Compiler Fuzzing via Guided Value Mutation - Final Report

PIUS KRIEMLER* and JONAS DEGELO*, ETH Zürich, Switzerland

Compilers and compiler-like programs serve as critical components for ensuring the safety, functionality, and speed of almost all other software in existence. Therefore rigorous validation and testing of compilers are crucial. In this report we present a new approach for compiler testing, making use of multiple well-established software testing techniques such as fuzzing, mutation testing, and regression testing. We developed a multithreaded, easily extensible program for generating new, valid C programs by mutating constant values in existing C programs. By comparing the resulting assembly instructions across different compiler versions, we aim to identify mutants that exhibit a substantial increase in instruction count when compiled with newer compiler versions as compared to older ones. Such discrepancies in number of instructions can serve as potential indicators of bugs or vulnerabilities within the compiler.

ACM Reference Format:
Pius Kriemler and Jonas Degelo. 2023. Compiler Fuzzing via Guided Value Mutation - Final Report. In . ACM, New York, NY, USA, 7 pages.

1 INTRODUCTION

Compilers are essential to our digital world. They build the bridge between human readable programming languages and machine executable binaries. Therefore compiler bugs can affect the safety, correctness, performance, and functionality of all other software. Compilers are also extremely complex programs programs in their own right. They have a variety of goals. Firstly they need to produce correct machine instructions that correspond to the given source code. Secondly they should produce machine instructions that run as fast as possible. Both of these things are difficult to test and verify. To further complicate things, compilers often need to work for a variety of different hardware configurations. All this is to say that compiler testing is equally important as it is difficult.

One of the main challenges in compiler testing is finding valid input programs. For a test case to be valid, it must comply with the language standard. Mutation has been successfully used to generate new test, valid cases for compiler testing [7, 9]. We explore a new way of generating test cases for C compilers by the means of mutation. Our approach is to mutate constant values (literals) in existing test suites. We evaluate our mutated test cases by compiling them with different compiler versions and comparing the number of assembly instructions produced. If a new compiler version produces significantly more instructions for the same piece of code, this can be an indicator for the presence of a bug.

Throughout this project, we have created a well-documented, easily extendable, and open-source multithreaded Fuzzer [2].

2 RELATED WORK

We have already established the importance of compilers. Due to this, compiler testing is and has been subject to extensive research. Compiler testing via mutation of existing test inputs has been done in various ways. Equivalence Modulo Inputs (EMI) [6] have been used as a basis for multiple mutation approaches in compiler testing [7, 9]. Le et. al [7] both deletes and inserts dead code to mutate seed programs while keeping semantic equivalence. Sun et. al [9]

*Both authors contributed equally to this research.

further build on this approach by also mutating live code sections by careful semantic analysis of program parts and replacing them with semantically equivalent parts. Both of these approaches mutate significantly greater parts of code and are significantly more involved than the approach presented in this report.

As the name implies, with EMI, mutated programs are semantically equivalent to the seed programs for some inputs. In our approach we have no such constraints and can thus mutate more freely. This has an obvious benefit in terms of speed of mutation. But it also has the drawback that verifying the compiled program is a lot more difficult as the expected behaviour of the compiled program is not known. We face this issue by only comparing the disassembled assembly instructions when compiling the mutated program with different compiler versions. This is similar to Yang et. al [10] which use multiple, different compilers as testing oracles.

On a more general level, not specific to compilers, fuzzing, i.e. the rapid generation of test inputs is a popular technique for testing software and catching bugs. Two of the most commonly used fuzzing tools are AFL [11] and its more popular fork AFL++ [5]. They are both designed to test programs against abnormal inputs which might crash the program being tested. Both, AFL style fuzzing and our guided value mutation, rapidly generate test inputs, but they differ in the type of bugs they try to catch. In this project we are specifically only interested in inputs that are valide and that compilers can handle, we do not want to find inputs which cause compilers to crash.

3 APPROACH

Our work focused on the C language and the versions 11 and 12 of the GCC compiler. The compilers used are parameterized in our program and could easily be switched and we have also tried our tool with some of them, specifically we compared Clang-14 with Clang-15 and GCC-10 with GCC-12 in addition to extensive testing with GCC-11 and GCC-12. The general approach for compiler fuzzing presented in this report could also be used for testing compilers for different programming languages. We are interested in differences in the assembly instructions generated by compilers. While it is expected that newer compiler versions produce different machine instructions compared to older versions, it is also expected that the newer versions do not introduce significantly more complexity. In order to check if the output of a newer version is more complex, we use the number of assembly instructions as a proxy for the complexity. While this is not a perfect assumption, e.g. consider loop unrolling, we expect it to perform reasonably well.

Our strategy to find source files which have large differences in the number of assembly instructions across compiler versions is a form of mutation testing. Compilers can do a lot with constant values in source files. For example they can pre-compute computations involving multiple constants, unroll loops, or ignore conditional branches with constants in their conditions. Therefore we focus on constants, also called literals, that occur in source files. In a first approach we randomly mutate these values uniformly at random within different ranges. In a second approach we explore a different strategy to sample the values for the mutants and to improve its performance.

Performance of a mutation strategy has multiple meanings. Firstly we are only interested in runnable programs that compile. We also do not want any undefined behavior in our mutations. If a mutation compiles, is executable, and does not have any undefined behavior, we call it a valid mutation. Due to the way we check for undefined behavior, the mutation must also terminate within an adjustable timeout range in order to be considered valid. The second notion of of performance metric for a strategy is how many differences are found. This question is far more difficult. Finding only valid mutations could be achieved by very careful analysis of the code. But guiding the mutation to find large differences is hard for a variety of reasons. For one, we consider it to be an indicator for bugs, and bugs are by nature rare and difficult to find. Secondly we can consider our problem as a mathematical function. The function takes a source file and two compilers as input. It then computes the difference in number of assembly instructions. This describes a large and expensive black box function. Further the output is a discontinuous step function.

The approach presented here can be seen as a function, which takes as input a seed file and outputs the difference in number of assembly instructions. Our goal is to optimize the function, such that for a given input it finds a difference that is as large as possible. It is obvious that our function is really expensive to compute and without a lot of knowledge about the intricacies of the compiler versions being compared, it is also a black box. We initially planned to use existing optimization algorithms for expensive black box functions, such as Bayesian Optimization [8], but this proofed to be unfeasible due to a significant lack of feedback from our function. We came to notice that usually there either is a difference or there is not and that once there is a difference found, finding an even larger difference is very difficult. In general it does not appear to be the case that once there is a difference we can continue to mutate this file to get a larger difference. For this reason we ended up focusing our efforts to guide the mutation on reducing the number of invalid mutations. Indirectly this also improves the chances of finding differences, since we are not interested in checking the difference of invalid files.

As a starting point for our mutations we used existing test suites of C code. We refer to the files in these test suites as seed files. The main test suite we used was the *c-testsuite* [1]. We performed our benchmarking on this test suite. We also used the official GCC testsuite [3], specifically the gcc.c-torture/execute test cases. This specific part of the testsuite was chosen because it specifically fulfills our own requirements of the programs needing to be executable, i.e. they must have a main method. The gcc test suite was much larger which made it infeasible to repeatedly fuzz it, but we were interested in how many more results we would find with it.

4 IMPLEMENTATION

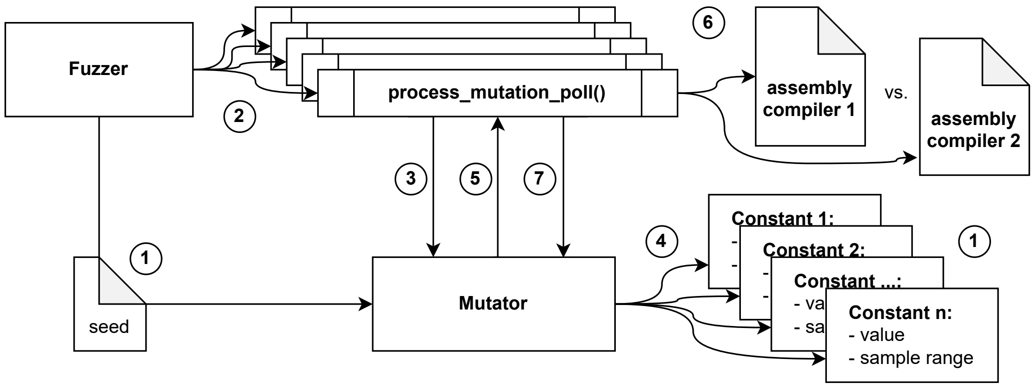


Fig. 1. Fuzzing process overview

① The algorithm is initiated by the **Fuzzer**, which starts by preprocessing a set of source C-files to generate the initial seed files. This preprocessing step is necessary because we are utilizing pycparser [4], which specifically requires preprocessed files as input. pycparser is a tool that can parse a preprocessed C-file into an Abstract Syntax Tree (AST). ④ The AST allows for manipulation, specifically the modification of all constants within it. The **Mutator** class takes care of these steps, including writing the manipulated AST back to a C-file, thus creating a mutation. It also tracks the progress of currently running and completed mutations, ensuring adherence to the termination criteria, which is defined as either reaching maximum number of generated mutations, no matter their validity, or achieving a target number of valid mutations. ② To speed-up the fuzzing process, multiple **worker threads** are spawned independently to carry out the next steps. ③ Each worker thread initiates by requesting a new mutation from the Mutator. ⑤ The Mutator provides the worker thread with the path to the newly generated mutation. A worker thread verifies the validity of the mutation. It compiles and runs the mutation using multiple sanitizer flags. An invalid file is characterized by exhibiting undefined behavior, address errors, divide by zero operations, or failing to return when the mutation is executed. ⑥ Upon successfully passing the validity check, the worker thread proceeds to compile and disassemble the mutation using two different compiler versions. ⑦ The resulting assembly lines are analyzed for differences, and any other relevant information gained from the mutation, such as produced errors or other output, is reported back to the Mutator.

This completes the handling of one mutation. The process then repeats for subsequent mutations, with each worker thread requesting a new mutation and performing the necessary checks and analysis, until the termination criteria is met.

4.1 Mutation Generation and Strategy

The time restrictions of this project didn't allow us to guide the mutation process any further, but we designed and built our **Fuzzer** with this clear goal in mind. We want to quickly elaborate on how an extension of the functionality can be achieved.

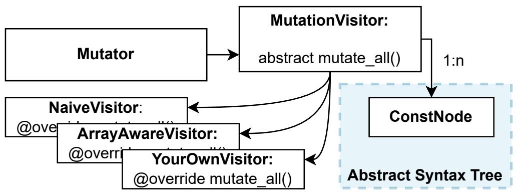


Fig. 2. Parsing overview

One of the primary design choices in our implementation is the selection of the parser. To achieve this, we created a class called **MutationVisitor**, which utilizes pycparser to construct the Abstract Syntax Tree (AST). During the parsing process, we identify and store references to interesting nodes and encapsulate them within a wrapper class called **ConstNode**. The **ConstNode** wrapper class enables each node to handle its value and the range of values that it can be mutated to. This provides the necessary functionality to parse and manipulate the AST effectively. To introduce a new mutation strategy, one simply needs to implement a new **Visitor** that inherits from the **MutationVisitor** class. By overriding the **mutate_all()** function with the specific logic for manipulating constant nodes in the new strategy, the implementation of the strategy is already completed. To enable the execution of the new strategy, it is sufficient to add the strategy as an option within the **Mutator.initialize()** function, assigning it a unique name. This allows the strategy to be executed by specifying the *−mutation-strategy* flag with the respective name during the next run. We provide two mutation strategies in the current version:

(1) random: every constant is mutated within it's given bounds uniformly at random
(2) array-aware: constants that are used to access an array are mutated within the bounds given by the array size. For example:
int a[const1]
a[const2] = const3
const2 ∈ [0, const1]

The second significant design choice revolves around the central role played by the **Mutator**. Instead of each worker thread having its own AST, our approach involves utilizing a single AST that is shared among multiple threads. While this introduces overhead due to the need for thread-safe mutation generation, it provides the advantage of a powerful coordinator. By consolidating all mutation results within the **Mutator**, one could leverage the ability to guide subsequent mutations based on the collective feedback from previous mutations. This coordination enables the possibility of incorporating guided value mutation into the current version of the Fuzzer. To support this extension, modifications would need to be made within the **Mutator** class, specifically within the **report_mutation_result()** function. This design choice, although adding complexity to the code and runtime, offers the advantage of leveraging the collective knowledge and feedback from mutations to guide and enhance the mutation process.

5 RESULTS

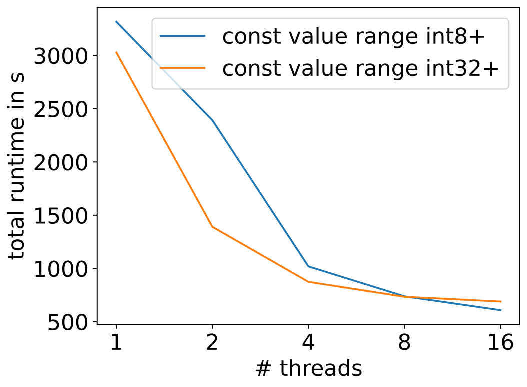
5.1 Performance

We conducted an evaluation of the Fuzzer's speed by varying the sample ranges and thread counts on two different machines, a MacBook Pro equipped with an M1 processor and a Lenovo Thinkpad with an Intel Core i7 1165G7 processor. The seed files for all performance tests came from the c-testuite. The results are depicted in Figure 3a. It is worth noting that there is a slight discrepancy in the runtimes. Specifically, when using a smaller value range, the Fuzzer exhibits approximately 10% slower performance with a small number of threads. This difference can be attributed to the fact that sampling from the int8+ value range resulted in a higher number of valid mutants, as seen in Figure 4a. Each valid mutant undergoes two additional compilation checks to assess the differences in the assembly code. Another possible factor contributing to the observed difference is the inclusion of results from multiple machines, which introduces some degree of variance.

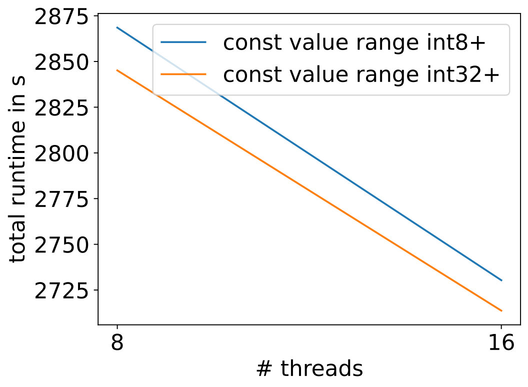
We conducted additional performance tests with an increased number of mutants per seed file for thread counts of 8 and 16. The objective was to assess whether the performance would plateau or if the threading overhead would become too significant when each mutant is responsible for handling 2-4 mutations. The results depicted in Figure 3b indicate that the performance improvement becomes negligible with more than 8 threads. This outcome can be attributed to the fact that both testing machines have 8 logical threads. Thus, adding more threads beyond this point does not yield a substantial increase in performance.

Another performance metric we focused on was the quality of the generated mutants. As depicted in Figure 4a, the number of valid mutants increases when using a smaller sample range. One of the key contributors to this improvement is the substantial reduction in overflows, such as in cases like int a = const1 * const2. We implemented another strategy that samples array accesses exclusively within the array bounds. However, upon evaluation, this strategy did not show any noticeable improvement over the random strategy, as seen in Figure 4b, at least not on this particular test set. The reason for this can be attributed to the bias present in the testsuite we utilized. The c-testuite aims to cover various functionalities, with array-specific seed files having a relatively smaller impact on the occurrence of compile errors and invalid mutations. Further analysis of the results revealed that overflows were the main source of errors in this testsuite. We are confident that on array heavy seed files the strategy performs better than random.

The complete data used for this performance analysis can be seen in the Git repository of this project [2] under *results/data.*

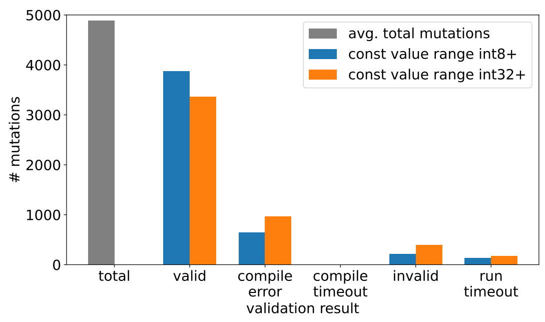


(a) Compare sample ranges int8+ vs. int32+
(random strategy, 32 Mutants per seed file)

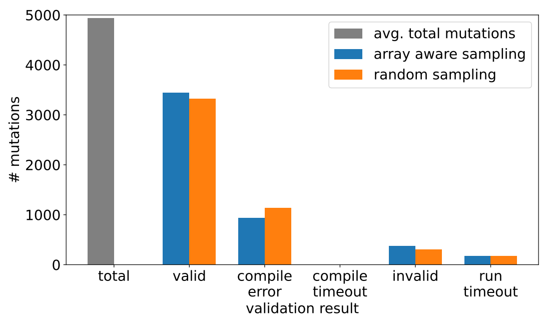


(b) Compare sample ranges int8+ vs. int32+
(random strategy, 128 Mutants per seed file)

Fig. 3. Comparison of the runtime between different sample ranges and strategies



(a) Compare sample range int8+ vs. int32+
(random strategy)



(b) Compare sample strategy random vs. array-aware
(sample range int32+)

Fig. 4. Comparison of mutant quality between different sample ranges and sample strategies.

5.2 Findings

We utilized two testsuites, namely the *c-testsuite* and the *gcc.c-torture/execute*, to obtain a total of 1750 usable seed files. Our Fuzzer was executed on both testsuites using two different machines with distinct hardware configurations. The first machine employed was a MacBook Pro equipped with an M1 processor. During the execution on this machine, our Fuzzer did not uncover any mutations where GCC-12 generated a higher count of assembly instructions compared to GCC-11. The second machine featured an Intel Core i7 1165G7 processor. Using this machine, we observed a significant number of mutations where GCC-12 generated more instructions than GCC-11. Furthermore, we extended our comparison to include GCC-10 and discovered even more substantial differences between GCC-12 and GCC-10. Additionally, we utilized the Intel machine to compare Clang-15 with Clang-14. However, no differences were found between these two compiler versions. For a comprehensive analysis of these runs and the mutations that exhibited varying instruction counts, please refer to our Git repository [2] under the *results/findings* directory. Further examination and analysis of these mutations are required to determine whether the observed differences are attributable to compiler bugs.

6 CONCLUSION

Our project addressed the challenge of compiler testing by focusing on the generation of valid input programs. By leveraging mutation techniques, we explored a novel approach to generating test cases for C compilers. Specifically, we mutated constant values in existing test suites and evaluated the resulting test cases by compiling them with different compiler versions. A significant increase in the number of assembly instructions produced by a new compiler version for the same code snippet could indicate the presence of a bug. Throughout this project, we developed a well-documented, easily extendable, and open-source multithreaded Fuzzer. This Fuzzer provides a valuable tool for compiler testing and demonstrates the importance of continuous testing and improvement in this critical domain. The findings indicate that the Fuzzer demonstrates effective generation of valid mutants, with the runtime being affected by both the sample range and the number of threads employed. However, the threading performance is constrained by the availability of physical cores due to the high CPU-intensive nature of the compilation process. The performance test also reveals deficiencies in adhering to the bounds of individual variables, suggesting the implementation of a strategy to validate variable bounds as a subsequent measure. Additionally, incorporating enhancements like guided value mutation and utilizing collective feedback from mutations holds significant potential to greatly improve the quality of mutants and, consequently, boost the fuzzer's performance. The current implementation allows for these improvements to be efficiently added and tested.

REFERENCES

[1] [n. d.]. c-testsuite. https://github.com/c-testsuite/c-testsuite Accessed: 2023-04-28.

[2] [n. d.]. Compiler Fuzzing via Guided Value Mutation. https://github.com/jnzd/ast-project Accessed: 2023-06-02.

[3] [n. d.]. GCC Testsuite. https://gcc.gnu.org/onlinedocs/gccint/C-Tests.html Accessed: 2023-06-01.

[4] [n. d.]. pycparser. https://github.com/eliben/pycparser Accessed: 2023-04-28.

[5] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*. 10–10.

[6] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices* 49, 6 (June 2014), 216–226. https://doi.org/10.1145/2666356.2594334

[7] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices* 50, 10 (Oct. 2015), 386–399. https://doi.org/10.1145/2858965.2814319

[8] Phuc Luong, Sunil Gupta, Dang Nguyen, Santu Rana, and Svetha Venkatesh. 2019. Bayesian Optimization with Discrete Variables. 473–484. https://doi.org/10.1007/978-3-030-35288-2_38

[9] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 849–863. https://doi.org/10.1145/2983990.2984038

[10] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. *ACM SIGPLAN Notices* 46, 6 (June 2011), 283–294. https://doi.org/10.1145/1993316.1993532

[11] Michał Zalewski. 2016. American Fuzzy Lop-Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt