Poster: MC/DC Coverage-Oriented Compiler Optimization for Symbolic Execution

Yijun Liu, Zhenbang Chen, Wei Dong, Chendong Feng
College of Computer, National University of Defense Technology
Changsha, China
{zbchen,wdong}@nudt.edu.cn

ABSTRACT

Complier optimizations influence the effectiveness and efficiency of symbolic execution. In this extended abstract, we report our recent results of recommending compiler optimizations for symbolic execution *w.r.t.* MC/DC coverage. We carried out extensive experiments to study the influence of compiler optimizations on MC/DC coverage. Then, an SVM-based optimization recommendation method is designed and implemented. The preliminary experimental results are promising.

1 INTRODUCTION

Symbolic execution [4, 6] provides a general way for systematically exploring the path space of a program. Due to its precision and path-sensitivity, symbolic execution has been be successfully applied to test case generation [2]. Usually, when using symbolic execution to analyze a program, the program will be *compiled* into a binary or an intermediate representation (IR) first. Then, symbolic execution is carried out on the binary or IR. During this process, compiler optimizations may influence the effectiveness and efficiency of symbolic execution [1, 3].

Existing work [1] discusses the impact of compiler optimization on symbolic execution. The influence of compile optimization is empirically studies *w.r.t.* statement coverage and decision coverage in [3]. However, as far as we know, there is no existing work on compiler optimization recommendation for symbolic execution. The feature extraction of a program *w.r.t.* the influence of compile optimization on symbolic execution is challenging. Besides, no existing work studies the impact of compiler optimization on symbolic execution *w.r.t.* MC/DC coverage [5], *i.e.*, an important industrial coverage criterion for safety-critical software systems.

In this extended abstract, we report our preliminary result of studying and recommending compiler optimizations for symbolic execution *w.r.t.* MC/DC coverage. The basic idea is to use empirical study to identify the key optimizations *w.r.t.* MC/DC coverage first. Then, based on the feature extraction *w.r.t.* key optimizations, we use machine learning techniques to train a recommendation model of compiler optimization for symbolic execution. The model can be used before symbolic execution to determine whether to apply the key optimizations, aiming to improve the MC/DC coverage achieved by the test cases produced by symbolic execution.

2 MOTIVATION EXAMPLE

Figure 1 displays an example program to demonstrate the problem. The program is from the example programs provided by KLEE [2], *i.e.*, a state-of-art symbolic executor for C programs.

The function `get_sign` has an input variable `x`, and there are three cases depending on the value of `x`. Clearly, three test cases are enough to test the program. However, if we analyze the program by KLEE using the default configuration in which compiler optimization is turned on, KLEE only generates *two* inputs (*e.g.*, 0 and 10), which result in 50% MC/DC coverage. If we turn off the compiler optimization, KLEE generates *three* inputs, whose execution achieves 100% MC/DC coverages.

```
1  int get_sign(int x) {
2    if (x == 0)
3      return 0;
4    if (x < 0)
5      return -1;
6    else
7      return 1;
8  }
```

Figure 1: An example program.

The reason is KLEE optmizes the LLVM IR of the program before symbolic execution. The second branch statement (Lines 4~7) is optimized into non-branched instructions, *i.e.*, an arithmetic shift right and a bitwise or. When the symbolic execution is carried out on the optimized IR, the path condition of the program is only produced by the condition of the first branch, *i.e.*, `x == 0`. Hence, only two inputs will be generated.

3 EMPIRICAL STUDY

We have the following three research questions to study:

- Dose increasing symbolic execution's time improve MC/DC coverage?
- Do compiler optimizations influence MC/DC coverage?
- Whether exist dominant compiler optimizations *w.r.t.* MC/DC coverage?

Experimental Framework & Benchmark. Figure 2 give the framework of experiments. The inputs of the framework are the programs under testing and the configuration (*e.g.*, which compiler optimizations are applied) of KLEE, and the framework's output is the test report containing MC/DC results. The main process of the framework can be divided into two stages: *test case generation* and *test execution*. In the first stage, KLEE is configured and used to automatically generate the test cases. Test driver generator extracts the input values and generates the driver code according to the test driver templates of C++Test [7]. At the second stage, the test driver code and the program under testing are fed into C++Test to execute the test cases and generate the test report. Inside the framework, the test driver generator is implemented by us. KLEE's version is

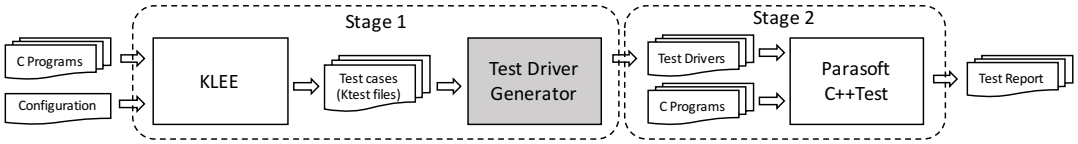

Figure 2: Experimental Framework

1.2, and C++Test's version is 9.6. We use the programs in Coreutils [2] as the studied benchmark. There are 89 Coreutils programs used in [2], and we filter 6 programs (*i.e.*, `kill`, `hostname`, `who`, `chmod`, `ln`, and `du`) because C++Test cannot produce the test report for them or the experiments failed. The numbers of MC/DC conditions insides these programs are well distributed from 3 to 927.

Experimental Results. To study the analysis time's influence on MC/DC coverage, we automatically tested all the 83 coreutils programs in 5, 10, 30 and 60 minutes. Out of 83 programs, there are 56 programs (67.4%) on which MC/DC coverages *do not* change when increasing the time of symbolic execution, *i.e.*, same coverage in 5, 10, 30, and 60 minutes. The MC/DC coverages of only 13 programs (15.3%) increase after 10 minutes. These results indicate *increasing the time of symbolic execution does not always increase the MC/DC coverage effectively.*

To study compiler optimization's influence on MC/DC coverage, we test each program in 5 minutes. We use the MC/DC coverage of not using any compiler optimization (denoted by `No`) as the baseline. The experimental results indicate *compiler optimization may increase or decrease the MC/DC coverage of a program.* Almost every optimization method used by KLEE influences (increase or decrease) at least one program's MC/DC coverage, except `FA`. There are 30 optimization methods that can increase the MC/DC coverage; whereas, only 14 methods can decrease the MC/DC coverage. Many optimization methods only influence a small number of programs, *i.e.*, less than 4 programs. Only six optimization methods (*i.e.*, `IC`, `FI`, `PMTR`, `SRA`, `Internalize` and `LR`) influence more than 10 programs.

Inside six optimizations, we use a two-stage procedure to find dominant ones. First, we identify the ones that are equivalent with or coincident with applying all optimizations (denoted by `ALL`) more. Then, we inspect the difference between `ALL` and `ALL` after disabling a single optimization method. If an optimization method is identified to be more equivalent or coincident with `ALL` at the first stage, and also very different from `ALL` after disabled at the second stage, then the method is considered to be a dominant or key optimization. Finally, `IC` *is identified as the dominant optimization method.* At the first stage, `IC` is the one that is equivalent (50.6%) and coincident (75.9%) with `ALL` most. At the second stage, the equivalence rate of disabling `IC` is 43.37%.

4 OPTIMIZATION RECOMMENDATION

To decide whether to adopt a compiler optimization, recommendation needs the program feature *w.r.t.* the optimization. The feature extraction and synthesis of a program *w.r.t.* compiler optimization methods is challenging. The results in Section 3 indicate `IC` is the dominant compiler optimization method. Based on this result, we propose a program feature extraction method *w.r.t.* `IC`. Then, a recommendation method for `IC` is designed and implemented.

`IC` is an intra-procedural optimization process. Each function in the program is optimized individually. For a function f, the instructions inside f are optimized using different rules if possible until no rule can be applied. Precise feature extraction *w.r.t.* `IC` is theoretically undecidable and not practical. Hence, we propose a lightweight method for feature extraction. The basic idea is to run IC optimization on the program, and records the times of *successful* optimizations on each LLVM instruction type. In total, `IC` handles 43 types of LLVM instructions. Hence, given a program $\mathcal{P}$, its feature *w.r.t.* IC, denoted by $F(\mathcal{P})$, is a vector of 43 dimensions, *i.e.*, $\langle c_1, ..., c_{43} \rangle$, where $c_i \geq 0$ and $1 \leq i \leq 43$, and c_i is the times of successful optimizations on ith instruction type.

We applied the feature extraction method to the 83 Coreutils programs, and used support vector machine (SVM) to train a classifier using the extracted features and the experimental results in Section 3. The trained classifier is then used before symbolic execution to decide whether to apply `IC`.

Experimental results. We implemented and integrated the recommendation method to the framework in Figure 2, and reran the experiments on the 83 Coreutils programs. On 83.13% programs, recommendation method can achieve the maximum MC/DC coverage. The recommendation method increases and decreases the MC/DC coverage on 18 and 13 programs, respectively; the numbers of `ALL` are 18 and 37; the numbers of `disable-IC` are 13 and 16. Hence, the recommendation method outperforms both of them, which indicates the effectiveness of the recommendation method.

5 CONCLUSION AND FUTURE WORK

This abstract reports our recent progress on recommending compiler optimizations for symbolic execution *w.r.t.* MC/DC coverage. Our empirical study indicates that compiler optimizations influence MC/DC coverage, and `IC` is the dominate optimization. Then, a lightweight recommendation method is designed and implemented *w.r.t.* `IC` towards improving MC/DC coverage. The experimental results indicate the method is effective. The next step has two aspects: (1) more extensive experiments on other benchmarks; (2) inspect whether the results are still valid *w.r.t.* other coverage criteria.

REFERENCES

[1] C. Cadar. Targeted program transformations for symbolic execution. In *ESEC/FSE*, pages 906–909, 2015.
[2] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.
[3] S. Dong, O. Olivo, L. Zhang, and S. Khurshid. Studying the influence of standard compiler optimizations on symbolic execution. In *ISSRE*, pages 205–215, 2015.
[4] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.
[5] H. Kelly J., V. Dan S., C. John J., and R. Leanna K. A practical tutorial on modified condition/decision coverage. Technical report, 2001.
[6] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19 (7):385–394, 1976.
[7] Parasoft. Parasoft C/C++test 9.6. https://www.parasoft.com/products/ctest.