An Enhanced Grammar-based Fuzzing Tool

Alyssa Yiqin Huang
UW ID: 20868286
y672huan@uwaterloo.ca

Su Jia Yin
UW ID: 20812709
sjyin@uwaterloo.ca

Dawson(Xiaohui) Men
UW ID: xmen
xmen@uwaterloo.ca

August 20, 2020

Abstract

We enhanced `gramfuzz`, a grammar-based fuzzing tool, by designed a runner program to select the program-under-test and its grammar, feed the inputs automatically, save suspicious inputs and display the testing status in an interface. Compared to the mutation-based fuzzing tool, our enhanced grammar-based fuzzing tool penetrates deeper into benchmarks' parsers. The grammar-based fuzzing tool generates less inputs that would be rejected, and finds more exceptions when feeding the same number of inputs to the benchmarks.

1 Introduction

It is difficult to explore unexpected or abnormal behaviour in software engineering, especially for closed source software.[6] Due to the complexity of software, it is almost unavoidable that several errors or exceptions, which may lead to economic loss or even security and safety problems, occurs in a piece of software. As Paul Strassman commented, software easily rates as the most poorly constructed, unreliable, and least maintainable technological artifacts invented by man. Thus, it is important to test a piece of software before it is used as a system component. It is also interesting to find out what kind of inputs could trigger the software-under-test to crash.

When we can not access the source codes or I/O behaviours of the target software, it is impossible to use gray-boxed or white-boxed methods, including static analysis or concolic testing. Thus, though with several disadvantages, black-box testing methods, such as fuzzing, are necessary.[1] Among all kinds of black-box fuzzing methods, the grammar-based fuzzing method is very suitable for the scenario of using closed source software as system components.[2, 4] As known to be an effective technique for checking security vulnerabilities in programs, grammar-based fuzzer generates well-formatted inputs by specifying inputs via the grammar of the program-under-test. [3, 5]. Furthermore, the application programming interface and the input format generated by other programs are already known, which limits the formats and range of inputs and decreases the cost

of writing grammars.

As we noticed, most of the grammar-based fuzzing tools require the users to input the generated files to the program-under-test manually, which would be extremely inefficient when there are thousands of test cases. Thus, we designed a runner program to feed the inputs generated by fuzzer into the program-under-test automatically and display the current status with an interface.

2 Description of the method

2.1 Framework or toolchain used

2.1.1 Fuzzing tools

We choose `gramfuzz`, an open-source grammar-based fuzzing Python library, to write the grammars for benchmarks and designed an input generator. As a comparison, we choose `pyZZUF`, a Python library implements bit-flip zzuf mutator, to generate inputs according to random inputs generated by `gramfuzz`.

2.1.2 Runner program

To increase the efficiency of the fuzzing test, we designed a runner program. As Fig. 1 shows that the runner program's interface allows the user to choose the fuzzer, benchmark, input type (feed), and timeout. The user can also select the number of input files that will be generated and fed to the benchmark in a single round and how many rounds will be executed.

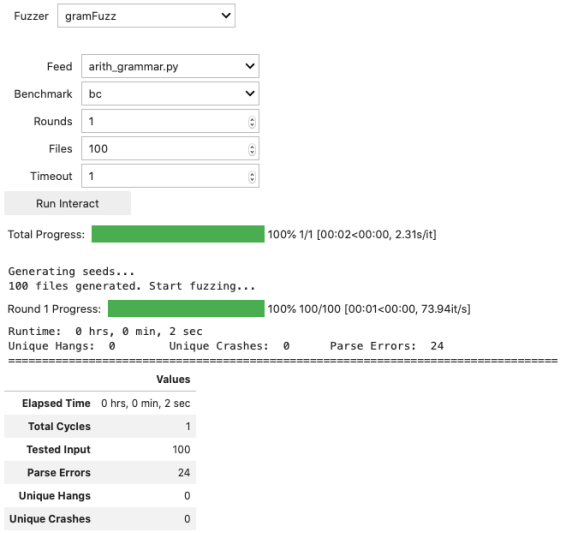


Figure 1: The interface of runner program

Fig. 2 shows the design of the runner program, which is designed based on the strategy design pattern. From the interface, the user can select the wanted fuzzer as the strategy. Then the strategy would be instantiated. For different instances, the program would invoke its generator, then put parameters user chose, including round, files, and instantiate `FuzzRunner`. The `FuzzRunner` would then be invoked to feed the generated inputs to the benchmarks. If any exception occurs, the `FuzzRunner` would call `record_exp()` to handle the exception and save the input that triggers the exception. Finally, the `FuzzRunner` would call `get_result()` to display the final results of the testing. Moreover, the `TestRunner` visualizes the process of testing by calling `run_widet()` to display the user interface.

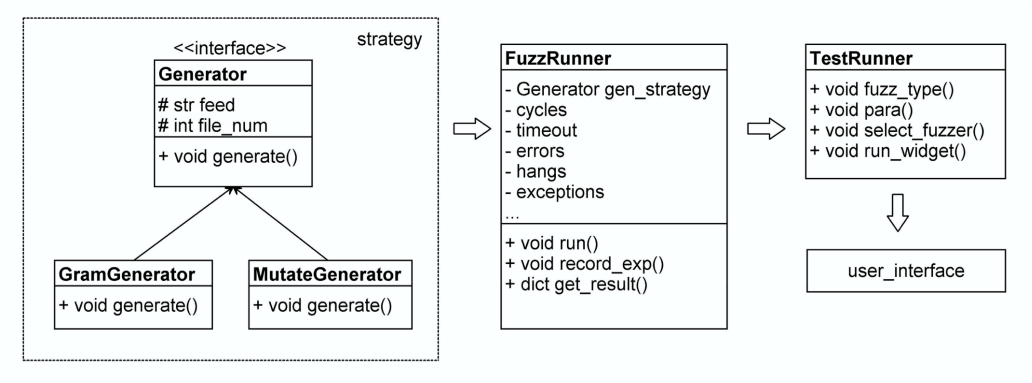


Figure 2: UML of runner program

2.2 Benchmarks used

We chose two calculators as benchmarks. One is `bc`, another one is `calc`. Both two programs are command-line programs, and their input formats are also similar to each other. Therefore, it would be clear to illustrate the strength of grammar-based fuzzing that it can decrease the possibility of the inputs rejected by the parser and penetrate more in-depth than other fuzzing methods by generating well-formatted inputs.

Furthermore, `bc` and `calc` could be considered as representative of program-under-test, because their inputs can be divided into two types, one type is the arithmetical calculation, and the other type is program statement, including defining or calling functions, if-else conditional branchings, for loops and while loops. Thus, these two benchmarks are very typical, which makes the results have general meanings.

2.3 Grammar designed

We noticed that the average execution time for two benchmarks is significantly longer when the input is arithmetical calculation. It is because that big numbers and complicated calculations, such

as exponent arithmetic, will trigger long-running uninterruptible computation. Since we detected the program hang by setting a timeout, we designed grammar for arithmetical calculation inputs and program statement inputs separately. Thus, we could set a longer timeout for arithmetical calculation when running the test. As for mutation-based fuzzer, we used seed inputs generated by the grammar-based fuzzer according to specific grammars to mutate and generate inputs for the specific type of inputs.

3 Results

3.1 Experimental setup

We run the testing on Mac OS X. Table 3.1 shows the version of software or library we used.

Table 1: Experimental setup

Software/Library	Version
`gramfuzz`	1.14.0
`pyZZUF`	0.1
`bc`	1.06
`calc`	2.12.7.1
Python	3.7.6

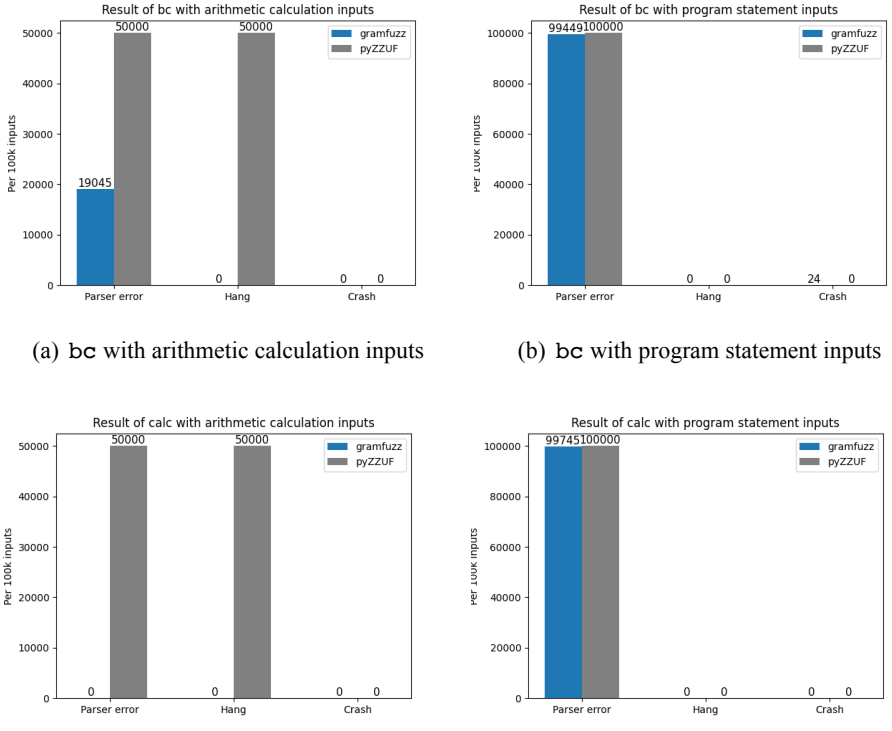
3.2 Experimental result

Table 3.2 shows the experimental results. We found crashes when using `gramfuzz` to test `bc` with program statement inputs, which were triggered by the segment fault.

Table 2: Experimental result for 100000 tested inputs

Fuzzer	Benchmark	Input type	Parser errors	Hang	Crash	Duration
`bc`	Arithmetic	`gramfuzz`	19045	0	0	00:30:52
		`pyZZUF`	50000	50000	0	14:10:24
	Statement	`gramfuzz`	9449	0	24	00:27:17
		`pyZZUF`	100000	0	0	00:51:06
`calc`	Arithmetic	`gramfuzz`	0	0	0	00:35:59
		`pyZZUF`	50000	50000	0	14:23:54
	Statement	`gramfuzz`	99745	0	0	00:47:46
		`pyZZUF`	100000	0	0	00:52:36

Compared `gramfuzz` to `pyZZUF` according to the data in Table 3.2, the results are presented as Fig. 3.2 and Fig. 3.2. From Fig. 3.2, we could see that the inputs generated by `gramfuzz` were significantly less likely to be rejected by the parser than inputs generated by `pyZZUF`, regardless of the benchmarks and the input type. Moreover, only inputs generated by `gramfuzz` were found to trigger the benchmark crashing. Furthermore, inputs generated by `pyZZUF` were more likely to make the benchmark hang.



(a) `bc` with arithmetic calculation inputs

(b) `bc` with program statement inputs

(c) `calc` with arithmetic calculation inputs

(d) `calc` with program statement inputs

Figure 3: Results compared `gramfuzz` to `pyZZUF`

From Fig. 3.2, we concluded that the grammar-based fuzzing tool was more efficient than the mutation-based fuzzing tool because it took less time to test 100 thousand input files in all cases when using `gramfuzz`, especially when the input type was arithmetic calculation.

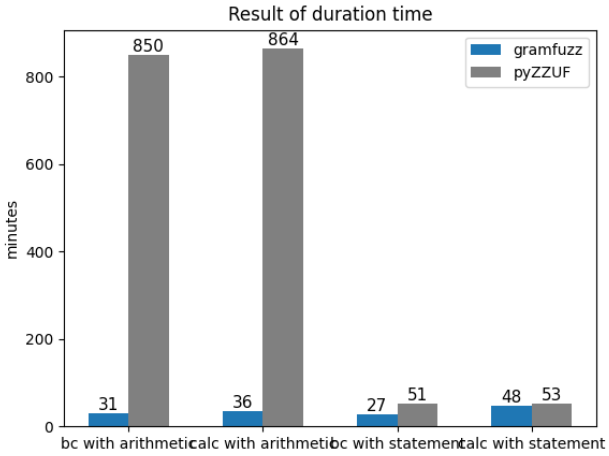


Figure 4: Duration time for testing 100000 inputs file

4 Conclusions and future work

4.1 Conclusions

The runner program we designed increased the efficiency when using the fuzzing tool by feeding the inputs automatically. Besides, the interface we designed made the fuzzing tool more user-friendly by using an interface to interact with the user and visualize the testing process.

In black-box testing, compared to the mutation-based fuzzing tool, our enhanced grammar-based fuzzing tool penetrated deeper into benchmarks' parsers. The grammar-based fuzzing tool generates less inputs that would be rejected, and found more exceptions when feeding the same number of inputs to the benchmarks.

4.2 Future work

The crashes we found in this project were all triggered by the segment fault, which was because that lack of in-depth knowledge of program behaviour limited the kinds of tests that can be generated. Thus, those different inputs we found exposed the same errors. We planned to enrich the grammar we designed to reduce the possibility of generating duplicated inputs.

Furthermore, we planned to analyze the reason why `bc` crashed on those inputs. Then we would contract with the developer of `bc` and contribute to fixing the code to prevent those crashes.

References

[1] James Fell. A review of fuzzing tools and methods. Technical report, Technical Report. https://dl. packetstormsecurity. net/papers/general/a …, 2017.

[2] T. Guo, P. Zhang, X. Wang, and Q. Wei. Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation. In *2013 Second International Conference on Informatics Applications (ICIA)*, pages 212–215, 2013.

[3] S. Sargsyan, S. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan, and S. Asryan. Grammar-based fuzzing. pages 32–35, 2018.

[4] Scott Seal. *Optimzing Web Application Fuzzing with Genetic Algorithms and Language Theory*. PhD thesis, Wake Forest University, 2016.

[5] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594, 2017.

[6] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. Saarland University, 2020. Retrieved 2020-01-21 11:54:50+01:00.