15-5-g462f71d
genhtml	1.15-5-g462f71d
jsonlint	2.2.4
jacoco	0.8.7.202105040129
`nlohmann/json`	3.10.5
Google's `gson`	2.8.3

Table 1: Software Package Versions For Our Project

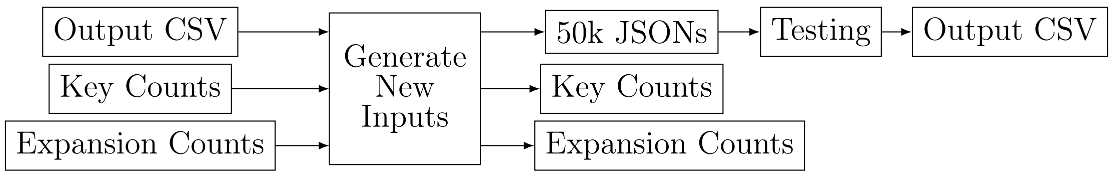


Figure 2: Iterations 1 & 2

5.1 Java gson Testing

5.1.1 Iteration 0

We start by using our `InputGen.py` to generate 50,000 random JSON files and store them in the folder `Inputs` directory in our github repository. Simultaneously, we store the key and expansion counts which will be useful xlater to generate a new probabilistic grammar in the folders `Counts/Key` and `Counts/Expansion` respectively. The process of generation of inputs took $\sim$ 24 hours on 5 parallelly running Linux VMs. As stated before, the approximate size of the JSON files is roughly 17 kB. This is quite high compared to other open-source grammar based JSON generators such as grammarinator, PyJFuzz, gramma etc. So, we do require a lot of time to generate these files.

We then used the folder `Inputs` in our `JavaGSONRunner.py` file and ran it. Using the process described in Section 4.1.7, we ran the parsing script which took roughly 8 hours for 50,000 inputs. At the end of the run, we generate the file, which contains all the coverage information, `0_java_gson_run.csv`.

5.1.2 Iteration 1

In this iteration, we use our file `0_java_gson_run.csv` in the `GenerateNewInputs.py` to generate new probabilistic grammar using the corresponding key and expansion count files for the top 10% = 5000 rows. We then proceed to generate 50,000 new JSON files using the new grammar and store the information in `1_Java_Inputs` and the corresponding key and expansion counts in `Counts/1_Java_Key` and `Counts/1_Java_Expansion`. This also took us $\sim$ 24 hours if we use 5 Linux VMs.

We then used the folder `1_Java_Inputs` in our `JavaGSONRunner.py` file and ran it for roughly 8 hours to generate the new line coverage information file, `1_java_gson_run.csv`.

5.1.3 Iteration 2

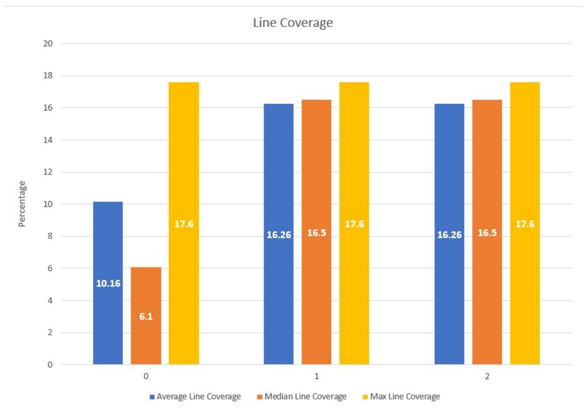
This is similar to iteration 1 where we repeated the process using `1_java_gson_run.csv` and generated `2_Java_Inputs`, `Counts/2_Java_Key`, and `Counts/2_Java_Expansion`. It took us similar amount of time for the iteration to complete. The final coverage results are present in `2_java_gson_run.csv`.

5.1.4 Results

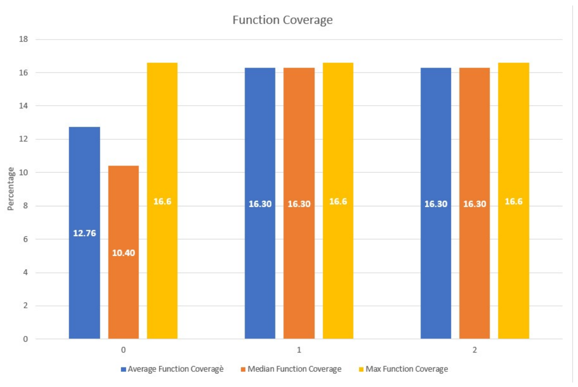
Based on the 3 CSV files which have the coverage results, we extracted a few metrics which are shown below.

	Line Coverage(%)			Function Coverage(%)			Error(%)
	Mean	Median	Max	Mean	Median	Max	
Iteration 0	10.161	6.1	17.6	12.75	10.4	16.6	0
Iteration 1	16.260	16.5	17.6	16.29	16.3	16.6	0
Iteration 2	16.260	16.5	17.6	16.29	16.3	16.6	0

Table 2: `gson` Metrics



(a) Line Coverages for 3 Iterations



(b) Function Coverages for 3 Iterations

Figure 3: `gson` Results

As we can see in the table 2 and in figure 3, we see an increase in the mean and the median line and function coverage from iteration 0 to iteration 1 and iteration 2. In iteration 0, we used uniformly random probabilistic grammar to generate our JSON files. In the latter iterations, we are learning a new probabilisitc grammar from the results of previous iteration's key and expansion counts.

The mean line coverage has increased by 60% from iteration 0 to iteration 2. The median line coverage increased by 170.5%. The mean function coverage increased by 27.7% and the median function coverage increased by 56.7%. We did not see any crashed or errors during our testing of `gson` parsing library. Hence, we are not including any graphs for that. This shows that using counting based grammar to update the probabilities on the fly during generation will produce new inputs which will increase the coverage. This is the "More of the Same" strategy discussed in the class.

5.2 C++ json Testing

5.2.1 Iteration 0

We used the same random 50,000 JSON files present in Inputs folder to test json using our CppJSONRunner.py file. The testing took ∼ 24 hours to complete and in the end, the coverage results are present in 0_cpp_json_run.csv file.

During this, we found triggers in the json library where it is not able to handle large numbers. The file Inputs/input-9984.json is a representative file for all the triggers generated. This trigger is a bug as the JSON file input-9984.json is a valid JSON file(confirmed by jsonlint as well), and Google's gson was also able to handle this input without crashing. So, we believe that with our fuzzer, we were able to identify a bug in the C++'s nlohmann/json JSON parsing library. We reported the bug to the nlohmann/json repository on github as well.

5.2.2 Iteration 1

This is also similar to iteration 1 of gson testing where we learned a new probabilistic grammar from 0_cpp_json_run.csv file. We then generated 50,000 new JSON files similar to what we did in gson testing, using the file GenerateNewInputs.py. But a major hurdle we faced is that due to the previous iteration's top results favour large numbers, the new probabilistic grammar heavily generates these files. A typical size of these large number JSON files generated from our fuzzer is roughly 230 kB. So, it took us ∼ 24 hours to generate 10,000 JSON files using this new probabilistic grammar. Hence, for this iteration, we only work with these 10,000 JSON files.

In this iteration, a majority of the JSON files generated triggers which correspond to the previous bug. The results of this iteration are 1_cpp_json_run.csv. And the JSON files for this iteration are present in 1_Cpp_Inputs, Counts/1_Cpp_Key, and Counts/1_Cpp_Expansion.

5.2.3 Iteration 2

In this iteration, we use the probabilistic grammar generated by GenerateGrammar.py and 1_cpp_json_run.csv to generate new JSON inputs and repeat our testing. In this iteration as well, we used 10,000 JSON files due to the same restriction we were facing in the previous iteration. The results of this iteration are 2_cpp_json_run.csv. And the JSON files for this iteration are present in 2_Cpp_Inputs, Counts/2_Cpp_Key, and Counts/2_Cpp_Expansion.

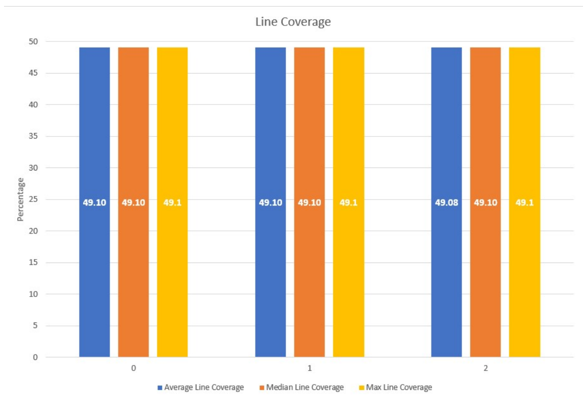
5.2.4 Results

Based on the 3 CSV files which have the coverage results, we extracted a few metrics which are shown below.

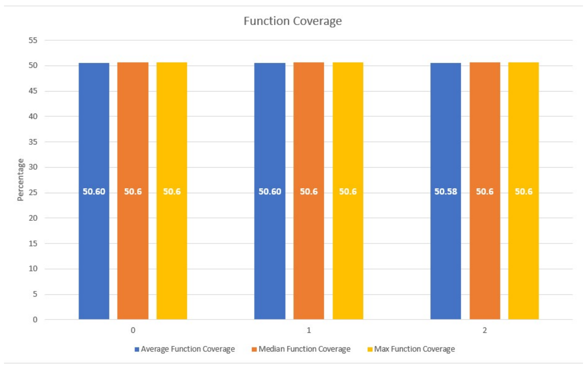
	Line Coverage(%)			Function Coverage(%)			Error(%)
	Mean	Median	Max	Mean	Median	Max	
Iteration 0	49.097	49.1	49.1	50.597	50.6	50.6	8.12
Iteration 1	49.097	49.1	49.1	50.598	50.6	50.6	62.63
Iteration 2	49.077	49.1	49.1	50.577	50.6	50.6	62.30

Table 3: json Metrics

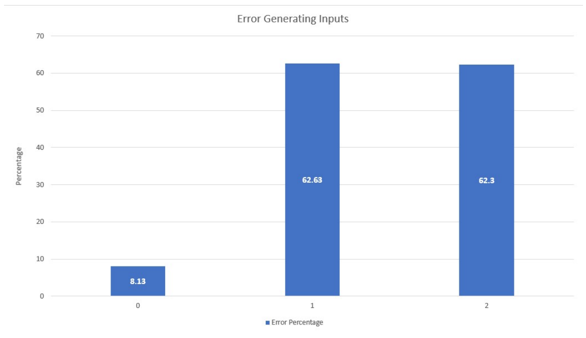
As we can see from table 3 and figure 4, we do not see much improvement in the line and function coverage metrics. But there is more than 666.3% increase in the number of JSON files that resulted in triggers. This is expected as we are prioritising errors over coverage in our



(a) Line Coverages for 3 Iterations



(b) Function Coverages for 3 Iterations



(c) Error Producing Input Percentage

Figure 4: `json` Results

experiments. This shows that "More of the Same" approach also generates inputs that produce similar triggers.

6 Future Improvements

As part of this project, we wrote a minimalistic fuzzer that works exclusively for JSON grammar. The features present in our fuzzer are essential for any probabilistic grammar fuzzer. We provide a list of improvements that can be implemented within a reasonable amount of time.

1. **General Purpose Fuzzer:** Our fuzzing code will work only with JSON grammar. We would like to increase to scope to take any kind of BNF or EBNF grammar.
2. **Code Optimisation:** The current input generation produces JSON files of large size and although this is what has allowed us to discover the bug for `json`, optimising our code to produce the same large outputs but in a short time will be a good improvement. One useful technique would be to use [4].
3. **User Friendliness:** We used our Python files in a ad-hoc manner and not included many user-friendly flags to make it general-purpose use. For instance, we manually edit the `GenerateNewInputs.py` to use the folder for inputs. Passing this using command line arguments will help in automation of our project
4. **Automation**: As mentioned in our project proposal, we wanted to automate our flow. If we had enough time, using the above two optimisations, automation would've been quite trivial as we do not much manual interference between iterations.

5. **Unlike the Rest:** In our project, we followed the “More of the Same” approach. But the other direction where we adjust the probabilities in `GenerateGrammar.py` which represent the polar opposite direction is also something we could’ve explored.

In our initial ideas, we wanted to test SUTs using mutation as well. We wanted to take a three-pronged approach where we test using adjusted grammar probabilities, tree mutations(valid and invalid), and newly generated uniform random grammar. This way, we believed that we would be able to produce various kinds of triggers. We understand that this is not reasonable to do in a course project. But it is one direction that we wanted to pursue.

7 Conclusion

In our project, we wrote a probabilistic grammar based fuzzer for JSON grammar and tested two SUTs, `nlohmann/json` and Google’s `gson`. We generated 50,000 uniformly random JSON files for iteration 0 and used the coverage metrics to generate more of the same JSON files in the upcoming 2 iterations. For `json`, due to time constraints, we only worked with 10,000 JSON files in the latter iterations. But using our testing we were able to see that more of the same approach helps in producing inputs that increase the coverage. We were also able to find a bug in `json` and reported it to the github repo.

ACKNOWLEDGEMENTS We would like to thank our instructor Prof. Benjamin Hardekopf for helping us finish the project and also guiding us in the required direction throughout the project. We would also like to thank our fellow classmate Nikunj Baid, our peers in Technology Management, Dharav Mehta and Kumaresan Mudliar for generously donating their College of Engineering Linux VMs which enabled us to parallelise our input generation and testing process.

References

[1] json.org - Website for JSON data-interchange format compliant with ECMA-404 definition

[2] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler: "Efficient Grammar Fuzzing". In Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler, "The Fuzzing Book", https://www.fuzzingbook.org/html/GrammarFuzzer.html.

[3] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler: "Probabilistic Grammar Fuzzing". In Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler, "The Fuzzing Book", https://www.fuzzingbook.org/html/ProbabilisticGrammarFuzzer.html

[4] Gopinath, Rahul, and Andreas Zeller. "Building fast fuzzers." arXiv preprint arXiv:1911.07707 (2019).