Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

# Modified condition/decision coverage (MC/DC) oriented compiler optimization for symbolic execution*

Wei-jiang HONG[1,2], Yi-jun LIU[1], Zhen-bang CHEN[†‡1], Wei DONG[†1], Ji WANG[†1,2]

[1]*College of Computer, National University of Defense Technology, Changsha 410073, China*

[2]*State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha 410073, China*

[†]E-mail: zbchen@nudt.edu.cn; wdong@nudt.edu.cn; wj@nudt.edu.cn

Received Apr. 26, 2019; Revision accepted Aug. 21, 2019; Crosschecked Mar. 16, 2020; Published online June 26, 2020

**Abstract:** Symbolic execution is an effective way of systematically exploring the search space of a program, and is often used for automatic software testing and bug finding. The program to be analyzed is usually compiled into a binary or an intermediate representation, on which symbolic execution is carried out. During this process, compiler optimizations influence the effectiveness and efficiency of symbolic execution. However, to the best of our knowledge, there exists no work on compiler optimization recommendation for symbolic execution with respect to (w.r.t.) modified condition/decision coverage (MC/DC), which is an important testing coverage criterion widely used for mission-critical software. This study describes our use of a state-of-the-art symbolic execution tool to carry out extensive experiments to study the impact of compiler optimizations on symbolic execution w.r.t. MC/DC. The results indicate that instruction combining (IC) optimization is the important and dominant optimization for symbolic execution w.r.t. MC/DC. We designed and implemented a support vector machine based optimization recommendation method w.r.t. IC (denoted as auto). The experiments on two standard benchmarks (Coreutils and NECLA) showed that auto achieves the best MC/DC on 67.47% of Coreutils programs and 78.26% of NECLA programs.

**Key words:** Compiler optimization; Modified condition/decision coverage (MC/DC); Optimization recommendation; Symbolic execution

https://doi.org/10.1631/FITEE.1900213 **CLC number:** TP311.5

## 1 Introduction

Testing is the mainstream method used in software development to improve the quality of software (Beizer, 2003). Software testing is a labor-intensive and time-consuming process. Improving the efficiency and evaluating the effectiveness are challenging. Automatic testing aims to improve the efficiency of testing by different techniques, such as automatic test-case generation (Cadar et al., 2008) and automatic test execution (Fewster and Graham, 2000). On the other hand, many approaches (Chen TY et al., 1998; McKeeman, 1998; Wong WE, 2001) have been proposed to evaluate the effectiveness of testing. For example, in mutation testing (Wong WE, 2001), the evaluation of effectiveness is related to the number of mutants that the test-case set can kill. The more mutants are killed, the more effective the testing is. Program coverage is an important way of evaluating the effectiveness of testing (Ammann and Offutt, 2016). The higher the coverage achieved, the more effective the

‡ Corresponding author

* Project supported by the National Key R&D Program of China (No. 2017YFB1001802) and the National Natural Science Foundation of China (Nos. 61472440, 61632015, 61690203, and 61532007)

ORCID: Wei-jiang HONG, https://orcid.org/0000-0002-7092-3658; Zhen-bang CHEN, https://orcid.org/0000-0003-0874-3231

testing is. Until now, existing program coverage criteria include statement coverage, branch coverage, decision coverage, and modified condition/decision coverage (MC/DC) (Zhu et al., 1997; Hayhurst and Veerhusen, 2001; Jia Y and Harman, 2011; Su et al., 2017).

Symbolic execution (King, 1976; Godefroid et al., 2005) provides a general way for systematically exploring the search space of a program and has been widely used for automatic test-case generation (Cadar et al., 2008; Godefroid et al., 2008), such as KLEE (Cadar et al., 2008), Pex (Tillmann and de Halleux, 2008), and CUTE (Sen et al., 2005). Compared with random testing (Duran and Ntafos, 1984), symbolic execution based automatic testing can generate test cases to achieve higher coverage quickly. In addition to test case generation, symbolic execution can automatically find subtle bugs in the tested programs (Cadar et al., 2008).

Usually, when using a symbolic executor to analyze a program, the program will be compiled into a binary or an intermediate representation (IR) first. Then, symbolic execution is carried out by running the binary representation or IR. During this process, compiler optimization influences the effectiveness and efficiency of symbolic execution (Cadar, 2015; Dong et al., 2015). For example, when we use symbolic execution for test-case generation, compiler optimization may simplify the path condition and further improve the efficiency of the satisfiability modulo theories (SMT) solver (de Moura and Bjørner, 2008), but it may also decrease the coverage, due to the simplification in control flows and the reduced instruction caused by different optimizations.

Existing work (Cadar, 2015) discussed the impact of compiler optimization on symbolic execution. The influence of compiler optimization was empirically studied in Dong et al. (2015) with respect to (w.r.t.) statement and decision coverage. However, to the best of our knowledge, there exists no work that studies the impact of compiler optimization on symbolic execution w.r.t. MC/DC (Hayhurst and Veerhusen, 2001), which is an important industrial coverage criterion for safety-critical software systems. Compared with statement, branch, and decision coverages, MC/DC measures the adequacy of test cases more strictly. For example, DO-178B (EUROCAE, 1998), which is an industrial avionics software development standard, requires use of MC/DC to measure the testing of the most critical (level A) software. In addition, MC/DC was used for satellite software in Dupuy and Leveson (2000), and the test cases that satisfy the MC/DC requirement can detect errors that could not be found by employing other coverage criteria. Therefore, considering symbolic execution as an important method for generating test cases, it is worth investigating the factors that influence MC/DC for symbolic execution. Although compiler optimization has a particular influence on the effectiveness and efficiency of symbolic execution, there are few works on compiler optimization recommendation for symbolic execution, because the feature extraction of a program w.r.t. the influence of compiler optimization on symbolic execution is challenging.

In this study, we propose an MC/DC oriented compiler optimization recommendation method for symbolic execution. The basic idea is to conduct an empirical study to identify the critical compiler optimizations w.r.t. coverage criteria first. Then, based on the feature extraction w.r.t. the key compiler optimizations, we use data mining techniques to train a recommendation model for compiler optimizations. The model can be used before symbolic execution to determine whether to apply the critical compiler optimizations, and aims to improve the coverages achieved by the test cases generated by symbolic execution.

The main contributions of this study are as follows:

1. We propose a general framework for the empirical influence study of compiler optimizations on symbolic execution w.r.t. a testing coverage criterion.

2. Based on this framework, comprehensive empirical studies are carried out on standard benchmarks and a state-of-the-art symbolic execution engine, i.e., KLEE, w.r.t. statement coverage, decision coverage, and MC/DC. The empirical studies indicate that instruction combining (IC) optimization (Lopes et al., 2015) is the important and dominant component influencing the effectiveness and efficiency of symbolic execution w.r.t. the three coverage criteria.

3. We propose a method to extract the program features w.r.t. IC optimization, based on which an optimization recommendation method for symbolic execution is proposed to improve the coverages. The

experimental results indicate that the recommendation method is effective.

## 2 Background and the motivating example

This section first provides a brief introduction to symbolic execution, MC/DC, and compiler optimization. Then, a motivating example is used to demonstrate the research problem.

### 2.1 Symbolic execution

Symbolic execution is an effective technique for exploring the program's search space. To symbolically execute a program, the inputs of the program are initialized as symbols, such as $x$ and $y$, which represent all the possible values of the inputs; then the symbolic inputs are used to execute the program based on the program's semantics. For each path in the program, symbolic execution obtains a path condition corresponding to this path. A path condition is the conjunction of the mathematical expressions about the input symbols, such as $(x \leq 1) \wedge (x + y \geq 2)$. At the beginning, path condition (PC) is True. A path's PC is calculated as follows: if the current path condition is PC while meeting a branch statement whose condition is $c$, then symbolic execution can derive two path conditions from PC, i.e., $\mathrm{PC} = \mathrm{PC} \wedge c$ and $\mathrm{PC} = \mathrm{PC} \wedge \neg c$, which correspond to the true and false branches of the statement, respectively. After that, symbolic execution adopts an SMT solver to check the satisfiability of each path condition. If the path condition is satisfiable (de Moura and Bjørner, 2011), i.e., the corresponding branch is feasible, symbolic execution will enter the branch to continue; otherwise, symbolic execution abandons the branch due to its infeasibility. In this way, symbolic execution can be used to automatically generate test cases for a program, i.e., generate a test case for each path in the program via the path's PC. Compared with random testing (Duran and Ntafos, 1984), symbolic execution can be more efficient in achieving a higher code coverage. There have been many successful symbolic execution based automatic testing tools, including KLEE, Pex, and CUTE.

### 2.2 Modified condition/decision coverage

MC/DC is an important industrial coverage criterion for safety-critical software systems. The cores of this coverage are: (1) All possible outcomes of the conditions and the decision are exploited; (2) Each condition in the decision can independently affect the outcome of the decision at least once.

For example, the decision in Fig. 1 is "$x > 0$ && $y > 0$," which contains two conditions. We can use the three test cases shown in Table 1 to achieve full MC/DC: test cases 1 and 2 exploit all possible outcomes of $x > 0$, which independently affects the outcome of the decision, and the same as $y > 0$ w.r.t. test cases 1 and 3.

```
int main (int x, int y) {
  if (x>0 && y>0)
    return 1;
  else
    return -1;
}
```

**Fig. 1 An MC/DC example program**

**Table 1 Test cases for MC/DC**

| Test case | $x > 0$ | $y > 0$ | Decision | Input |
|---|---|---|---|---|
| 1 | True | True | True | (1, 1) |
| 2 | False | True | False | (−1, 1) |
| 3 | True | False | False | (1, −1) |

### 2.3 Compiler optimization

Compiler optimization is generally used to transform the source program into a semantically equivalently optimized program. The rational use of compiler optimizations can effectively minimize the time taken to execute a program and the amount of memory needed to run the program (Aho et al., 1986). Modern compilers, such as GCC (https://gcc.gnu.org/) and LLVM (Lattner and Adve, 2004), employ many optimizations before generating binaries. KLEE (https://klee.github.io/tutorials/testing-function/) is a symbolic virtual machine built on top of the LLVM compiler infrastructure. KLEE carries out symbolic execution on the LLVM IR of the program. Before starting symbolic execution, KLEE employs compiler optimizations that are provided by LLVM to optimize the LLVM IR of the program to improve the efficiency of symbolic execution. The compiler optimizations using

KLEE include IC, loop rotation (LR), and constant merge (CM).

### 2.4 Motivating example

Fig. 2 displays an example program to demonstrate the research problem. The program is from the example programs provided by KLEE.

```
int get_sign (int x) {
  if (x==0)
    return 0;
  if (x<0)
    return −1;
  else
    return 1;
}
```

**Fig. 2 An example program**

The function `get_sign` has an input variable $x$, and there are three cases depending on the value of $x$. There are three paths in the function. However, if we analyze the program by KLEE using the default configuration in which compiler optimization is turned on, KLEE generates only two inputs (e.g., 0 and 10), which result in 80% statement coverage, 75% condition coverage, and 50% MC/DC, respectively. If we turn off the compiler optimization, KLEE generates three inputs, whose execution achieves full statement coverage, condition coverage, and MC/DC. The reason is that KLEE optimizes the LLVM IR of the program before symbolic execution.

Fig. 3 displays the IR of the second branch statement after optimization, where $r$ is the return value. The compiler optimization optimizes the second branch statement into two instructions, i.e., an arithmetic shift right and a bitwise or. When the symbolic execution is carried out on the optimized IR, only the first branch's condition, i.e., $x == 0$, produces path conditions. Hence, KLEE explores only two paths, resulting in two test cases. As demonstrated by this example, although the efficiency of symbolic execution may be improved by compiler optimization due to a reduction of a program's search space, the achieved coverages of the tests produced by symbolic execution may also be reduced.

```
y = ashr x, 31;
r = y or 1;
```

**Fig. 3 IR of the optimized second branch statement**

## 3 Empirical study

This section begins by stating our research questions. Then, we describe our design of the experiment and its framework to answer the research questions. After that, we show and discuss the experimental results. Finally, we discuss the threats to the validity of this experiment.

### 3.1 Research questions

The influence of compiler optimization on statement and decision coverages has already been studied in Dong et al. (2015). In our study, we are concerned with MC/DC. We have the following research questions to answer by empirical studies:

RQ1: Does increasing the time of symbolic execution improve the program's MC/DC?

RQ2: Does the compiler optimization influence the program's MC/DC?

RQ3: Do dominant compiler optimizations exist w.r.t. MC/DC?

RQ4: Are the results of the aforementioned three questions still valid w.r.t. statement or decision coverage?

### 3.2 Experimental framework

Fig. 4 gives the framework of the experiments. The inputs of the framework are the programs under test and the configuration of the symbolic executor KLEE, and the framework's output is the test report, which contains the different kinds of coverage information. The primary process of the framework can be divided into two stages, in which test-case generation and test execution are carried out, respectively. In the first stage, KLEE is configured and used to systematically explore the search space of the program under test. For each path, KLEE generates a test case in a Ktest file containing the program inputs. Then, each Ktest file is input into the test driver generator to extract the input values of the program and generate the driver code according to the test driver templates of Parasoft C/C++test (https://www.parasoft.com/products/ctest). In the second stage, the test driver code and the program under test are fed into Parasoft C/C++test to execute the test cases and generate the test report in HMTL format. The coverage information in the current report includes statement coverage, condition coverage, and MC/DC. Finally, we extract the

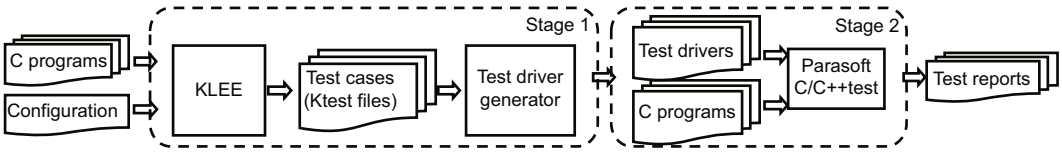


**Fig. 4 Experimental framework**

coverage values in the HTML report (this step is omitted in Fig. 4 due to the space limitation). The whole process of the framework is automated to facilitate the experiments. Inside the framework, we implement the test driver generator and the coverage information extractor. The versions of LLVM, KLEE, and Parasoft C/C++test are 3.4, 1.2, and 9.6, respectively. Our experiments are carried out on a server with eight cores and 16 GB memory, and the OS is Ubuntu Linux 14.04 in a 64-bit architecture.

We use the programs in Coreutils (https://www.gnu.org/software/coreutils/coreutils.html) as the studied benchmark. KLEE originally uses Coreutils as the main benchmark, and Coreutils is later widely adopted to evaluate the symbolic execution techniques (Wong E et al., 2015) that do the implementations based on KLEE. Coreutils contains Unix utility programs in which many diverse operations exist, such as file operations and system invocations. There are 89 Coreutils programs used in the original KLEE paper (Cadar et al., 2008). We filter six programs (i.e., kill, hostname, who, chmod, ln, and du), because Parasoft C/C++test cannot produce the test reports for them due to abnormal exit during testing. Table 2 shows the 83 programs in our experiments and their numbers of MC/DC conditions. As shown in Table 2, the number of MC/DC conditions is well distributed; the smallest is three and the largest is 927, which indicates how representative the benchmark is.

The KLEE configuration contains the following items:

1. Whether to do compiler optimization before symbolic execution. The `--optimize` parameter of KLEE can enable or disable all the default optimizations carried out by KLEE before symbolic execution. Also, we modify the KLEE optimization module to control a specific optimization item.

2. The analysis time of symbolic execution, which can be controlled by the `--max-time` parameter. KLEE analyzes each program at 5, 10, 30, and 60 min. The search strategy is depth-first search (DFS).

3. Other configurations are the same as those in KLEE's original paper for testing Coreutils (https://klee.github.io/docs/coreutils-experiments/). Specifically, we use the default cache-based query optimizations (i.e., `-use-cache` and `-use-cex-cache`) provided by KLEE.

### 3.3 Experimental results

#### 3.3.1 Influence of analysis time

In real-world programs, there often exist branches or conditions that are difficult for the symbolic executor to explore in a limited time, e.g., complex non-linear conditions and program statements.

**Table 2 Eighty-three Coreutils programs in this study**

| $P$ | $\#c$ | $P$ | $\#c$ | $P$ | $\#c$ |
|---|---|---|---|---|---|
| base64 | 108 | id | 53 | setuidgid | 21 |
| basename | 11 | join | 259 | shred | 207 |
| cat | 136 | link | 6 | shuf | 130 |
| chcon | 134 | logname | 4 | sleep | 11 |
| chgrp | 22 | ls | 876 | sort | 668 |
| chown | 21 | md5sum | 220 | split | 91 |
| chroot | 8 | mkdir | 13 | stat | 58 |
| cksum | 90 | mkfifo | 10 | stty | 927 |
| comm | 61 | mknod | 24 | sum | 75 |
| cp | 165 | mktemp | 29 | sync | 3 |
| csplit | 171 | mv | 52 | tac | 130 |
| cut | 209 | nice | 23 | tail | 377 |
| date | 70 | nl | 163 | tee | 76 |
| dd | 230 | nohup | 30 | touch | 84 |
| df | 368 | od | 32 | tr | 256 |
| dircolors | 177 | paste | 118 | TRUE | 47 |
| dirname | 11 | pathchk | 56 | tsort | 219 |
| echo | 50 | pinky | 92 | tty | 6 |
| env | 59 | pr | 379 | uname | 24 |
| expand | 101 | printenv | 14 | unexpand | 128 |
| expr | 178 | printf | 121 | uniq | 131 |
| factor | 22 | ptx | 339 | unlink | 5 |
| FALSE | 47 | pwd | 28 | uptime | 27 |
| fmt | 145 | readlink | 8 | users | 32 |
| fold | 126 | rm | 23 | wc | 225 |
| ginstall | 93 | rmdir | 25 | whoami | 4 |
| head | 259 | runcon | 34 | yes | 7 |
| hostid | 3 | seq | 128 | | |

$P$: program; $\#c$: MC/DC condition

To improve the coverage, a common method is to increase the time of test case generation to produce more test cases.

To study the analysis time's influence on MC/DC, we automatically test all the 83 Coreutils programs at 5, 10, 30, and 60 min of execution time, where we disable the compiler optimization used by the symbolic executor. Out of 83 programs, there are 56 programs (67.5%) on which MC/DC does not change when increasing the time of symbolic execution, i.e., the same coverage occurs at 5, 10, 30, and 60 min. Within these 56 programs, only two programs achieve full MC/DC in 5 min. For the other 27 programs, MC/DC increases, and there are 14 programs whose MC/DC increases using 10 min, 7 using 30 min (but the same for 5 and 10 min), and 6 using 60 min (but the same for 5, 10, and 30 min). These results indicate that increasing the time of symbolic execution does not always increase MC/DC effectively. MC/DC of only 13 programs (15.7%) increases after 10 min, which is a reasonably long time for test-case generation. Hence, we have the answer to RQ1: Increasing the time of symbolic execution does not always effectively increase MD/DC.

This result also indicates that it is important to investigate more efficient methods of path exploration in symbolic execution w.r.t. MC/DC. In addition, the result indicates that 5 min is enough to inspect the compiler optimization's influence on MC/DC for the Coreutils programs.

### 3.3.2 Influence of compiler optimizations

To study compiler optimization's influence on MC/DC, we test each program for 5 min. We use MC/DC of not using any compiler optimization (denoted as NO) as the baseline. We enable only one compiler optimization method and compare the resulting MC/DC with the baseline. Instead of focusing on the detailed coverage value of each program, here we are more concerned about the extent of coverage change on each program and the number of programs influenced after applying the compiler optimization method. If the coverage of a program is increased or decreased by at least 10% w.r.t. NO when using an optimization method, we count the program as being influenced w.r.t. the optimization method. Fig. 5 displays the result. The $x$ axis shows the abbreviation of each optimization method, where ALL represents using all the optimizations. The $y$ axis shows the number of affected programs (i.e., decreased and increased ones) for each optimization method.

As indicated in Fig. 5, compiler optimization may decrease or increase the MC/DC of a program. Almost every optimization method used by KLEE influences at least one program's MC/DC, except Function Attrs (FA). However, many optimization methods influence only a small number of programs, i.e., fewer than four programs. There exist only six optimization methods that influence more than 10 programs, i.e., IC, Function Inlining (FI), Promote Memory To Register (PMTR), Scalar Replacement of Aggregates (SRA), Internalize, and Loop Rotate (LR). The functionalities of these compiler optimizations (https://llvm.org/docs/Passes.html) are as follows:

IC: combining instructions to form fewer or simpler instructions;

FI: making the functions inline in a bottom-up manner w.r.t. the callees;

PMTR: transferring memory references to register references, which reduces memory load and stores instructions;

SRA: replacing the aggregate type's `alloca` instructions with the individual `alloca` instructions for each member in the aggregate type;

Internalize: making all the global variables with initializers internal if there exists a main function;

LR: rotating a loop, e.g., moving the loop control statement before the loop body to the place after the loop body.

The number of programs influenced using all the compiler optimizations (denoted by ALL) is the largest; i.e., 37 decreased and 18 increased. There are 30 optimization methods that can increase the MC/DC, whereas only 14 optimization methods can decrease the MC/DC. In addition, the number of programs with decreased MC/DC using ALL is close to that of IC, and the number of the programs with increased MC/DC using ALL is smaller than that of FI. Hence, this indicates that the influences of different optimization methods cannot be accumulated, and that different optimization methods may influence each other. Therefore, we have the answer to RQ2: Compiler optimization may increase or decrease the MC/DC of a program.

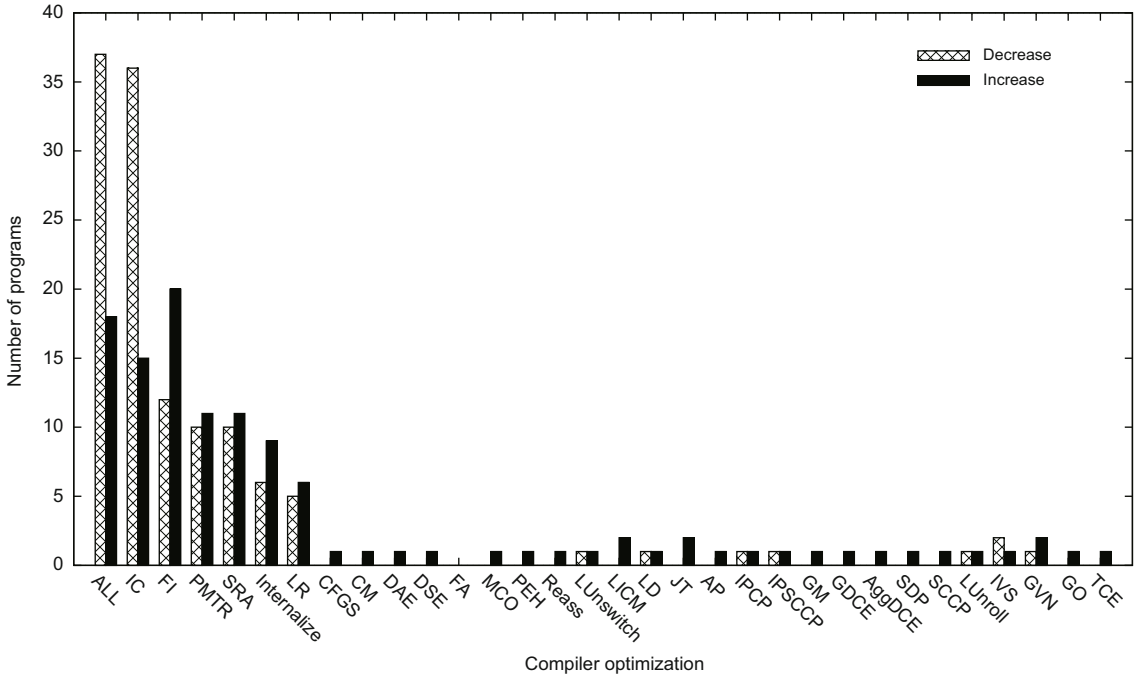


**Fig. 5 Number of influenced programs w.r.t. MC/DC**

ALL: using all the optimizations; IC: Instruction Combining; FI: Function Inlining; PMTR: Promote Memory To Register; SRA: Scalar Replacement of Aggregates; LR: Loop Rotate; CFGS: Control Flow Graph Simplification; CM: Constant Merge; DAE: Dead Argument Elimination; DSE: Dead Store Elimination; FA: Function Attributes; MCO: MemCpy Optimization; PEH: Prune Exception Handling; Reass: Reassociate; LUnswitch: Loop Unswitch; LICM: Loop Invariant Code Motion; LD: Loop Deletion; JT: Jump Threading; AP: Argument Promotion; IPCP: InterProcedural Constant Propagation; IPSCCP: InterProcedural Sparse Conditional Constant Propagation; GM: Globals Mod/ref; GDCE: Global Dead Code Elimination; AggDCE: Aggressive Dead Code Elimination; SDP: Strip Dead Prototypes; LUnroll: Loop Unroll; IVS: Induction Variables Simplify; GVN: Global Value Numbering; GO: Global Optimizer; TCE: Tail Call Elimination

### 3.3.3 Dominant compiler optimizations

According to the results in Fig. 5, there are six optimization methods that mainly influence MC/DC. We want to study whether there exist key or dominant ones inside these six for MC/DC. To evaluate that, we give some evaluation criteria. Two optimization methods coincide on a program if they both increase or decrease the MC/DC of the program. Two optimization methods are equivalent on a program if they coincide on the program and the difference between the MC/DC using the two methods is less than 10%.

The basic idea of finding dominant optimization methods is a two-stage procedure. First, we identify the ones that are more equivalent to or coincident with ALL. 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 to or coincident with ALL at the first stage, and also very different from ALL after being disabled at the second stage, then the method is considered to be a dominant or essential optimization for MC/DC.

At the first stage, because the number of the programs influenced by IC is close to that of ALL, we first inspect the rates of coincidence and equivalence of IC with ALL. Fig. 6 displays the MC/DC of NO, ALL, and IC. The $x$ axis displays the program index, and the $y$ axis shows MC/DC. The programs

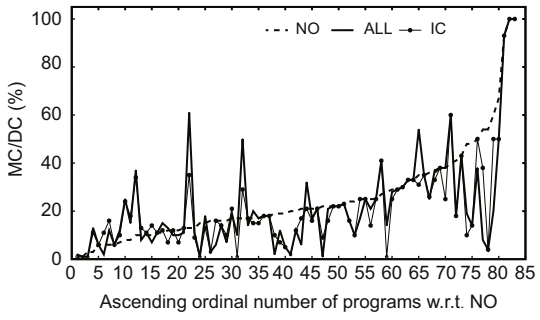


**Fig. 6 MC/DC of NO, ALL, and IC**

are ordered by the MC/DC of NO. Compared with NO, ALL may decrease or increase MC/DC on some programs, and the trend of IC is coincident with that of ALL on most programs. Furthermore, Fig. 7 shows the rates of coincidence and equivalence for the six optimization methods with ALL. As shown by Fig. 7, IC is the one that is most equivalent to (50.6%) and coincident with (75.9%) ALL. The coincidence rate of IC is at least 20% higher than those of the remaining five methods. Hence, IC is likely to be the dominant optimization method for KLEE.

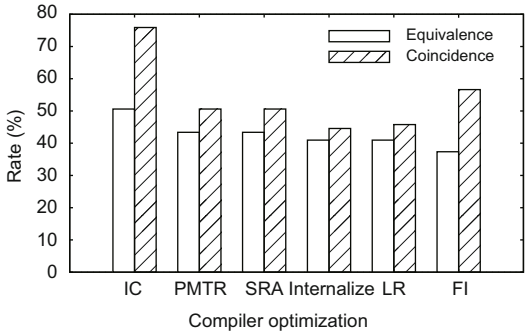


**Fig. 7 Rates of equivalence and coincidence with ALL**

At the second stage, we disable each optimization method and inspect the rates of equivalence and coincidence. Fig. 8 shows the MC/DC of ALL, disable-PMTR, disable-SRA, disable-LR, disable-Internalize, and disable-FI. The programs are ordered by MC/DC of ALL. We find that the equivalence rate of disabling the optimization method other than IC is extremely higher, with the lowest 87.95% and the highest 96.39%. As indicated in Fig. 8, the six coverage lines coincide. On the other hand, Fig. 9 shows the results of disabling IC. The equivalence rate of disable-IC is 43.37%, which is much less than those of the other five methods. These results indicate that disabling IC influences ALL most, but the influence of disabling other optimization methods is small. Therefore, we have the answer to RQ3: IC is the dominant optimization method in KLEE w.r.t. MC/DC.

### 3.3.4 Results on other coverage criteria

Based on our framework, we also study compiler optimization's influence on statement and decision coverages to see whether the results of MC/DC are still valid w.r.t. statement or decision coverages. The experimental results indicate that the result of RQ1

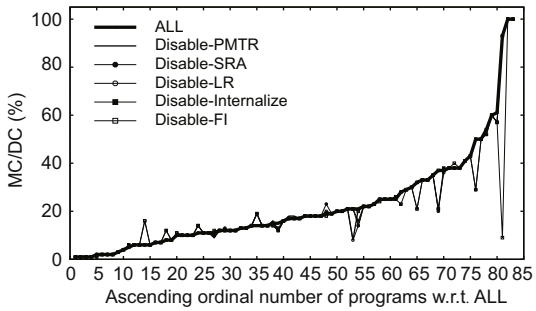


**Fig. 8 MC/DC of disabling PMTR, SRA, LR, Internalize, and FI**

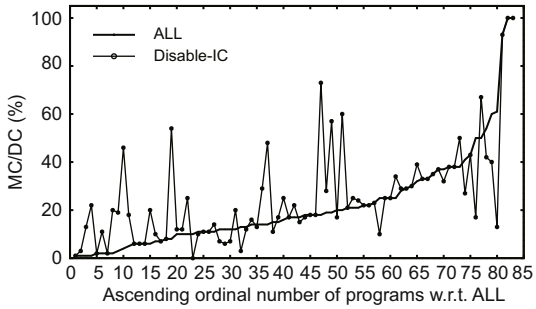


**Fig. 9 MC/DC of disabling IC**

w.r.t. statement or decision coverages is the same as that of MC/DC. There are 58 programs having the same statement coverage at 5, 10, 30, or 60 min, and 55 for decision coverage. In summary, more than 60% of the programs coverages do not increase after 5 min, regardless of the coverage type.

Considering the influence on coverage, the important compiler optimization methods w.r.t. statement or decision coverages are the same as those of MC/DC. Fig. 10 shows the number of influenced programs w.r.t. statement and decision coverage. As shown in Fig. 5, ALL also influences most programs w.r.t. statement or decision coverage, and IC's influence is close to ALL's. The number of influenced programs w.r.t. statement or decision coverage is smaller than that of MC/DC, and the number of influenced programs w.r.t. statement coverage is smaller than that of decision coverage. This indicates that more programs are influenced by compiler optimization when the coverage criterion is more complicated.

To study the dominant optimization method, we adopt the same process as MC/DC. Fig. 11 shows the equivalence and coincidence rates of the six optimization methods w.r.t. statement and decision coverages. As we see for MC/DC, IC also has the highest equivalence and coincidence rates. For statement

coverage, the equivalence and coincidence rates of IC are 62.65% and 78.31%, respectively, whereas the rates are 59.04% and 78.31% for decision coverage. In addition, the coincidence rate of IC is at least 20% higher than those of the remaining five methods on both statement and decision coverages.

Fig. 12 shows the equivalence and coincidence rates of disabling a single optimization method. As shown in the figure, the rates of disabling IC are the lowest on both statement and decision coverages. The equivalence rate of disabling IC is 51.81% for statement coverage, and the rate is 46.99% for decision coverage. The rates of the remaining five methods are very high, i.e., at least 90.36% for statement coverage and 87.95% for decision coverage. These results indicate that IC is also the dominant optimization method when the coverage criterion is statement or decision coverage. Hence, we have the

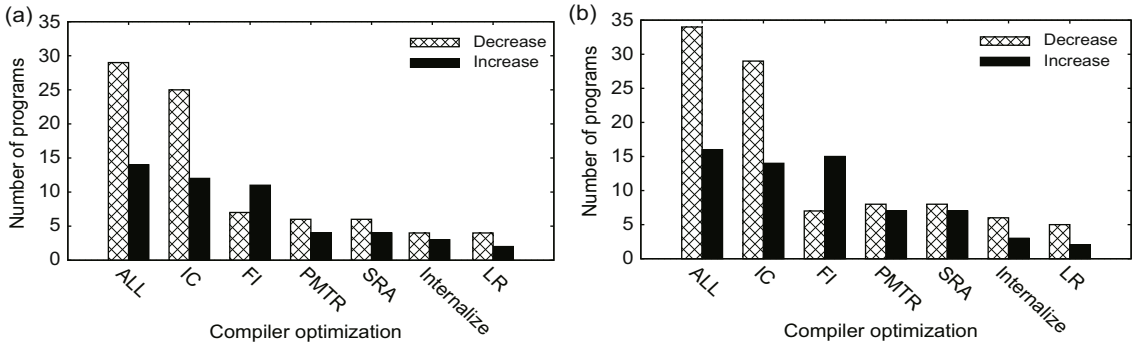


**Fig. 10 Number of influenced programs w.r.t. statement (a) and decision (b) coverages**

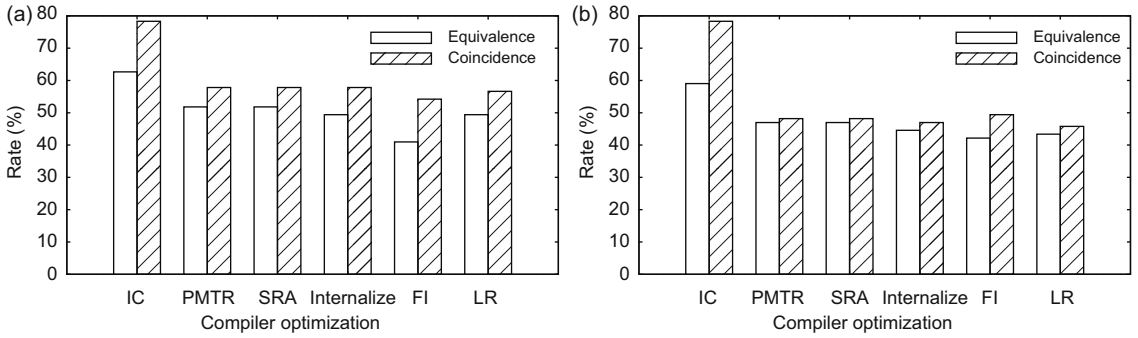


**Fig. 11 Rates of equivalence and coincidence with ALL w.r.t. statement (a) and decision (b) coverages while enabling optimization**

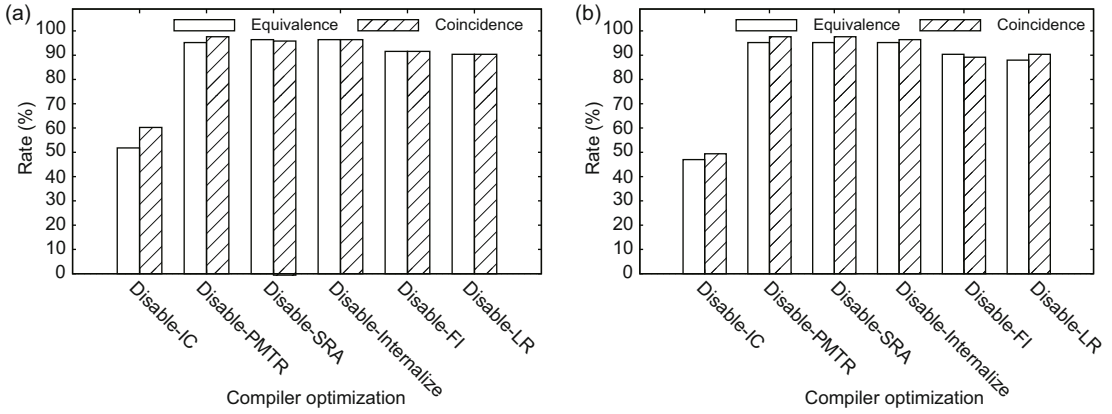


**Fig. 12 Rates of equivalence and coincidence with ALL w.r.t. statement (a) and decision (b) coverages while disabling optimization**

following answer to RQ4: Compiler optimization also influences the statement and decision coverages; IC is also the dominant optimization method in KLEE w.r.t. statement or decision coverage.

### 3.4 Discussion

According to the experimental results, compiler optimization influences testing coverage, which is coincident with the conclusion in Dong et al. (2015). However, different from Dong et al. (2015), we study MC/DC and the dominant compiler optimization methods of different coverage criteria. The experimental results also indicate that IC is the dominant compiler optimization method in KLEE. Actually, IC, i.e., InstCombine LLVM pass, is used to combine LLVM instructions into fewer and simpler instructions. For example, two constant add instructions can be combined into one add instruction. There are 1028 rules in IC, including arithmetic, memory-related, and control-flow optimizations. Usually, IC will be used multiple times during the LLVM optimization to reduce the number of instructions while ensuring semantic equivalence.

We explain IC's influence on MC/DC as follows. On one hand, reduced instructions can reduce the search space of the program and improve the efficiency of symbolic execution, i.e., exploring more paths in the same period, which is why IC can increase the coverages for some programs. This reason also applies to other optimization methods. On the other hand, the control-flow optimizations in IC, i.e., Br and Select instruction-related optimizations, may reduce the branches in the programs (Fig. 3). Hence, the symbolic executor will explore fewer paths and generate fewer test cases when analyzing the optimized IR program, resulting in a decreased coverage of source code. Usually, if there are optimized branches for a program, the coverage of the program will very likely decrease. In addition, compared with the other optimization methods, IC is the most frequently used during LLVM compiler optimization, which is also one reason why IC is the dominant method.

### 3.5 Threats to validity

There are possible external and internal threats to the validity of our results. The external threats come from the limited programs used for analysis. However, we offset these external threats in the following aspects to show the representativeness of the Coreutils program:

1. All the programs come from Coreutils and are widely used on Unix-like operating systems. Coreutils programs are often used as the benchmark programs in previous studies (Fehnker et al., 2006; Wagner et al., 2013; Joshi et al., 2015; Converse et al., 2017).

2. Our work concerns MC/DC, and the benchmark Coreutils programs are well distributed w.r.t. MC/DC conditions (Table 2).

The internal threats come from our evaluation methods for the dominant compiler optimization. The justification of this method is given as follows:

1. To evaluate the dominant compiler optimizations, we explore both the positive and negative influences of compiler optimization in a two-step analysis.

2. The experimental results are coincident with Dong et al. (2015) in statement and decision coverages.

3. The identification method of the dominant compiler optimizations is a general framework, which makes the results more convincing.

## 4 Optimization recommendation

Compiler optimization influences the coverages of the programs under test. If we can intelligently turn on or turn off the optimization method, e.g., turn it off when the optimization decreases the coverage, the coverage achieved by symbolic execution can be improved. In the remainder of this section, we first introduce our coverage-oriented method of recommending compiler optimizations for symbolic execution. Then we give and explain the experimental results. Finally, we discuss the threats to the validity of the recommendation method.

### 4.1 Recommendation method

Optimization recommendation needs the program feature w.r.t. the optimization to decide whether to adopt a compiler optimization. The feature extraction and synthesis of a program w.r.t. compiler optimization methods are challenging. The answer to RQ3 in Section 3.3 indicates that IC is the dominant compiler optimization method. Hence, whether to adopt IC is critical and directly determines the coverage results in many cases. According

to this result, we propose a program feature extraction method w.r.t. IC, based on which 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. There are 11 kinds of rules in IC, including AddSub, AndOrXor, and compare. Each kind has a set of optimization rules that optimize the LLVM instructions of several types. For example, the compare kind contains the rules for FCompInst and ICmpInst LLVM instructions. As pointed out by Alive (Lopes et al., 2015), each IC rule has a precondition that specifies the requirements of the rule. The rule preconditions can be used for feature extraction, i.e., calculating a program's chance of applying a rule. However, for precise feature extraction, analysis on the program w.r.t. the preconditions of IC rules is required, which is not practical and introduces large overhead for feature extraction. Hence, we propose a lightweight method for feature extraction. The basic idea of feature extraction is to run IC optimization on the program, and record the times of successful optimizations on each LLVM instruction type. In total, IC handles 43 types of LLVM instructions. Hence, given a program $P$, its feature w.r.t. IC, denoted by $F(P)$, is a vector of 43 dimensions, i.e., $\langle c_1, c_2, \ldots, c_{43}\rangle$, where $c_i \geq 0$ $(i = 1, 2, \ldots, 43)$ is the number of successful optimizations on the $i^{\text{th}}$ instruction type.

Based on IC's feature extraction method, we can extract the features of the 83 Coreutils programs, and use the support vector machine (SVM) (Cortes and Vapnik, 1995) to train a classifier with the extracted features. According to the experimental results in Section 3, we can label the feature of each program. If IC increases the MC/DC of a program, we label the feature as 1; otherwise, we label the feature as $-1$, including the cases of decreasing and no influence. There exists a problem of data imbalance (Chawla, 2005), because disable-IC outperforms ALL in more cases. To alleviate the imbalance problem, we make several copies of positive samples, making the ratio between positive and negative samples 1 : 2. The reason why the ratio is not 1 : 1 is that a large number of duplications may lead to bias. Then we use LIBSVM (Chang and Lin, 2011) to train the model. The adopted optimal SVM type is C-SVM, and the core function is radial basis function (RBF) (Chen S et al., 1991), whose parameters $c$ and $g$ are 8.0 and 0.5, respectively. The precision of the trained model is 83.13%. Based on the trained model, we can decide whether to carry out IC before symbolic execution. Fig. 13 shows the procedure of training and recommendation.

The feature extractor provides the functionalities of both IR generation and feature extraction. The extractor accepts a program and produces the program's IR representation and its IC feature. Then the feature and the IR representation are input into an SVM classifier using the trained model and the optimizer. If the classifier decides that IC needs to be carried out, all the optimizations will be applied on the IR representation; otherwise, IC will not be applied, but the remaining optimizations are still applied. Symbolic execution will be carried out on the optimized IR representation to generate test cases. In this way, we can recommend whether to apply IC before symbolic execution to improve the coverage.

The recommendation method considers only two cases: ALL and disable-IC. Then we can inspect the limit of our recommendation method on Coreutils programs. We collect the maximum MC/DC value of each program in all the configurations

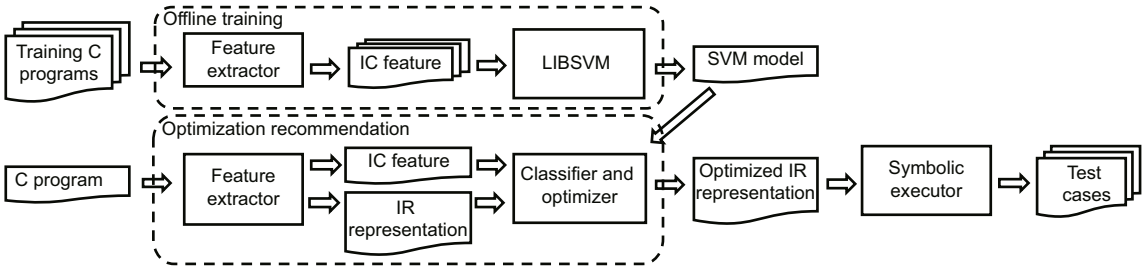


**Fig. 13 Training and recommendation procedure**

(denoted by ideal), and the maximum MC/DC of ALL and disable-IC. Fig. 14 shows the results. The programs are ordered by the MC/DC of ALL and disable-IC. The coincidence rate of two lines is 72.29%, which means that the recommendation method's rate of achieving the maximum MC/DC value is at most 72.29%. In the next subsection, the experimental results demonstrate the effectiveness of the recommendation method in practice.

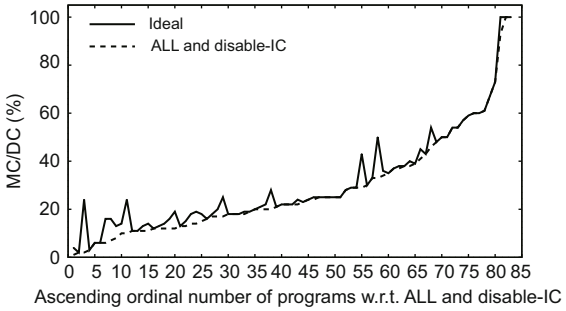


**Fig. 14 Limit of the recommendation method**

## 4.2 Experimental results

### 4.2.1 Effectiveness of the recommendation method

We implement the optimization recommendation in KLEE and provide an optimization option auto. Our experiments are carried out on Coreutils programs and other real-world programs. Experiments are expected to answer the following research questions:

RQ5: How effective is the recommendation method on MC/DC?

RQ6: Is the recommendation method also effective for other programs?

We integrate the recommendation method into the framework in Fig. 4 and reran the experiments on the 83 Coreutils programs. Fig. 15 displays the results of the recommendation method. The programs are ordered by the MC/DC of auto. We find that the line of ALL and disable-IC and the line of auto are basically coincident at 83.13%. On 50.60% (less than 72.29%) of the programs, the recommendation method achieves the maximum MC/DC, i.e., ideal.

Furthermore, we inspect the number of programs influenced by auto. The recommendation method increases and decreases MC/DC on 18 and 13 programs w.r.t. NO, respectively; the numbers for ALL are 18 and 37, respectively; the numbers for disable-IC are 13 and 16, respectively. The recommendation method increases the coverage on more programs and decreases the coverage on fewer programs. Hence, the recommendation method outperforms both ALL and disable-IC.

We also compare the recommendation method with NO and ALL. Fig. 16 shows the results. The programs are ordered by the MC/DC of NO. As shown in the figure, the recommendation method's line is above those of ALL and NO on most (67.47%) programs, which indicates the effectiveness of the recommendation method. In addition, it indicates that simply turning on or turning off all the optimizations cannot achieve better coverage.

Therefore, we have the answer to RQ5: The recommendation method likely provides a sensible recommendation w.r.t. MC/DC, which shows a high rate of coincidence, 83.13%, with the lines of ALL and disable-IC.

To further investigate our recommendation method's effectiveness, we apply it to other programs like NECLA (v1.1) (https://www.neclabs.com/research/system/systems_SAV-website/benchmarks.php), which is a set of C programs with various C-specific bug patterns. We select 23 programs from the benchmark according to whether the program contains branch statements and the uncertainty of inputs.

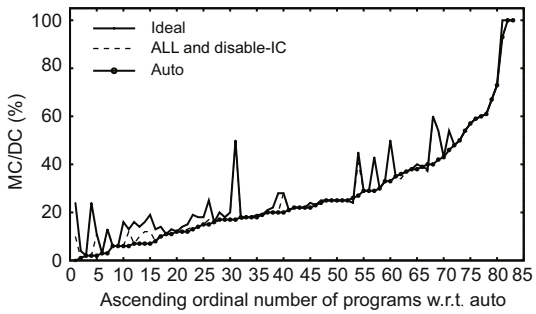


**Fig. 15 MC/DC of the recommendation method**

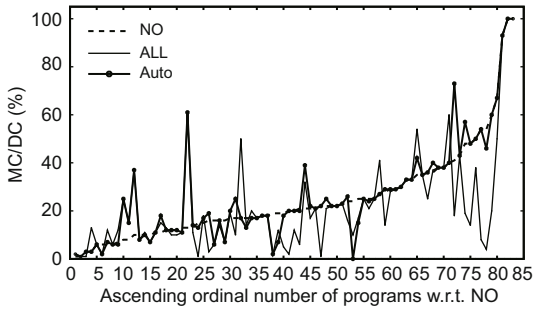


**Fig. 16 Comparing recommendation with NO and ALL**

Similar to Fig. 16, we compare the recommendation method with NO and ALL on the NECLA benchmark. The results are shown in Fig. 17; the programs are ordered by the MC/DC of NO. As the figure shows, the recommendation method's line (auto) is above those of ALL and NO on most (78.26%) programs, which implies the effectiveness of this method on the NECLA benchmark.

Therefore, we have the answer to RQ6: The recommendation method is also effective in improving the coverage on the NECLA benchmark.

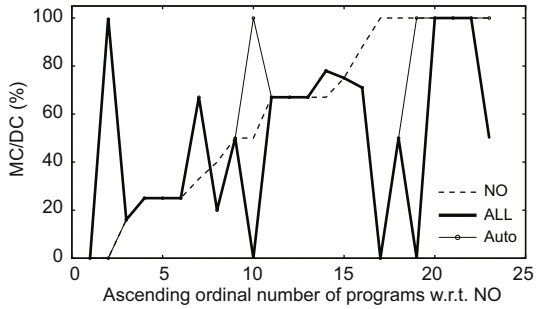


**Fig. 17 Comparison on the NECLA benchmark**

### 4.2.2 Comparison with LEO

Chen JJ et al. (2018) proposed a machine learning based method—the prototype implementation's name is LEO (https://github.com/JunjieChen/leo)—of recommending compile optimizations for symbolic execution to improve statement coverage. LEO extracts the static program features (e.g., the number of basic blocks and the number of direct calls) from the program under testing and the libraries. Then, LEO recommends the compile optimizations for each function unit of the program and the libraries. Compared with LEO, we study the influence of compiler optimization w.r.t. MC/DC, and recommend the compiler optimizations w.r.t. MC/DC. Different from LEO, our optimization recommendation considers only IC, which is the dominant optimization method according to the empirical study in Section 3. In addition, compared with the static program feature extraction by LEO, we extract the program features based on the internal procedure information of applying IC optimization, which is more accurate w.r.t. the optimization. However, in our method the optimization needs to be done once more to obtain the features.

It is interesting to compare the two methods empirically. Because LEO's code and data are incomplete in its GitHub repo (the feature extraction component, the trained model, and the test cases are unavailable, which are checked on August 21, 2019), we fail to use LEO to train a model w.r.t. MC/DC. Although our goal is not to improve statement coverage, we try to make a tradeoff to compare the two methods w.r.t. statement coverages. We extract the results in Chen JJ et al. (2018) which were obtained by disabling the cache-based query optimizations during training and testing. However, because query cache is an effective method of KLEE to improve statement coverage in practice, we enable cache-based query optimizations during training as used in the above experiments.

We use KLEE with auto to analyze the Coreutils programs in 10 min (with DFS search heuristic) and collect the statement coverages achieved by the generated test cases. Considering the difference between the experimental environments and the configurations of the two methods, we consider only the programs on which LEO and auto (disabling query optimizations) have equivalent results (less than 5% difference) on ALL and NO. We collect the testing's coverage results of both turning on and turning off the query optimizations to make an indirect comparison. Table 3 shows the results.

We can observe the following in Table 3:

1. The query optimizations, i.e., `-use-cache` and `-use-cex-cache`, influence the statement coverage results of different compiler optimization configurations. When we turn on the query optimizations, KLEE achieves a higher statement coverage on nine programs with NO than with ALL, and an equal statement coverage on five programs. However, on these programs (14/17), KLEE performs better with ALL than with NO when turning off the query optimizations.

2. LEO turns off the two query optimizations during training and testing; however, we turn on the two optimizations during training and testing. As shown by the table and the results of the previous experiments, auto tends to select disable-IC for Coreutils programs, which has a similar result to NO. When turning off the query optimizations, auto also recommends disable-IC, which decreases the statement coverages for many programs, because ALL performs better than NO when disabling query optimizations.

**Table 3 Experimental results of 10 min using depth-first search**

| Program | Statement coverage (%) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Disable query optimizations | | | | | | Enable query optimizations | | |
| | Chen JJ et al. (2018) | | | Ours | | | Ours | | |
| | NO | ALL | LEO | NO | ALL | auto | NO | ALL | auto |
| chgrp | 34.44 | 67.78 | 67.78 | 34.44 | 68.89 | 34.44 | 86.67 | 41.11 | 82.22 |
| chown | 30.11 | 60.22 | 65.59 | 30.11 | 61.29 | 30.11 | 84.95 | 29.03 | 84.95 |
| cp | 26.29 | 46.34 | 41.46 | 25.82 | 46.47 | 23.91 | 49.73 | 30.43 | 46.47 |
| df | 63.61 | 64.42 | 64.15 | 59.30 | 67.39 | 59.03 | 60.92 | 65.23 | 65.77 |
| dircolors | 10.00 | 20.00 | 73.16 | 10.00 | 20.00 | 10.00 | 39.47 | 14.74 | 38.42 |
| factor | 64.18 | 71.64 | 71.64 | 64.18 | 71.64 | 64.18 | 65.67 | 49.25 | 65.67 |
| hostid | 59.09 | 63.64 | 63.64 | 59.09 | 63.64 | 59.09 | 63.64 | 63.64 | 63.64 |
| link | 60.71 | 64.29 | 75.00 | 64.29 | 64.29 | 64.29 | 64.29 | 64.29 | 64.29 |
| logname | 52.00 | 56.00 | 56.00 | 52.00 | 56.00 | 52.00 | 56.00 | 56.00 | 56.00 |
| mkdir | 34.85 | 66.67 | 77.27 | 34.85 | 71.21 | 28.79 | 71.21 | 34.85 | 68.18 |
| mkfifo | 36.17 | 74.47 | 82.98 | 36.17 | 74.47 | 36.17 | 74.47 | 48.94 | 74.47 |
| pinky | 79.91 | 83.33 | 83.33 | 83.76 | 83.33 | 83.76 | 83.33 | 76.07 | 83.33 |
| pwd | 20.34 | 20.34 | 20.34 | 20.34 | 20.34 | 20.34 | 20.34 | 20.34 | 20.34 |
| sleep | 43.48 | 45.65 | 45.65 | 43.48 | 45.65 | 43.48 | 45.65 | 45.65 | 65.22 |
| tr | 40.52 | 39.15 | 22.15 | 40.36 | 40.36 | 40.52 | 37.48 | 33.08 | 33.69 |
| uname | 19.32 | 77.27 | 79.55 | 19.32 | 79.55 | 19.32 | 80.68 | 21.59 | 80.68 |
| whoami | 50.00 | 53.85 | 53.85 | 50.00 | 53.85 | 50.00 | 53.85 | 53.85 | 53.85 |

We also compare the two methods using the configuration of turning on query optimizations and employing a random search strategy (default `nurs:covnew` search strategy), under which the statement coverage can be improved further. Table 4 gives the results. As in Table 3, we consider only the programs on which these two methods have equivalent results on NO and ALL. As shown in the table, if we use a random search strategy and enable query optimization, the statement coverages achieved by both LEO and auto are close to those of NO and ALL, which indicates that the improvements become relatively small when employing more orthogonal optimization techniques to improve the efficiency of symbolic execution.

**Table 4 Experimental results using random search and query optimizations**

| Program | Statement coverage (%) | | | | | |
|---|---|---|---|---|---|---|
| | Chen JJ et al. (2018) | | | Ours | | |
| | NO | ALL | LEO | NO | ALL | auto |
| cp | 48.78 | 49.05 | 48.24 | 50.27 | 50.54 | 53.80 |
| ls | 53.46 | 50.34 | 53.93 | 55.76 | 51.22 | 46.14 |
| nice | 96.61 | 94.92 | 96.61 | 94.92 | 94.92 | 94.92 |
| od | 84.11 | 86.08 | 86.08 | 79.47 | 84.11 | 76.79 |
| paste | 92.51 | 92.51 | 92.51 | 92.51 | 92.51 | 92.51 |
| printenv | 100 | 100 | 100 | 100 | 100 | 100 |
| pwd | 20.34 | 20.34 | 20.34 | 20.34 | 20.34 | 20.34 |

In summary, LEO and auto can perform better than other methods on different programs, but a direct comparison between LEO and auto w.r.t. statement coverage is not preferable. The main reason is that the two methods have different configurations of query cache optimization. It is more reasonable to train the optimization recommendation model using the data generated under the configuration where orthogonal techniques are used to improve the efficiency of symbolic execution.

### 4.3 Threats to validity

The external threats come from the limited program for analysis and training. We relax this threat as discussed in Section 3.5. In addition, we plan to address this threat further in the future by analyzing more real-world programs. As for the training process, although the problem of imbalance exists for the original training data, we duplicate the positive samples to handle the imbalance problem. For the 52 programs (there are 83 in total) that ALL and disable-IC perform differently, auto performs the same as ALL on nine programs (17%).

The internal threats come from the design and implementation of the recommendation method. We control these threats as follows:

1. Although the recommendation method

considers only the two cases ALL and disable-IC, the two compilation optimization options can manifest the various compilation optimization options. For example, as shown in Fig. 8, the optimization ALL presents a high equivalent rate with the optimizations that disable the specific optimization other than IC, which indicates that the performance of ALL can represent those of these optimizations. In addition, the coincidence rates of the optimization methods disable-IC with PMTR and FI are 84.34% and 78.31%, respectively. These high coincidence rates also imply that the performance of disable-IC is representative of these single optimization options.

2. The classifier trained by Coreutils programs can be applied to other programs, e.g., the NECLA benchmark. In addition, we plan to train the classifier on more programs.

## 5 Related works

Our work is related to the existing studies that improve the effectiveness and efficiency of symbolic execution, study the impact of compiler optimizations on symbolic execution, and recommend optimal compiler optimization for symbolic execution.

There exist many studies for improving the efficiency of symbolic execution. Many highly efficient search heuristics have been proposed, such as target-oriented search (Ma et al., 2011), coverage-oriented search (Li et al., 2013), and context-guided search (Seo and Kim, 2014). Wang et al. (2018) provided an optimal strategy to improve the effectiveness of dynamic symbolic execution. In addition, an alternative method is to use some abstraction techniques to reduce the search space of the program, such as state merging (Kuznetsov et al., 2012), partial order reduction (Khurshid et al., 2003), and slicing (Cui et al., 2013). Another option is to employ distributed computing techniques to parallelize symbolic execution (Fan et al., 2009; Staats and Păsăreanu, 2010; Bucur et al., 2011). Christakis et al. (2016) guided dynamic symbolic execution by aborting tests that lead to verified execution and further explore more unverified executions. Sen et al. (2015) provided a tool, MultiSE, which uses a value summary to represent the state and shows high efficiency compared with traditional dynamic symbolic execution. Based on the logical relations among constraints, GreenTrie reuses the constraint-solving results to reduce the constraint solver time during symbolic execution (Jia XY et al., 2015). Zhang et al. (2015) and Yu et al. (2018) used static and dynamic analysis to boost the verification of regular properties by dynamic symbolic execution. Our work is complementary to the existing work.

Different from the above work on symbolic execution, we explore the effectiveness of symbolic execution from the perspective of compiler optimizations. Compiler optimization plays an important role in the testing and verification processes. For example, the impact of compiler optimizations on mutation testing was investigated in Hariri et al. (2016). Cadar (2015) considered program transformations as an important aspect for scalable symbolic execution and gave several examples of why transformations influence scalability. However, no empirical studies were provided, and no recommendation method was proposed in Cadar (2015). Dong et al. (2015) and Wagner et al. (2013) studied the relation between compiler optimizations and symbolic execution. Specifically, Dong et al. (2015) found that compiler optimization influences symbolic execution w.r.t. statement and decision coverages. Different from Dong et al. (2015), we inspect the influence w.r.t. MC/DC. In addition, we investigate the key and dominant optimizations. Also, recommending compiler optimizations was not investigated in Dong et al. (2015). In Barr et al. (2018), the program was rewritten by indexification to obtain a subset of the program's original search space, which makes the path conditions easier for the SMT solver. In Converse et al. (2017), non-semantics-preserving program transformations were studied to improve the coverage achieved by symbolic execution. In addition, semantics-preserving transformations were proposed and used online to simplify the array path constraints during symbolic execution (Perry et al., 2017).

There has been much work on recommending optimal compiler optimizations for a given program to achieve better performance (Tiwari et al., 2009). In recent years, many studies have used machine learning methods to recommend the optimal compiler optimization sequence for a program. For example, an iterative selection method for optimization options was proposed based on program code features (Agakov et al., 2006). Another example was to use an automatic performance-tuning algorithm,

i.e., combined elimination, to select the optimal compilation optimization combination in a shorter time (Pan and Eigenmann, 2006). However, as far as we know, there exists less work for recommending compiler optimizations in the context of symbolic execution. The only work is LEO (Chen JJ et al., 2018). We have compared our method with LEO in Section 4.2.2.

## 6 Conclusions

Compiler optimizations influence the effectiveness and efficiency of symbolic execution. In this study, we took MC/DC as the criterion for evaluating symbolic execution. Our empirical study indicated that compiler optimizations influence MC/DC, and IC is the dominant optimization method. Then, we designed and implemented a lightweight recommendation method w.r.t. IC towards improving MC/DC. The experimental results indicated that the method is effective. Future work lies in four aspects: (1) conducting more extensive experiments on other benchmarks; (2) designing a more effective recommendation method; (3) inspecting whether the recommendation method is still valid w.r.t. other coverage criteria; (4) exploring the influence of combinatorial compiler optimization options.

### Contributors

Wei-jiang HONG and Yi-jun LIU designed the research. Zhen-bang CHEN guided the research. Wei-jiang HONG and Zhen-bang CHEN drafted the manuscript. Wei DONG helped organize the manuscript. Ji WANG polished the paper. Wei-jiang HONG and Zhen-bang CHEN finalized the paper.

### Acknowledgements

We thank Jun-jie CHEN and Dan HAO with Peking University for supporting the empirical comparison with LEO.

### Compliance with ethics guidelines

Wei-jiang HONG, Yi-jun LIU, Zhen-bang CHEN, Wei DONG, and Ji WANG declare that they have no conflict of interest.

### References

Agakov F, Bonilla E, Cavazos J, et al., 2006. Using machine learning to focus iterative optimization. Proc Int Symp on Code Generation and Optimization, p.295-305. https://doi.org/10.1109/cgo.2006.37

Aho AV, Sethi R, Ullman JD, 1986. Compilers: Principles, Techniques, and Tools. Addison-Wesley, Heidelberg, Berlin, Germany.

Ammann P, Offutt J, 2016. Introduction to Software Testing. Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/9781316771273

Barr ET, Clark D, Harman M, et al., 2018. Indexing operators to extend the reach of symbolic execution. https://arxiv.org/abs/1806.10235

Beizer B, 2003. Software Testing Techniques. Dreamtech Press, New Delhi, India.

Bucur S, Ureche V, Zamfir C, et al., 2011. Parallel symbolic execution for automated real-world software testing. Proc 6th Conf on Computer Systems, p.183-198. https://doi.org/10.1145/1966445.1966463

Cadar C, 2015. Targeted program transformations for symbolic execution. Proc 10th Joint Meeting on Foundations of Software Engineering, p.906-909. https://doi.org/10.1145/2786805.2803205

Cadar C, Dunbar D, Engler D, 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. Proc 8th USENIX Symp on Operating Systems Design and Implementation, p.209-224.

Chang CC, Lin CJ, 2011. LIBSVM: a library for support vector machines. *ACM Trans Intell Syst Technol*, 2(3):27. https://doi.org/10.1145/1961189.1961199

Chawla NV, 2005. Data mining for imbalanced datasets: an overview. In: Maimon O, Rokach L (Eds.), Data Mining and Knowledge Discovery. Springer, Boston, USA, p.853-867. https://doi.org/10.1007/0-387-25465-X_40

Chen JJ, Hu WX, Zhang LM, et al., 2018. Learning to accelerate symbolic execution via code transformation. Proc 32nd European Conf on Object-Oriented Programming, Article 6. https://doi.org/10.4230/LIPIcs.ECOOP.2018.6

Chen S, Cowan CFN, Grant PM, 1991. Orthogonal least squares learning algorithm for radial basis function networks. *IEEE Trans Neur Netw*, 2(2):302-309. https://doi.org/10.1109/72.80341

Chen TY, Cheung SC, Yiu SM, 1998. Metamorphic Testing: a New Approach for Generating Next Test Cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, China.

Christakis M, Müller P, Wüstholz V, 2016. Guiding dynamic symbolic execution toward unverified program executions. Proc 38th Int Conf on Software Engineering, p.144-155. https://doi.org/10.1145/2884781.2884843

Converse H, Olivo O, Khurshid S, 2017. Non-semantics-preserving transformations for higher-coverage test generation using symbolic execution. Proc IEEE Int Conf on Software Testing, Verification and Validation, p.241-252. https://doi.org/10.1109/icst.2017.29

Cortes C, Vapnik V, 1995. Support-vector networks. *Mach Learn*, 20(3):273-297. https://doi.org/10.1007/BF00994018

Cui HM, Hu G, Wu JY, et al., 2013. Verifying systems rules using rule-directed symbolic execution. Proc 18th Int Conf on Architectural Support for Programming Languages and Operating Systems, p.329-342. https://doi.org/10.1145/2499368.2451152

de Moura L, Bjørner N, 2008. Z3: an efficient SMT solver. Proc 14th Int Conf on Tools and Algorithms for the Construction and Analysis of Systems, p.337-340. https://doi.org/10.1007/978-3-540-78800-3_24

de Moura L, Bjørner N, 2011. Satisfiability modulo theories: introduction and applications. *Commun ACM*, 54(9): 69-77. https://doi.org/10.1145/1995376.1995394

Dong S, Olivo O, Zhang L, et al., 2015. Studying the influence of standard compiler optimizations on symbolic execution. Proc 26th Int Symp on Software Reliability Engineering, p.205-215. https://doi.org/10.1109/issre.2015.7381814

Dupuy A, Leveson N, 2000. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. Proc 19th Digital Avionics Systems Conf, Article 1.B.6. https://doi.org/10.1109/DASC.2000.886883

Duran JW, Ntafos SC, 1984. An evaluation of random testing. *IEEE Trans Softw Eng*, SE-10(4):438-444. https://doi.org/10.1109/tse.1984.5010257

EUROCAE (European Organisation for Civil Aviation Equipment), 1998. Software Considerations in Airborne Systems and Equipment Certification, RTCA DO-178B. EUROCAE, Paris, France.

Fan WQ, Liang HL, Yang YX, et al., 2009. A novel symbolic execution framework for multi-procedure program analysis. Proc 2nd IEEE Int Conf on Broadband Network & Multimedia Technology, p.858-863. https://doi.org/10.1109/icbnmt.2009.5347802

Fehnker A, Huuck R, Jayet P, et al., 2006. Goanna—a static model checker. Proc Int Workshop on Parallel and Distributed Methods in Verification, p.297-300. https://doi.org/10.1007/978-3-540-70952-7_20

Fewster M, Graham D, 2000. Software Test Automation: Effective Use of Test Execution Tools. Emerald Group Publishing, Bingley, UK. https://doi.org/10.1108/k.2000.29.3.392.5

Godefroid P, Klarlund N, Sen K, 2005. DART: directed automated random testing. Proc ACM SIGPLAN Conf on Programming Language Design and Implementation, p.213-223. https://doi.org/10.1145/1065010.1065036

Godefroid P, Levin MY, Molnar D, et al., 2008. Automated whitebox fuzz testing. Proc Network and Distributed System Security Symp, p.151-166.

Hariri F, Shi A, Converse H, et al., 2016. Evaluating the effects of compiler optimizations on mutation testing at the compiler IR level. Proc 27th Int Symp on Software Reliability Engineering, p.105-115. https://doi.org/10.1109/issre.2016.51

Hayhurst KJ, Veerhusen DS, 2001. A practical tutorial on modified condition/decision coverage. Proc 20th Digital Avionics Systems Conf, Article 1.B.2. https://doi.org/10.1109/dasc.2001.963305

Jia Y, Harman M, 2011. An analysis and survey of the development of mutation testing. *IEEE Trans Softw Eng*, 37(5):649-678. https://doi.org/10.1109/tse.2010.62

Jia XY, Ghezzi C, Ying S, 2015. Enhancing reuse of constraint solutions to improve symbolic execution. Proc Int Symp on Software Testing and Analysis, p.177-187. https://doi.org/10.1145/2771783.2771806

Joshi HP, Dhanasekaran A, Dutta R, 2015. Impact of software obfuscation on susceptibility to return-oriented programming attacks. Proc 36th Sarnoff Symp, p.161-166. https://doi.org/10.1109/sarnof.2015.7324662

Khurshid S, Păsăreanu CS, Visser W, 2003. Generalized symbolic execution for model checking and testing. Proc Int Conf Tools and Algorithms for the Construction and Analysis of Systems, p.553-568. https://doi.org/10.1007/3-540-36577-x_40

King JC, 1976. Symbolic execution and program testing. *Commun ACM*, 19(7):385-394. https://doi.org/10.1145/360248.360252

Kuznetsov V, Kinder J, Bucur S, et al., 2012. Efficient state merging in symbolic execution. 33rd ACM SIGPLAN Conf on Programming Language Design and Implementation, p.193-204. https://doi.org/10.1145/2345156.2254088

Lattner C, Adve VS, 2004. LLVM: a compilation framework for lifelong program analysis & transformation. Proc Int Symp on Code Generation and Optimization, p.75-88. https://doi.org/10.1109/cgo.2004.1281665

Li Y, Su Z, Wang L, et al., 2013. Steering symbolic execution to less traveled paths. *ACM SIGPLAN Not*, 48(10):19-32. https://doi.org/10.1145/2544173.2509553

Lopes NP, Menendez D, Nagarakatte S, et al., 2015. Provably correct peephole optimizations with alive. *ACM SIGPLAN Not*, 50(6):22-32. https://doi.org/10.1145/2813885.2737965

Ma KK, Phang KY, Foster JS, et al., 2011. Directed symbolic execution. Proc Int Static Analysis Symp, p.95-111. https://doi.org/10.1007/978-3-642-23702-7_11

McKeeman WM, 1998. Differential testing for software. *Dig Techn J*, 10(1):100-107.

Pan ZL, Eigenmann R, 2006. Fast and effective orchestration of compiler optimizations for automatic performance tuning. Proc Int Symp on Code Generation and Optimization, p.319-332. https://doi.org/10.1109/cgo.2006.38

Perry DM, Mattavelli A, Zhang X, et al., 2017. Accelerating array constraints in symbolic execution. Proc 26th ACM SIGSOFT Int Symp on Software Testing and Analysis, p.68-78. https://doi.org/10.1145/3092703.3092728

Sen K, Marinov D, Agha G, 2005. CUTE: a concolic unit testing engine for C. Proc 10th European Software Engineering Conf held jointly with 13th ACM SIGSOFT Int Symp on Foundations of Software Engineering, p.263-272. https://doi.org/10.21236/ada482657

Sen K, Necula G, Gong L, et al., 2015. MultiSE: multi-path symbolic execution using value summaries. Proc 10th Joint Meeting on Foundations of Software Engineering, p.842-853. https://doi.org/10.1145/2786805.2786830

Seo H, Kim S, 2014. How we get there: a context-guided search strategy in concolic testing. Proc 22nd ACM SIGSOFT Int Symp on Foundations of Software Engineering, p.413-424. https://doi.org/10.1145/2635868.2635872

Staats M, Păsăreanu C, 2010. Parallel symbolic execution for structural test generation. Proc 19th Int Symp on Software Testing and Analysis, p.183-194. https://doi.org/10.1145/1831708.1831732

Su T, Ke W, Miao W, et al., 2017. A survey on data-flow testing. *ACM Comput Surv*, 50(1):5. https://doi.org/10.1145/3020266

Tillmann N, de Halleux J, 2008. Pex—white box test generation for .NET. Proc Int Conf on Tests and Proofs, p.134-153. https://doi.org/10.1007/978-3-540-79124-9_10

Tiwari A, Chen C, Chame J, et al., 2009. A scalable auto-tuning framework for compiler optimization. Proc IEEE Int Symp on Parallel & Distributed Processing, p.1-12. https://doi.org/10.1109/ipdps.2009.5161054

Wagner J, Kuznetsov V, Candea G, 2013. -OVERIFY: optimizing programs for fast verification. 14th USENIX Conf on Hot Topics in Operating Systems, p.1-6.

Wang X, Sun J, Chen Z, et al., 2018. Towards optimal concolic testing. Proc 40th Int Conf on Software Engineering, p.291-302. https://doi.org/10.1145/3180155.3180177

Wong E, Zhang L, Wang S, et al., 2015. DASE: document-assisted symbolic execution for improving automated software testing. Proc 37th IEEE Int Conf on Software Engineering, p.620-631. https://doi.org/10.1109/icse.2015.78

Wong WE, 2001. Mutation Testing for the New Century. Springer, Boston, USA. https://doi.org/10.1007/978-1-4757-5939-6

Yu H, Chen Z, Wang J, et al., 2018. Symbolic verification of regular properties. Proc 40th Int Conf on Software Engineering, p.871-881. https://doi.org/10.1145/3180155.3180227

Zhang Y, Chen Z, Wang J, et al., 2015. Regular property guided dynamic symbolic execution. Proc 37th IEEE Int Conf on Software Engineering, p.643-653. https://doi.org/10.1109/ICSE.2015.80

Zhu H, Hall PA, May JH, 1997. Software unit test coverage and adequacy. *ACM Comput Surv*, 29(4):366-427. https://doi.org/10.1145/267580.267590

Zhen-bang CHEN received his BS degree and PhD degree in Computer Science in 2002 and 2009, respectively, from the College of Computer, National University of Defense Technology, Changsha, China. He is now an associate professor at the College of Computer, National University of Defense Technology, China. His research interests include program analysis, formal methods, and their applications.

Ji WANG received his BS degree and PhD degree in Computer Science in 1987 and 1995, respectively, from the College of Computer, National University of Defense Technology, Changsha, China. He is now a full professor at the State Key Laboratory of High Performance Computing, National University of Defense Technology, China. He is an executive associate editor-in-chief of *Frontiers of Information Technology & Electronic Engineering*. His research interests include formal methods and software engineering.