Finding Small Code with Large Binaries

VIKTOR GSTEIGER and NIKODEM KERNBACH*, ETH Zürich, Switzerland

We propose to implement a framework that, given a source code, reduces the source code size while maximizing the binary size to source code ratio. The framework will implement an iterative approach, that optimizes a heuristic function.

1 MOTIVATION

Compilers are large and complex programs that are a crucial part of the development of software. The goal of a compiler is twofold: (1) produce an executable binary which can be run on a given architecture, and (2) optimize the workflow, such that it requires the least possible amount of operations when being run. Providing correctness for the first and efficiency for the second is a hard task that needs to be solved as compilers are so fundamental to the modern technology stack. Thus generating test programs that trigger specific undefined or unwanted behaviour in compilers is paramount. Automatic tools like Csmith [7] rely on static analysis to generate valid programs and thus are limited with regards to generating all potential valid programs. To generate more programs that trigger undefined behaviour in compilers, one needs to extend the capabilities of such tools.

To expand current research we propose to also investigate non-equivalence relations in compilers such as binary size. It is expected that compilers, when optimizing for binary size, should return a smaller binary after optimization. Failures to do so may show potential unwanted behaviour in compilers and investigations into the code patterns that produce this behaviour may lead to specific conclusions with regard to concrete bugs in compilers. Therefore we propose a framework that, given a source code, reduces the source code size while maximizing the binary size to source code ratio. If successful, this tool might help to specifically show issues with regard to before mentioned non-equivalence relations in compilers.

2 RELATED WORK

As compilers are so ubiquitous in computer systems, significant resources have been deployed to develop tests and verifiers. Recent surveys of the field [1] have shown considerable effort in the direction of compiler testing and verification. The before-mentioned tool Csmith [7] enables the generation of valid code that can be used as a foundation for additional mutations. Surveys into the characteristics of compiler bugs by Sun et al.[6] found that programs that trigger bugs are usually small but complex. Previous research has proposed several approaches for generating mutants with regard to non-equivalence relations. For example, le et al.[2] proposed *Athena* which introduces a guided mutation strategy of compiler input programs that maximise the distance between the mutant and the original file which is partially also based on program sizes which is the non-equivalence relation of interest in our project as well. As this project aims to minimize the source code while maximizing the generated binary, approaches [2] that delete code or approaches [5] mutate code that gets executed are of particular interest.

Authors' address: Viktor Gsteiger, vgsteiger@student.ethz.ch; Nikodem Kernbach, nkernbach@student.ethz.ch, ETH Zürich, Switzerland.

3 APPROACH

The proposed framework needs to conform to the following constraints: (1) the framework should create semantic-preserving mutations of the original, arbitrary, source code, (2) the returned mutant needs to be smaller in size than the source code, (3) the binary size to source code ratio of the mutant needs to be maximised within a reasonable run time compared to the source code.

Our proposal for the framework workflow approach is the following: (1) Generate a random valid code sample. Random to introduce different code patterns. (2) Apply code equivalence transformations which reduce the code size in parallel to get multiple next-round candidates. (3) Filter out samples which do not adhere to the constraints. (4) Evaluate which of the samples are best according to a heuristic and use them as inputs in the next round. (5) Tighten up the heuristic requirements for the next round. (6) Repeat step 1.

The used heuristic function will evaluate the quality (relative to the defined goal) of a specific code sample. It will be based upon the following parameters: (1) Size of the `.text`-section from the binary sample (larger is better). (2) Size of the corresponding code sample (must not be larger than the original, smaller is better). (3) Binary size to source code ratio (larger is better). (4) Large difference to previously generated mutants where we utilize similarity measures (such as proposed by [4] or investigated by [3]) to introduce a penalty if mutants are too similar compared to previously generated mutants.

The framework can then be run for a fixed amount of loop repetitions or until the binary size to source code ratio converges towards a value. In the implementation, we plan to use several tools, such as *creduce*[1]/*cvise*[2] to reduce *C*/*Python* source files, while preserving defined properties, *diopter*[3] to test and clean given programs and *csmith*[4] to generate random code with a specific size.

4 WORK SCHEDULE

Our approach allows for a clear distinction of work and allows us to establish a schedule with enough backup time. We, therefore, propose the following major waypoints for the duration of our project:

Task	Deadline
Implement overall loop based on simple metric	02.04.2023
Combine all metrics into one heuristic function	16.04.2023
Progress report hand-in date	30.04.2023
Optimize heuristic and ensure results are satisfactory	21.05.2023
Final report hand-in date	06.06.2023
Presentation recording hand-in date	10.06.2023

Table 1. Internal Waypoint Deadlines

[1] https://embed.cs.utah.edu/creduce/
[2] https://github.com/marxin/cvise/
[3] https://github.com/DeadCodeProductions/diopter
[4] https://github.com/csmith-project/csmith

REFERENCES

[1] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Comput. Surv.*, 53(1), feb 2020.

[2] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices*, 50(10):386–399, 2015.

[3] Matija Novak, Mike Joy, and Dragutin Kermek. Source-code similarity detection and detection tools used in academia: a systematic review. *ACM Transactions on Computing Education (TOCE)*, 19(3):1–37, 2019.

[4] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. Similarity of source code in the presence of pervasive modifications. In *2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM)*, pages 117–126. IEEE, 2016.

[5] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*, pages 849–863, 2016.

[6] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 294–305, 2016.

[7] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.