When Compiler Optimizations Meet Symbolic Execution: An Empirical Study

Yue Zhang[*]
Drexel University
Philadelphia, PA, USA
yz899@drexel.edu

Melih Sirlanci
The Ohio State University
Columbus, OH, USA
sirlanci.2@osu.edu

Ruoyu Wang
Arizona State University
Tempe, AZ, USA
fishw@asu.edu

Zhiqiang Lin
The Ohio State University
Columbus, OH, USA
zlin@cse.ohio-state.edu

Abstract

Compiler optimizations intend to transform a program into a semantic-equivalent one with improved performance, but it is unclear how these optimizations may impact the performance of dynamic symbolic execution (DSE) on binary code. To systematically understand the impact of compiler optimizations on two popular DSE techniques (i.e., symbolic exploration and symbolic tracing), this paper presents an empirical study that quantifies 209 GCC compilation flags and 73 Clang compilation flags to reveal both positive and negative optimizations to DSE. Our data set contains 992 unique test cases, which are produced from 3,449 source files in the GCC test suite. After analyzing 2,978,976 binary programs that we compiled with two compilers and various compilation flags, we found that although some optimizations make DSE faster, most optimizations will actually slow down DSE. Our analysis further reveals *root causes* behind these impacts. The most positive impacts that optimizations have on DSE come from the reduction of the number of instructions and program paths, whereas negative impacts are caused by a series of unexpected behaviors, including increased numbers of instructions or program paths, library function inlining preventing DSE engines from using function summaries, and arithmetic optimizations leading to more sophisticated constraints. Being the first in-depth analysis on *why* compiler flags influence the performance of DSE, this project sheds light on program transformations that can be applied before performing DSE tasks for better performance.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Compiler Optimization; Symbolic Execution; Measurement Study

ACM Reference Format:
Yue Zhang, Melih Sirlanci, Ruoyu Wang, and Zhiqiang Lin. 2024. When Compiler Optimizations Meet Symbolic Execution: An Empirical Study. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3658644.3670372

1 Introduction

Compiler optimizations are transformations that take a program as input and produce a semantically equivalent executable, usually with improved performance, such as lower memory usage, smaller images, and most importantly, faster execution: An optimized C executable may run twice as fast as its unoptimized counterpart [3]. Unfortunately, compiler optimizations often hamper key security applications, such as binary code matching [38], function signature recovery [28], and dynamic symbolic execution (DSE) [12, 16, 21].

Being a popular software testing technique, DSE symbolically executes a program by simulating it in a software execution engine where the program's variables, registers, and memory data are symbolic values [25, 39]. A key advantage of DSE is that it can reason about input along a certain program path by solving collected path predicates (also known as path constraints) in a constraint solver. Theoretically, given enough time and computing resources, DSE can systematically explore all paths (or states) of any program. However, DSE techniques are extremely expensive, and even more so when applied on binary programs [36].

Some compiler optimizations have been known to negatively affect the performance of DSE [12, 16, 21]. Nevertheless, existing studies are limited because: (a) they focus on combinations of optimization flags (e.g., -O1 or -O2, which are combinations of multiple flags) or manually selected optimization flags, (b) they only study the source-based DSE solution (KLEE) while DSE on binary code is often used in security settings [6], and (c) they evaluate optimization flags by performing DSE on large code bases (e.g., Coreutils), which hides the performance impact of individual flags on code with different properties. More importantly, existing studies leave open a critical research question, which we seek to answer in this paper: *Why do compiler optimizations impact the performance of DSE on binary code?*

Instead of statistically measuring the impact of a few compiler flags on DSE (as in existing work), in this paper, we study a comprehensive set of compiler options (including 209 GCC flags and 73

[*] This work was completed while this author was affiliated with The Ohio State University.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670372

Clang flags) and reveal *why* these optimizations affect the performance of DSE by measuring their impact on low-level operations in DSE engines. We create a framework to automatically and systematically **M**easure the impacts of compiler **O**ptimizations **O**n **S**ymbolic **E**xecution (MOOSE). The high-level idea is to select a set of concise and focused C programs and generate their variants with and without each optimization flag[1], and measure DSE performance on these variants pairwise. If a program with an optimization enabled achieves better (or worse) performance than its counterpart without optimization, MOOSE knows that the optimization positively (or negatively) impacts the performance of the DSE, as well as reasons behind the impact.

We build our data set using the GCC test suite, which contains 3,449 C source files. Through filtering and editing, 992 were used for the benchmark (we were unable to retrofit the rest of the programs to insert symbolic variables, which means we cannot use them to test symbolic execution), with which MOOSE produced 2,978,976 binary programs (including those produced by both Clang and GCC). MOOSE found that a total of 209 GCC flags impact 176,546 binaries, and 73 Clang flags impact 54,686 binaries. Due to the variance in properties these programs have, a given flag may not have an impact on every program. Therefore, we filtered out binaries that are not affected by any compilation flags, and eventually only evaluated 7.76% binaries. We then measure two DSE techniques: symbolic exploration [8, 23] and symbolic tracing [5, 41, 45] on produced program variants. Since we have identified more optimization flags for GCC (209) than for Clang (73), our analysis focuses mainly on the GCC flags and uses Clang flags as a comparison. Our results show that among the 209 GCC flags, 43 optimizations accelerate symbolic exploration, and 9 optimizations accelerate symbolic tracing; 23 optimizations slow down symbolic exploration, and 21 slow down symbolic tracing. Among the 73 Clang flags, we identified 14 that accelerate symbolic exploration, and 24 that decelerate symbolic exploration. Similarly, we identified 7 optimizations that accelerate symbolic tracing, and 15 that decelerate symbolic tracing.

Most compiler optimizations were created to improve CPU execution performance. Nevertheless, DSE engines are different from CPUs: Their performance is largely determined by the speed of software emulation and constraint solving. We identify four main reasons why some compiler optimizations (and flags) negatively impact the performance of DSE engines: (i) some optimizations (e.g.,-fstack-protector) generate more instructions, causing more time spent during emulation; (ii) some optimizations (e.g., -fsplit-stack) create additional conditional branches, which leads to more program states; (iii) some optimizations (e.g., -fbuiltins) inline library calls into binary code, which prevents DSE engines from using optimized function summaries; (iv) some arithmetic optimizations (e.g., reciprocal optimization of divisions) destroy linearity in arithmetic expressions, which makes DSE engines generate more complex and harder-to-simplify symbolic constraints. In particular, we obtained several key findings through our study, and we list some of them below.

- **There were no suitable test suites.** We failed to find any test suites of C programs that would both take symbolic input and expose the effects of all compiler optimizations during DSE. Therefore, we created a new test suite for MOOSE by taking the GCC test cases and modifying them to accept symbolic input.
- **Symbolic exploration and tracing have different performance characteristics.** DSE is a complex procedure with many stages, such as emulation, constraint collection, constraint solving, and state forking. While symbolic exploration techniques spend a lot of time on constraint solving and state forking, symbolic tracing does not fork states and generally solves much less. We measured the impact of optimizations on both symbolic exploration and tracing, and find that most optimizations impact exploration and tracing differently.
- **Many optimizations decelerate DSE.** Optimized binaries are not necessarily friendly to DSE engines. For example, on angr, 21 optimizations decelerate symbolic exploration and 7 optimizations decelerate symbolic tracing. We observe a similar impact on different compilers (GCC and Clang) and different DSE engines (angr, Maat and SymQemu) that our study covers. Previous research did not demonstrate this finding due to the limited number of optimizations studied.
- **Compilers do not expose all optimizations as compiler flags.** We found that certain compiler optimizations, such as the reciprocal optimization of divisions, are enabled even when all optimizations are turned off (by specifying -O0). Some of these optimizations significantly impact DSE performance, but they are completely ignored in prior works.
- **Our findings apply to real-world programs.** We further verified through experiments that our findings apply to real-world programs: By applying DSE-accelerating optimizations and disabling DSE-decelerating optimizations (or by intentionally transforming the optimized programs to non-optimized ones), we observed an improvement of symbolic execution.

Contributions. This paper makes the following contributions:

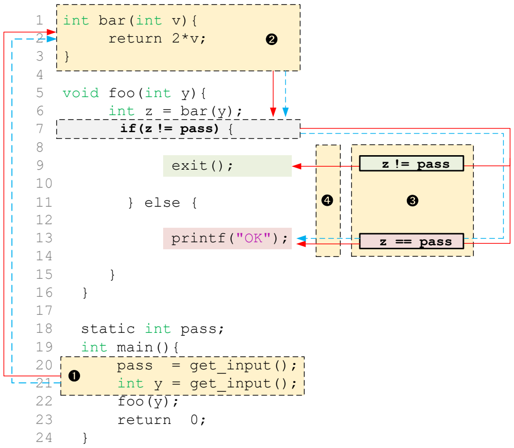
- We designed a new framework, MOOSE, to (i) generate valid test cases for DSE, and (ii) pinpoint, measure, and reason about the impacts to DSE caused by compiler optimizations.
- We studied the impact of a large number of optimization flags through automated binary program generation and pairwise comparison. Our study includes multiple compilers (GCC and Clang) and DSE engines (angr and SymQemu).
- We created a new data set using the GCC test suite with 992 test cases (which are produced from the 3,449 programs) and evaluated MOOSE with both GCC and Clang. Our analysis revealed a large set of compilation flags that positively or negatively impact DSE performance. We analyze their root causes and how we can improve DSE performance on binary code.
- We confirmed that our results help improve the performance of DSE in real settings. Through recompiling real-world programs using a specific combination of flags (where we enabled DSE-accelerating flags and disabled DSE-decelerating flags), we observed improved DSE performance.

[1] We use the term "optimization flags" loosely in this paper. Some flags are for purposes unrelated to program optimization, such as hardening or instrumenting the program. Enabling such flags may make the compiled executable larger or run slower.

2 Background

2.1 Dynamic Symbolic Execution

Dynamic symbolic execution (DSE) is a technique that systematically traverses some or all states of a program. DSE is commonly used in software testing and vulnerability discovery [7, 25, 34, 35]. Based on its purpose, DSE can be further categorized to symbolic exploration [8, 23] and symbolic tracing [5, 41, 45]. Symbolic exploration is mostly used for generating inputs that trigger a certain state, where the DSE engine takes as input an initial state, explores multiple execution paths until it reaches a specific state, and generates one or more input cases that can lead the execution into this state. Unlike symbolic exploration, symbolic tracing only traverses one path. In symbolic tracing mode, the DSE engine accepts a concrete input, and traverses the execution path as determined by the given input until the program terminates. It is usually used in hybrid fuzzing to find new seed input that would explore new branches in a program. With the simple example program shown in Figure 1, we next explain how symbolic tracing and exploration work.



```
1  int bar(int v){
2       return 2*v;
3  }
4
5  void foo(int y){
6       int z = bar(y);
7       if(z != pass) {
8
9               exit();
10
11         } else {
12
13              printf("OK");
14
15         }
16    }
17
18    static int pass;
19    int main(){
20         pass  = get_input();
21         int y = get_input();
22         foo(y);
23         return  0;
24    }
```

Figure 1: A simple example illustrating the difference between symbolic exploration and tracing. Red lines are symbolic execution paths, and blue dashed lines show the symbolic tracing path.

```
int a[1];
for (int i
{
  if(i>50){
      a[i]=8
  }else{
      a[i]=0
  }
}
```

Symbolic Tracing. During symbolic tracing, the DSE engine first takes a concrete input (assuming 6 for pass and 3 for y) in main() (Step ❶). Then it simulates the multiplication in bar() (3*2) and gets the result 6 (Step ❷). Next, the simulation reaches the if statement (line 7) and evaluates the Boolean condition z!=pass (Step ❸). After comparing pass against z, the engine will follow the branch determined by z==pass (line 13) and outputs “OK” (Step ❹). Because the DSE engine tracks how each value is derived from user input, we solve for new input assuming different conditions hold (e.g., negating z==pass, or forcing z to be 10).

Symbolic Exploration. Unlike symbolic tracing, symbolic exploration takes partial or no user input to begin with. In Step ❶, the DSE engine assigns symbolic expressions instead of concrete integers to corresponding variables (pass=α and y=γ where α and γ are unknown variables). Next, the engine simulates multiplication and assigns $2*\gamma$ to z (Step ❷). When the engine reaches the if statement (line 7), it evaluates Boolean condition $2*\gamma$!= α. Because γ and α are both symbolic, the solver determines that this condition can be both satisfiable and unsatisfiable. The DSE engine then “forks” the simulated program state into two states (where each follows a different path) and adds the Boolean condition and its negation to each state as *path constraints* (Step ❸). The first state (where $2*\gamma$!= α holds) invokes exit() while the second state (where $2*\gamma$ == α holds) outputs “OK”. Optionally, for each final state, we may use the solver to solve for a concrete initial input. For example, the solver generates input γ=3 and α=6, which leads the program to print “OK”.

2.2 Taxonomy of DSE Engines for Binary Code

DSE engines works on either source code (e.g., KLEE [13] and SymCC [34]) or binary code (e.g., angr, QSYM, and SymQemu). This paper focuses on binary-only DSE engines. To better understand how DSE engines work on binary programs, we inspect the source code of several DES engines. At a high level, performing DSE on a binary program involves three major phases: (a) **I**nitialization, where the DSE engine creates an simulated program state with a simulated environment; (b) **E**xecution, where DSE engine simulates the execution of instructions in its execution engine, forks states when necessary, and collects path predicates; (c) **T**ermination, where DSE engine performs remaining tasks (e.g., final constraint solving during tracing) and clean-ups. Next, we detail five key operations and their most relevant phases (I, E, and T).

(I) Program Loading and Initial State Creation. This step is where the DSE engine loads the binary executable and creates the initial simulated state (including registers, stack, and heap) with a simulated environment (including OS kernel, files, and file descriptors). In symbolic tracing, concrete execution traces are loaded during this step.

(E) Interpreting or JITing Instructions. When instructions use symbolic data, DSE engines cannot execute them directly on the CPU. As such, DSE engines generally fall into two categories according to how they execute these instructions: some (angr, S2E [18], and Maat [1]) interpret instructions in emulated CPUs that handle symbolic data while others (QSYM and SymQemu) just-in-time compile (JITs) handler functions into the main program. Optionally, DSE engines may first lift instructions into a side-effect-free intermediate representation (IR), and then interpret IR statements for the ease of implementation (e.g., angr uses VEX IR [4, 30], and SymQemu uses TCG). QSYM does not use any IR but instruments the binary dynamically using Pin.

(E) Accesses to Simulated Registers and Memory. DSE engines translate memory and register accesses in each instruction into accesses to their own simulated register and memory objects that are associated with each simulated program state. These simulated register and memory objects are sparse. They may support copy-on-write to reduce RAM usage, especially for symbolic exploration techniques, where many emulated states exist in memory.

(E,T) Constraint Creation and Solving. During symbolic exploration, at every conditional branch instruction, DSE engines

DSE	SExec.?	STrac.?	Style	Concrete.	Multi-Plat.?	Lang.	IR	Solver
angr	✓	✓	Inter.	Unicorn	✓	Python	VEX	Z3
S2E	✓	✓	Inter.	Qemu w/ KVM	✓	C++	LLVM	Z3
Maat	✓	✓	Inter.	N/A	✓	C++	P-Code	Z3
QSYM	✓	✗	Instr	Pin	✗	C++	N/A	Z3
SymQemu	✓	✗	JIT	Qemu	✓	C++	TCG	Z3

Table 1: Qualitative comparison between major DSE engines for binary code. SExec. refers to symbolic exploration, and STrac. refers to symbolic tracing. Inter. is means Interpretation and Instr. means Instrumentation. Multi-Plat. refers to if the engine supports analyzing binaries for more than one operating system.

may determine if both branches are satisfiable. For each satisfiable branch, the engine collects its corresponding path predicates (constraints) by creating new Boolean expressions and adding them to the constraint store associated with program states. In symbolic tracing, DSE engines do not need to test satisfiability at every condition branch because the satisfiability is determined by the concrete input. Rather, the engines may negate certain branch conditions to solve for new input that allows the traversal of a previously unsatisfiable branch.

(E) State Forking. State forking only occurs during symbolic exploration when both branches of a conditional branch instruction are satisfiable. Because we do not consider state merging in our study, once a state is forked into two, they will never be merged into one state. Note that symbolic-tracing-only solutions (QSYM and SymQemu) do not fork states.

Table 1 shows a taxonomy of several popular DSE engines for binary code. Our study covers two representative DSE engines, namely angr (which interprets) and SymQemu (which JIT compiles). To minimize discrepancies caused by solvers, we configure all engines to use the Z3 solver [29].

2.3 Compiler Optimizations

Compiler optimizations are sequences of program transformations that convert a program into another semantically equivalent but (usually) more efficient program. Compiler optimizations are generally controlled by compilation flags. For example, the compiler flag `-funroll-loops` enables the loop unrolling transformation when compiling the program.

Optimization levels (e.g., `-O0`, `-O1`, and `-O2`) are special compiler flags that represent combinations of compiler optimizations. For example, `-Os` enables the flags for decreasing executable sizes. Surprisingly, optimization levels may impact other flags in unexpected ways: Some flags only take effect under a specific optimization level, and some optimizations are only controllable by optimization levels. Typically, the order of compiler optimizations (as provided to the compiler) does not impact the generated binary unless some flags conflict (e.g., when `-flto` and `-fno-lto` are both specified).

3 Design of MOOSE

MOOSE automatically recognizes compilation flags that impact DSE performance on binary programs. Figure 2 illustrates three high-level components of MOOSE:

- **Test Case Generator** (§3.1), which takes the source files from the GCC test suite and produces transformed source files that accept user inputs.
- **Binary Generator** (§3.2), which takes generated source files as input, compiles them, and identifies binary programs that are affected by specific compilation flags.
- **Performance Evaluator** (§3.3), which performs DSE (symbolic exploration and symbolic tracing) on compiled binaries and measures their performance.

3.1 Test Case Generator

To evaluate the performance impact of compiler optimizations on dynamic symbolic execution (DSE), we require a dataset of various source files. Two main challenges arise in this process, which are that each test program must be small and simple and take user input and use it in at least one branch condition. To address these challenges, we synthesized a new dataset based on compiler test suites, which are inherently concise and simple, and developed an approach to automatically transform them into programs that take user input. Our approach includes two steps:

(I) Scanning and replacing variables. For source files that contain only concrete values, we must find all concrete variables and replace them with variables that take user input. First, we scan the source files and use regular expressions to find all variable-defining statements (e.g., `int a;`) and assignments (e.g., `a=1;`). Next, we create a new assignment statement that reads user input from `stdin` (whose length is determined by the type of the original variable in the statement). Finally, we insert the newly crafted statement after the variable-defining statement or the assignment.

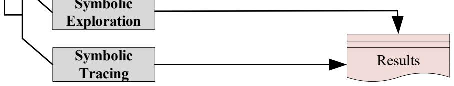
(II) Verification. Although our source modification does not create new compilation errors, the resulting programs may still not receive user input. Therefore, we perform symbolic exploration on newly generated binaries to examine if it creates symbolic variables and symbolic branch conditions as expected. We discard any programs that do not generate any symbolic branch conditions during symbolic exploration and use the remaining programs for Binary Generator.

3.2 Binary Generator

Binary Generator takes the source files generated in the previous step as input and produces executables with different compiler optimizations enabled. We detail our approach next.

(I) Selecting compilation flags. We collect a set of valid compiler flags by parsing the official GCC and Clang documentation pages on compiler flags and extracting all optimizations that the compilers support. We exclude compiler flags that do not work for C code by compiling a simple C program with each flag once. We remove a flag from our collection if we observe any compilation errors.

(II) Compiling binaries and building the dataset. We then pair each program with the optimizations that influence it. To test whether an optimization has an effect on a source program, we generate two programs for each source file, where one program has the optimization enabled and the other has the optimization disabled, and compare their bytes. If the metadata, code, and data



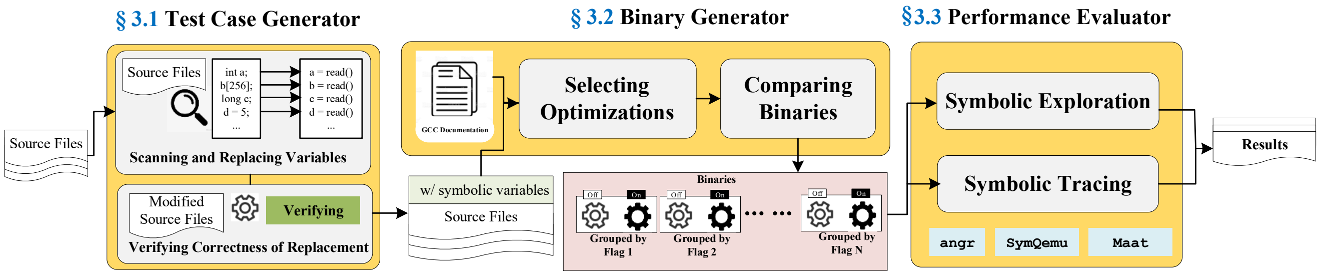


Figure 2: Major components and the workflow of MOOSE.

sections of two binaries are equivalent at the byte level, then the optimization has no effect on this program. Once this step finishes, we curate a large set of programs with all compiler flags that have an effect on each program.

(III) Generating seed input for symbolic tracing. Symbolic tracing requires seed input, and different programs require input of varying lengths. Because Test Case Generator generates all statements that consume user input, we acquire from there the length of input that each program expects and generate a seed input for each test program.

3.3 Performance Evaluator

Because symbolic exploration and symbolic tracing have vastly different natures in performance, we measure them separately. Using time as the only metric can be noisy (because solvers work in a black-box and non-deterministic manner [21]) and does not help us understand the root causes behind. Therefore, we measure the raw numbers of several metrics for each key operation.

- **Interpreting or JITing instructions.** We record both the total elapsed time and the number of executed instructions or IR statements (if IRs are used).
- **Accesses to simulated registers and memory.** We record the number and the total elapsed time of all accesses.
- **Constraint creation and solving.** We record the time elapsed to solve symbolic constraints, the total number of solving attempts, and the total number of constraints, as well as their depths (which is an approximate factor for constraint complexity).
- **State forking.** We record the number of dead ended paths (i.e., terminated) and the time spent in forking states (only for symbolic exploration).

Next, we introduce how we instrument each DSE engine.

angr. angr supports both symbolic exploration and tracing. For the former, we use angr's depth-first search (DFS) exploration, which explores all possible paths of a program until reaching time or path limits. For the latter, we use angr's tracer exploration technique (`tracer.Tracer`), which follows the execution traces using seed input. angr does not support profiling, so we customized angr to record profiling results.

Maat. Maat supports both symbolic exploration and symbolic tracing, and provides rich interfaces for customization. For example, we can collect the number of IO operations by implementing callbacks for `EVENT.REG_R` and `EVENT.REG_W`, and the number of paths by implementing the callback `EVENT.PATH`.

SymQemu. SymQemu only supports symbolic tracing. It uses QSYM as its backend, which supports recording the solving and execution time. We further customize its pintool component to collect profiling results, e.g., number of IO operations. Additionally, we modify SymQemu's TCG translation component to record its IR translation time.

4 Evaluation

4.1 Experiment Setup

Data set. We built a data set using 3,449 C source files from the GCC test suite. This dataset was chosen to ascertain the effects of each compiler flag. We consider this dataset representative, given its purpose of evaluating compiler flags, aligning with our goal of identifying flags impacting DSE. To demonstrate the applicability of our findings to real-world programs, we created a secondary data set comprising six programs and their seed input based on a benchmark used by `Ferry` [46]. We collected 218 compiler flags from the GCC documentation, and 211 flags from the Clang documentation. Our system generated 992 source files that use user input. We produced 2,978,976 binary programs under seven optimization levels (`-O0`,`-O1`,`-O2`,`-O3`,`-Os`,`-Og`, and `-Ofast`), by compiling every flag-file combination under all seven optimization levels, as some flags only take effect when they are specified in tandem with certain optimization levels. Among all binaries produced by GCC, 176,546 are affected by 209 optimizations. Additionally, 54, 686 clang-produced binaries are affected by 73 flags.

Testing environment. We use GCC `9.3.0` and `Clang 10.0.0`. All experiments were conducted on two servers, each equipped with a 16-core Intel(R) Core(TM) i7-10700 CPU at 2.90GHz and 32 GB of RAM, running Ubuntu 21.04. For symbolic execution, we set a timeout of 15 minutes and terminated programs that did not complete within this time frame. Results of timed-out runs were still recorded.

4.2 Experiment Results

RQ1. How do compiler optimizations impact the performance of symbolic exploration and symbolic tracing?

Method. We run DSE using angr on all GCC-compiled binaries and record their running time. For each GCC optimization, (i) we run the experiments five times for each pair of the binaries (i.e., one

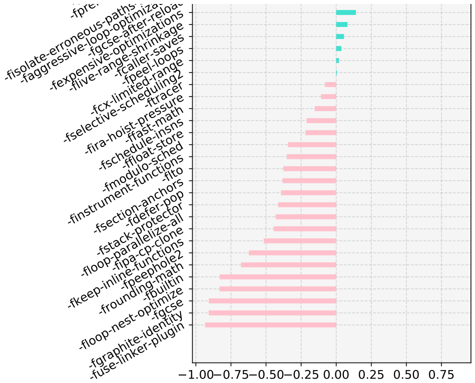


Figure 3: Distribution of performance impacts of flags on symbolic exploration (angr).

compiled with the flag and the other complied without) to reduce the impact of noises. (ii) We filter out the optimizations that do not affect symbolic execution. (iii) For each GCC optimization that affects symbolic execution, we need to understand whether the performance impact is positive or negative. Thus, for each pair of binaries, we subtract the symbolic execution time of the binary with the optimization enabled (T_e) from the symbolic execution time of the binary with the optimization disabled (T_d). The difference ($T_d - T_e$) is either positive (i.e., the optimization accelerates symbolic execution) or negative (i.e., the optimization decelerates symbolic execution); (iv) we then use the difference to divide the larger time cost ($(T_d - T_e)/max(T_d, T_e)$) to normalize the result to [-1, 1]. (v) Finally, we take the mean value of all normalized differences as the final result.

Results. Figure 3 and Figure 4 visualize the overall results. Overall, 43 (67.19%) optimizations accelerate symbolic exploration, while 21 (32.81%) optimizations decelerate symbolic exploration. Meanwhile, 9 (56.25%) optimizations accelerate symbolic tracing, while 7 (43.75%) optimizations decelerate symbolic tracing. We further found that optimizations that accelerate symbolic exploration or tracing generally produce smaller binaries or binaries with simpler logic. For example, `-fcx-limited-range` (which accelerates all binaries) avoids redundant type checking by removing certain instructions. Optimizations that decelerate symbolic exploration or tracing usually introduce more instructions. For example, `-finstrument-functions` (which decelerates all binaries) adds instructions for instrumenting programs. We also observed that when compared with the number of optimizations that affect symbolic exploration, the number of optimizations that affect the symbolic tracing is much smaller. This is because symbolic exploration explores many paths of a program, while symbolic tracing only traverses *one* path that the user input determines. During tracing, the given input has a lower chance to trigger specific paths where the optimization takes effect. If the program executes a path that is not impacted by the optimization, it is likely that no differences can be observed across multiple runs.

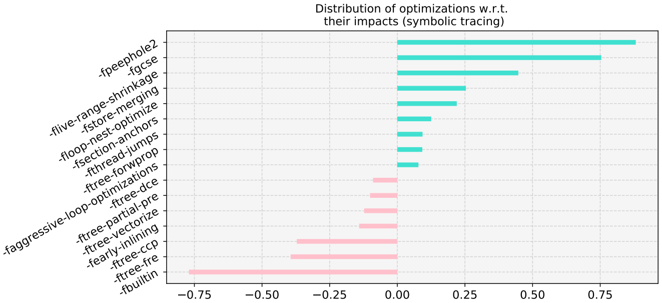


Figure 4: Distribution of the impacts of flags on symbolic tracing (angr).

RQ2. Does each compiler optimization similarly impact the performance of symbolic exploration and tracing?

Method. For each GCC optimization, we compare its performance impacts on symbolic exploration and symbolic tracing. Then we split all optimizations into seven groups: (i) optimizations that accelerate both exploration and tracing (①); (ii) optimizations that only accelerate exploration (②); (iii) optimizations that accelerate exploration but decelerate tracing (③); (iv) optimizations that only accelerate tracing (④); (v) optimizations that only decelerate tracing (⑤); (vi) optimizations that decelerate tracing but accelerate exploration (⑥); and (vii) optimizations that only decelerate exploration (⑦). We do not observe any optimizations that decelerate both execution and tracing (⑧).

Results. As shown in Table 2, only five optimizations accelerate both symbolic exploration and tracing, while all other optimizations have contradictory impacts on symbolic exploration and tracing. We believe that this discrepancy of impacts is caused by the different natures of exploration and tracing: As discussed, symbolic tracing only traverses one path that the user input determines. Although an optimization may accelerate or decelerate DSE on a binary, its effect may not be prominent on all paths. Symbolic exploration explores many paths of a program and has a higher chance to cover paths that are actually affected by an optimization.

① -ftree-ccp	① -fearly-inlining	① -ftree-dce	① -faggressive-loop-optimizations	① -flive-range-shrinkage	② -ftree-sra
② -ftree-pre	② -fhosted	② -ftree-vrp	② -ftree-switch-conversion	② -ftree-loop-optimize	② -fgcse-after-reload
② -ftree-loop-vectorize	② -fmerge-all-constants	② -fpeel-loops	② -fipa-pta	② -fsplit-loops	② -fgcse-lm
② -fprefetch-loop-arrays	② -fgcse-sm	② -ftree-loop-if-convert	② -freciprocal-math	② -finline-small-functions	② -fexpensive-optimizations
② -fbranch-probabilities	② -ftree-slsr	② -funroll-loops	② -ftree-loop-distribution	② -fcaller-saves	② -fcx-limited-range
② -foptimize-strlen	② -ftree-copy-prop	② -fivopts	② -ftree-dse	② -finline-functions-called-once	② -fdce
② -funconstrained-commons	② -fselective-scheduling2	② -findirect-inlining	② -fisolate-erroneous-paths-attribute	③ -ftree-fre	③ -ftree-vectorize
③ -ftree-forwprop	④ -fthread-jumps	④ -fstore-merging	⑤ -ftree-partial-pre	⑥ -fsection-anchors	⑥ -floop-nest-optimize
				⑦ -fuse-linker-plugin	⑦ -fkeep-inline-functions
				⑦ -fira-hoist-pressure	⑦ -fdefer-pop

exploration and tracing. Seven cell colors (green, lime,

exploration and symbolic tracing on the programs produced by gcc, for each optimization, we compare its impacts on symbolic exploration and symbolic tracing, and the way of our categorization is the same as that introduced in *RQ1* and *RQ2*.

Results. Figure 5 and Figure 6 show the results. We identified 14 optimizations (37.83%) that accelerate symbolic exploration and 23 optimizations (62.17%) that decelerate symbolic exploration. Similarly, we have identified 7 optimizations (31.82%) that accelerate symbolic tracing, and 15 optimizations (62.18%) that decelerate symbolic tracing. Our results indicate that optimizations (which may be implemented differently) provided by GCC and Clang generally have a similar impact on the performance of DSE. For example, -finline-functions (this optimization has the same name in both Clang and GCC) makes all functions inline even if they are not declared inline. This optimization accelerates symbolic exploration, and the impacts are reflected by both Clang and GCC.

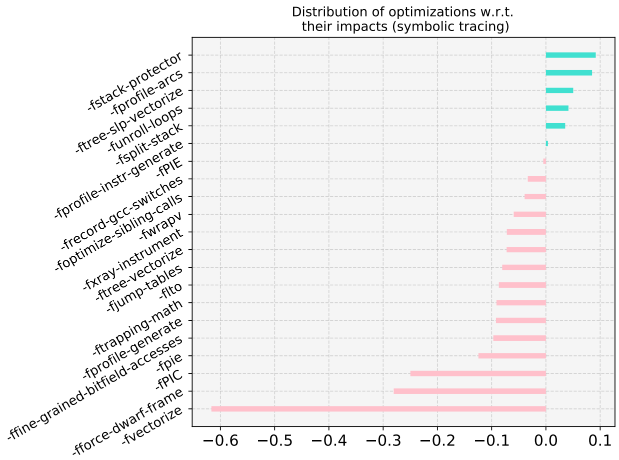
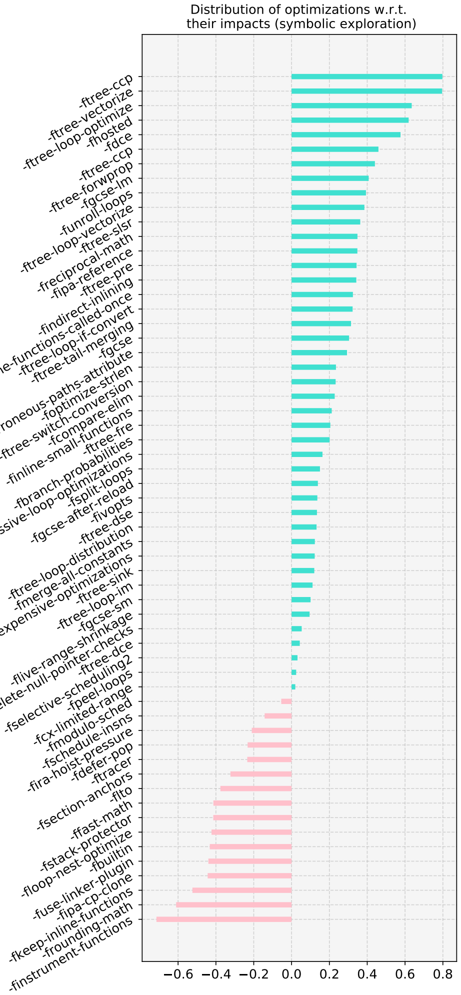


Figure 6: Distribution of the impacts of flags on symbolic tracing (Clang).

RQ4. Does each compiler optimization similarly impact the performance of different DSE engines?

Method. To answer this research question, we include two DSE engines, Maat and SymQemu, in addition to angr. We use the same approach in RQ1 to measure the performance impact of optimizations on Maat for both symbolic exploration and tracing, and compare the results with our findings in RQ1. Because SymQemu does not support symbolic exploration, we evaluate symbolic tracing using SymQemu and then compare our findings against symbolic tracing results acquired using angr and Maat.

Symbolic Exploration (angr vs Maat). Figure 7 shows the impacts of flags on symbolic exploration using Maat. We identified 43 optimizations that accelerate and 16 that decelerate symbolic exploration. By comparing the results against Figure 3, we found 52 flags similarly impact the performance of symbolic exploration for both angr and Maat, and the orders of these flags in both figures are close. Despite major differences between the designs and implementation



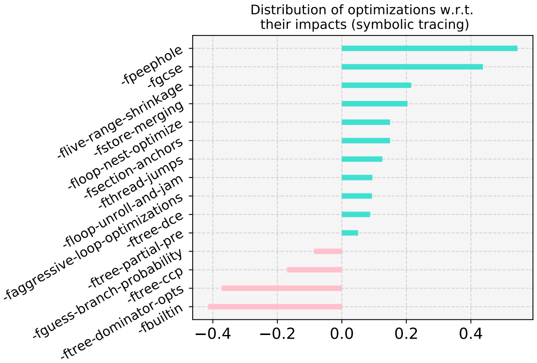


Figure 8: Distribution of the impacts of flags on symbolic tracing (Maat).

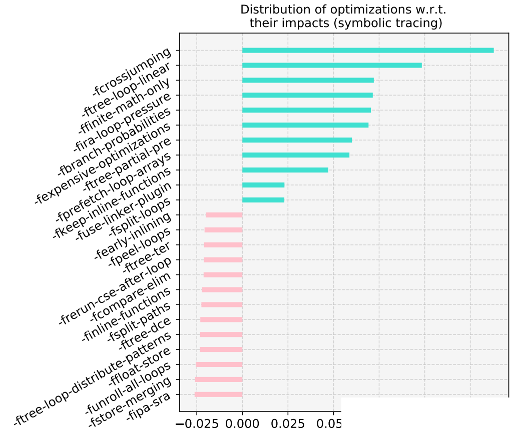


Figure 9: Distribution of the impact
tracing (SymQemu).

(II) Interpreting or JITing Instruction
of instructions in a binary and the num
ments are proportional. Some optimizat
of relatively complex arithmetic operatio
fewer instructions. For example, `-frecip`
reciprocal of a value to be used instead
if this enables optimizations” [2]. Listing
code of `3.0 / (x * 2.0) + 1.0`. Listing
code of the same expression when `-freci`
(the corresponding C representation is `3.`

This optimization simplifies the divisio
the divisor (2.0) with its reciprocal (0.5), t
dividend. It reduces the number of instru
terpretation. Additionally, a set of compile
DSE by removing redundant instructions.
removes unnecessary copy operations. Du
optimizations decelerate DSE by introducing redundant code. This is because one optimization is usually enabled together with other optimizations during compilation (e.g., `-O3` involves tens of flags). While one optimization may create duplicated or redundant instructions, other optimizations that are usually enabled simultaneously may become effective to remove the redundancy. Listing 2 shows an example where `-ftracer` performs tail duplication that duplicates branch tails. Other optimizations, e.g., `-ftree-tail-merge`, will recognize these duplicated tails and re-merge or remove them if possible. However, if we only enable `-ftracer` (without its peers), it negatively impacts DSE performance.

(III) State Forking. Symbolic exploration usually suffers from path explosion. The root cause is that the number of feasible states in a program grows exponentially. Compiler optimizations may create or eliminate program paths, leading to fewer explored paths and forked states. Some optimizations reduce the number of program paths by removing conditional statements, which are commonly used for checks. Examples of such optimizations include `-ftree-vrp` and `-fdelete-null-pointer-checks`. `-ftree-vrp` removes unnecessary range checks (e.g., array bound checks when an access to the array is always in-bound). In Listing 3, the optimization

```
0010117f f2 0f 10        MOVSD      XMM0,qword ptr [DAT_00102008] ; x
          05 81 0e
          00 00
00101187 f2 0f 59 c8     MULSD      XMM1,XMM0 ; 2.0 * x
0010118b f2 0f 10        MOVSD      XMM0,qword ptr [DAT_00102010] ; 3.0
         05 7d 0e
         00 00
00101193 f2 0f 5
00101197 66 0f 2
0010119b f2 0f 1
         05 75 0
         00 00
001011a3 f2 0f 5
```

```
0010117f f2 0f 1
         05 81 0
         00 00
```

```
int i = 0;
while (i < 100) {
 x = y+9;
 a[i] = 2 * x;
 ++i;
}
```

```
int i = 0;
x = y+9;
while (i < 100) {
```

```
int bar
  retur
}

void fo
  int z
  if (z

  }else
```

```
int bar(int v) {
  return 2*v;
}

void foo (int y) {
  int z = bar(y);
  if (z != pass) {
        exit();
  }else{
        printf("OK");
  }
}

static int pass;
int main() {
  pass = get_input();
  int y = get_input();
  foo(y);
  return 0;
}
```

```
int i = 0;
while (i < 100) {
 x = y+9;
 a[i] = 2 * x;
 ++i;
}
```

```
int i = 0;
x = y+9;
while (i < 100) {
 a[i] = 2 * x;
 ++i;
}
```

```
int i = 0;
while (i < 100) {
 x = y+9;
 a[i] = 2 * x;
 ++i;
}
```

```
int i = 0;
x = y+9;
while (i < 100) {
 a[i] = 2 * x;
 ++i;
}
```

```
int
for
{

}

int
```

```
1 for (i
2   if(i
3     fc
4 }
```

```
1 for(
2   foo(
3 }
```

(a)

```
1 for (i = 4; i < 100; i+= 1){
2   if(i>3)
3     foo(i);
4 }
```

```
1 for (i = 4; i < 100; i+= 1){
2   foo(i);
3 }
```

(b) w `-ftree-vrp`

Listing 3: The C code in (a) is transformed into the code in (b) using compiler optimization `-ftree-vrp`, leading to fewer basic blocks and program states during DSE.

removes Line 2, since the check on whether `i` is greater than 4 is redundant. While this optimization does not decrease the number of program paths (since `i` will never be less than or equal to 3), it does lead to fewer program states being created, which accelerates both symbolic exploration and symbolic tracing.

Some optimizations increase the number of program states during DSE. For example, `-fsplit-stack` adds logic to automatically grow the size of the stack as needed. Similarly, when `-finstrument-functions` is enabled, GCC will instrument the binary with function calls at both the entries and exits of the functions in the binary. The calls inserted will invoke profiling functions, which monitor the execution frequency and duration of the instrumented functions. The use of either optimization will result in an increase in the number of program states, which decelerates both symbolic exploration and tracing.

(IV) Constraint Creation and Solving. Some optimizations perform *strength reduction*, which essentially replaces arithmetic operations that are slower (on CPUs) with those that are faster (on CPUs)

```
+9;
 (i < 100) {
 = 2 * x;
```

```
          00 00
00101193 f2 0f 5e c1       DIVSD       XMM0,XMM1  ; 3.0/(2.0 * x)
00101197 66 0f 28 c8       MOVAPD      XMM1,XMM0
0010119b f2 0f 10          MOVSD       XMM0,qword ptr [DAT_00102018]  ; 1.0
         05 75 0e
         00 00
001011a3 f2 0f 58 c1       ADDSD       XMM0,XMM1  ; 3.0/(2.0 * x) + 1.0
```

```
0010117f f2 0f 10          MOVSD       XMM0,qword ptr [DAT_00102008]  ; x
         05 81 0e
         00 00
00101193 f2 0f 5e c1       DIVSD       XMM0,XMM1  ; 3.0*0.5/x=1.5/x
00101197 66 0f 28 c8       MOVAPD      XMM1,XMM0
0010119b f2 0f 10          MOVSD       XMM0,qword ptr [DAT_00102018]  ; 1.0
         05 75 0e
         00 00
001011a3 f2 0f 58 c1       ADDSD       XMM0,XMM1  ; 1.5/x + 1.0
```

```
AT_00102008] ; x

.5/x=1.5/x

AT_00102018] ; 1

 + 1.0

0040114d
00401152
00401155
00401158
0040115a
0040115d
00401160
00401162
00401166
```

```
long lVar1;
byte *pbVar2;
byte *pbVar3;
bool in_CF;
bool in_ZF;

lVar1 = 5;
pbVar2 = *(byte **)(lParm2 + 8);
pbVar3 = &DAT_00102004; //'BCDE'
do {
   if (lVar1 == 0) break;
   lVar1 = lVar1 + -1;
   in_CF = *pbVar2 < *pbVar3;
   in_ZF = *pbVar2 == *pbVar3;
   pbVar2 = pbVar2 + 1;
   pbVar3 = pbVar3 + 1;
 } while (in_ZF);
 if ((!in_CF && !in_ZF) == in_CF) {
   printf("you win");
 }
```

(b) With -fbuilt-in.

Listing 5: Ghidra's decompilation output for a C program, with and without `-fbuilt-in`, affects `strcmp` symbolic string length limits, impacting DSE performance.

without altering the semantics of the program. While the impact of faster arithmetic operations is less evident in DSE engines than on CPUs, sometimes strength reduction leads to fewer instructions, which benefits DSE. For example, `-ftree-slsr` recognizes multiplications used in a program and replaces them with cheaper arithmetic operations if possible. As shown in Listing 4, after optimization, lines 2 and 3 reuse existing results, and two additions are eliminated. For DSE engines, fewer arithmetic operations usually result in better performance. Note that in this case, when `b` is a symbolic expression, the final symbolic expression in `t3` remains unchanged.

Having fewer instructions does not always result in better performance for dynamic symbolic execution (DSE). Some optimizations can actually slow down DSE by generating less optimal symbolic constraints. Modern DSE engines rely on function summaries to handle missing library functions. These function summaries usually run faster than corresponding binary code in emulation, as they skip the emulation loop completely, and implement semantics-specific limits to give DSE engines more control over symbolic data. For example, angr provides Python-based function summaries (`SimProcedures` in angr) to replace library functions such as `strlen` and `strcmp`.

When the optimization `-fbuilt-in` is enabled, GCC will rewrite the program to inline these common functions, including `strlen` and `strcmp` (see Listing 5). This inlining prevents DSE from using the `strcmp` function summary that angr provides. When the data passed to `strcmp` is symbolic, both symbolic exploration and symbolic tracing will generate more complex constraints with inlined `strcmp`, which slows down constraint solving. In our experiments, the inlined version of `strcmp` is always much slower than its function summary.

```
00401161 89 c8             MOV         EAX,ECX
00401163 99                CDQ
00401164 b9 03 00          MOV         ECX,0x3
         00 00
00401169 f7 f9             IDIV        ECX
```

```
0040114d 89 c8             MOV         EAX,ECX
00401152 6b c0 56          IMUL        EAX,EAX,0x56
00401155 0f b7 c0          MOVZX       EAX,AX
00401158 89 c1             MOV         ECX,EAX
0040115a c1 e9 0f          SHR         ECX,0xf
0040115d c1 e8 08          SHR         EAX,0x8
00401160 00 c8             ADD         AL,CL
00401162 48 0f be f8       MOVSX       RDI,AL
00401166 31 c0             XOR         EAX,EAX
```

(b) Assembly with division optimization

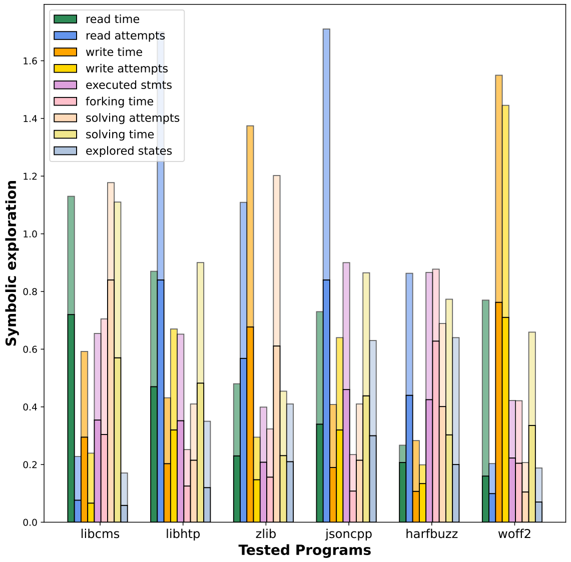
Listing 6: Assembly code in (a) is transformed into the code in (b) when division optimization is enabled.

Interestingly, during our study, we found that some optimizations are enabled even in `-O0` but negatively impact DSE performance. Listing 6 shows an example where the division optimization converts a division instruction into multiple instructions involving multiplications and bit shifting. This is faster for execution on CPUs [43]. However, since this optimization breaks the linearity of division (bit shifting is less optimizable in constraint solvers), angr will generate more complex symbolic constraints, leading to much longer solving time in Z3. In our experiment, we tested a program with a simple `if (a % 100 == 20)` condition where `a` was an int symbolic variable. The div-optimized version spent 20x more time in solving than the non-optimized version (21 seconds versus 1 second). This is an unexpected example where seemingly unoptimized binaries are optimized for CPU-execution, which significantly hampers their DSE performance.

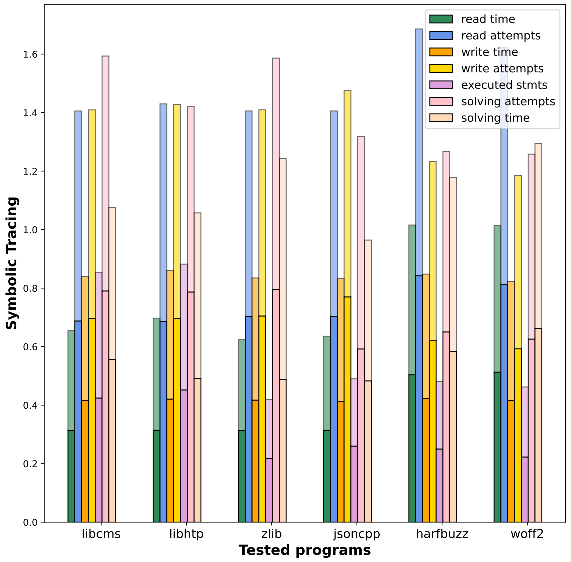
RQ6. Do our findings apply to real-world programs for accelerating symbolic exploration or symbolic tracing?

The ultimate goal of our project is to improve binary DSE performance by taking advantage of compiler optimizations. Having understood the impacts, we next explore how to apply our results to real-world binaries. There are multiple ways to apply our results to real-world programs: We can preemptively enable or disable compiler optimizations on the source files. Or, we can retroactively modify the binaries to optimize DSE performance. For example, because division optimization decelerates DSE, we can recognize such patterns (e.g., bit shifting) in binary code and transform the binary back to its original, non-optimized form. We designed two experiments to validate these two approaches accordingly.

Experiment (I) – Applying optimizations. We now directly enable or disable compiler optimizations when building source code



(a)



(b)

Figure 10: The performance impact on symbolic exploration (a) and symbolic tracing (b), benchmarked on real-world programs. The upper parts of each bar are profiling results on baseline binaries, while the lower parts are profiling results on optimized binaries. Because symbolic tracing only involves one path, we do not report forking time or numbers of explored states.

and observe the impacts on DSE performance. We used a data set of real-world programs from Ferry [46]. Based on prior experiment results, we identified two sets of GCC compiler flags: One set accelerates both symbolic exploration and tracing, and the other set decelerates symbolic exploration or tracing. Then we attempted to build all the programs with these flags, and six programs were successfully compiled (other programs were not able to be built). We also built baseline versions of these six programs using their default configurations. Then for each pair of (baseline-optimized) binaries, we performed symbolic exploration and tracing on programs for two hours before collecting results.

Results. We present the results in Figure 10 (a) and Figure 10 (b) and normalize them in the range of 0 and 1 for better visualization. Figure 10 (a) shows that our combinations of compilation flags accelerate symbolic exploration by allowing DSE to execute more statements and spend less time in forking states. Other metrics, such as the number of explored states, the number of solving attempts, and the solving time, are similar. It is also worth noting that some optimizations, such as the division optimization, cannot be disabled by compiler flags. Because running symbolic exploration on more code may lead to discrepancies in other metrics (e.g., number of register reads, constraints solved, etc.), it is difficult to make a fair comparison. Therefore, we only use the number of executed statements to quantify the improvement. On average, symbolic exploration on optimized binaries executes 8.04% more statements than on baseline binaries. For symbolic tracing, since DSE only executes a single path determined by user input, the solving time reliably reflects the improvement. On average, symbolic tracing on optimized binaries spends 7.94% less time in constraint solving than on baseline binaries. In conclusion, our experiment results indicate that our findings do apply to real-world programs to accelerate DSE.

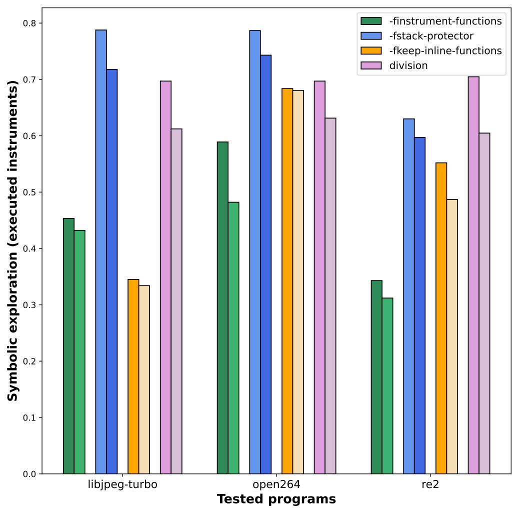
Experiment (II) – Transforming binaries. We now aim to transform the optimized binaries to their non-optimized form and observe the impacts on symbolic execution. We selected the dataset used in Ferry [46], but due to the manual modification required, we only chose three small programs (less than 2 MB) for this experiment. The optimizations selected for verification include `-finstrument-functions`, `-fstack-protector`, `-fkeep-inline-functions` and division optimizations. These optimizations are known to slow down symbolic execution and were removed from the binaries. Specifically, the first three optimizations add extra instructions, which can be de-optimized by removing those added instructions. Division optimization converts the division opcode to bit shifting and multiplications, therefore, these instructions were converted back to the division opcode. We also compiled the baseline versions of these three programs using their default configurations. For each pair of (baseline-optimized) binaries, we collected results after performing symbolic exploration and tracing on the program for two hours. Based on our previous experiment, we determined that executed statements and solving time are good metrics to evaluate the performance of symbolic execution. Therefore, in this experiment, we only selected these two metrics to demonstrate the impacts and used the same method to visualize.

Results. We present the results in Figure 11 (a) and Figure 11 (b). For each pair of the program, the left part is the optimized program, while the right part is the baseline. It can be observed that for `-finstrument-functions`, symbolic exploration on optimized binaries executes 12.97% more statements than on baseline binaries. For `-fstack-protector`, the value becomes 7.12%, 5.27% for `-fkeep-inline-functions` and 13.55% for the division optimization. For symbolic tracing, for `-finstrument-functions`, symbolic tracing on optimized binaries spends 1.78% less time solving constraints than in baseline binaries. 1.44% for `-fstack-protector`, 0.42% for `-fkeep-inline-functions` and 5.02% for division optimization. Our experiments confirmed that the findings can be applied to real-world programs.

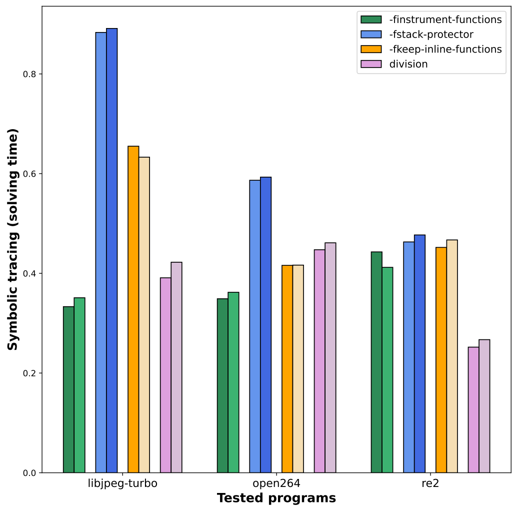
5 Flags with Negligible Impacts

We also observed that some optimizations have a minimal impact on the performance of dynamic symbolic execution (DSE). These optimizations do not affect symbolic execution when analyzing the results of different runs. Some of the optimizations that have a negligible impact on DSE include:

- **Aliasing.** Some optimizations increase or decrease the amount of aliasing in assembly code. These optimizations typically have minimal effects on performance. For example, the `-frename-registers` flag causes the generated binary code to use a wider range of registers (to avoid false dependencies in code scheduling), but does not show any observable impact on performance in our experiments.
- **Block or function reordering.** Some compiler optimizations reorder basic blocks or functions to improve code locality. Programs with better code locality generally run faster on real CPUs due to caching. Similarly, some optimizations swap code fragments to improve cache performance. For example, `-floop-interchange` interchanges instructions within a loop. However, this impact on CPU cache is typically eliminated when DSE engines lift, instrument, or JIT new code for symbolic execution.
- **Value signedness.** Some optimizations control how the compilers handle signed numbers. For example, when `-fsigned-zeros` is enabled, `+0.0` and `-0.0` are treated as distinct floating-point values. As such, `+0*x` will be treated as a positive number, while `-0*x` will be treated as a negative number. When some branch conditions depend on the signs of these expressions, the control flow of a program may change according to the optimization value. A similar compiler optimization is `-fsigned-chars`.
- **Non-program transforming compiler flags.** Some optimizations do not directly transform the program. Instead, these compiler optimizations perform compile-time checks, and may cause the compiler to throw errors and terminate the compiling process if certain checks fail. These compiler flags are not optimizations at all. For example, `-fstrict-aliasing` requires that if a variable of one type is assigned to an address, variables of different types cannot be reassigned to the same address.



(a)



(b)

Figure 11: The performance on symbolic exploration (a) and symbolic tracing (b), benchmarked on real-world programs.

6 Discussion

Threats to validity. Multiple issues could influence the conclusions we draw in our study. First, we did not exhaust all optimizations that are not controlled by optimization flags. For example, we use division optimization in our paper, which has negative impacts on DSE. This optimization is enabled by default, and is not controlled by the optimization flags. We believe that there are some other optimizations that are not controlled by the compiler flags. Second, our data set is not large enough to cover all optimizations. For example, there are at least 11 GCC flags that are not covered by this data set of programs. We investigate the reason that why those flags are not covered. The reasons for this include non-compatibility of operating systems and languages, for example, certain flags may only be intended for use with C++, and not for C. Third, in our experiments, the impacts of some "less-impactful" optimizations may be shadowed other "more-impactful" optimizations. Since those "more-impactful" optimizations may not be controlled by the compiler flags, their impacts may not be canceled by configurations.

Impact of our study. After understanding how the optimizations affect the performance of DSE, we can use these findings to optimize DSE. We can select the optimizations that negatively impact (especially those that are widely used in real-world binaries) the performance of DSE and then revert them. For example, because it is known that division optimization has a negative impact on DSE, we can recognize such patterns in binary code (e.g., bit shifting and multiplications) and transform the binary back to its original, non-optimized form. However, since there are more than 200 optimizations in GCC alone, significant manual efforts may be required to recognize all the patterns and create a one-to-one transformation relationship between them. Therefore, we leave this for future work. Another direct impact of our findings is that the results can be applied to accelerate source-code based symbolic execution (e.g., SymCC), where the optimizations can be configured when performing symbolic tracing.

7 Related Work

Compiler Optimizations and Security. Our study is unique, being the first to comprehensively measure the potential impact of compiler flags on DSE. Our tool, MOOSE, assesses flag impact automatically, a novel approach. Despite a few prior attempts have been made to study the impacts of compiler optimizations on DSE, their solutions are subject to systematicness and their experimental results lack of comprehensiveness. -Overify [44] investigated the impacts of optimization levels on program verification including DSE. However, it did not study the impacts of each optimization, and therefore, their result could only provide limited information regarding how each optimization affect the performance of symbolic execution. There are also works [12, 21] that attempt to evaluate the impacts of each optimization, but since they checked the impact manually, very limited optimizations were discussed (e.g., 33 optimizations out of more than 200 flags). It is also not clear why these optimizations are chosen in their experiments.More recently, LEO[16] transformed the produced programs into semantic-preserving programs using machine leaning, where the authors trained model on the program produced with different optimizations, and used the learnt rules to guide their transformations. The drawbacks of their methods are also obvious: the performance of machine learning is highly related to the dataset (the produced programs), and may fail to achieve similar efforts when the dataset is changed. In addition to the impacts on DSE, the security community also studied how compiler optimizations affect the function signatures of programs [28], and the transformation caused by the compiler optimizations may significantly influence results of similarity analysis [15] and malware detection [20], so that efforts have been made to recover the function signatures (e.g., [9, 19, 40]). Our work is symbolic execution specific, and their approaches cannot be applied to solve our research problems.

Dynamic Symbolic Execution. While many dynamic symbolic execution (DSE) engines exist and are actively used (e.g., KLEE [13], angr [41], QSym [45], SymCC [34], SymQemu [35], and Maat [1]), DSE is known for its sub-par performance and unsatisfactory scalability. To improve the scalability of DSE, research has been conducted on combining fuzz testing and DSE [10, 14, 22, 31, 42, 45] by feeding heuristically mutated inputs based on the formulas produced by symbolic execution. This approach can achieve more code coverage [11, 24]. However, symbolic execution guided fuzzing is limited by the size of the input constraints produced by symbolic execution. To address this limitation, VUzzer [37], FairFuzz [26], Angora [17] and Steelix [27] have introduced program analysis (e.g., static and dynamic program analysis) to guide input mutation. For example, Angora [17] uses taint analysis to solve path constraints, and Steelix [27] leverages observations made in the comparison progress to drive effective mutation. Previous works also have aimed to accelerate symbolic execution through code transformation, such as T-Fuzz [32] and [33], which bypass time-consuming paths for better performance. Our work is distinct as we systematically study the impact of compiler optimizations on DSE performance. Our results can inspire future research on semantic-preserving binary transformations to improve DSE performance.

8 Conclusion

Compiler optimizations can affect the performance of Dynamic Symbolic Execution (DSE), but current research does not systematically profile their impact on DSE. In this paper, we systematically study the impact of compiler optimizations on the performance of DSE, including both symbolic exploration and tracing, covering two compilers and three DSE engines. Our profiling results on small unit test-style binaries as well as real-world binaries show that significant performance gains can be achieved in binary-based DSE tasks by applying or removing certain compiler optimizations. We hope that our findings will provide insight into this research problem and guide future research on accelerating DSE on binaries.

Acknowledgement

We thank the anonymous reviewers for their insightful feedback. This research was supported in part by Army Research Office (ARO) award W911NF2110081, Defense Advanced Research Projects Agency (DARPA) awards N6600120C4020 and N6600122C4026, Advanced Research Projects Agency for Health (ARPA-H) Contract No. SP47012 3C0074, and National Science Foundation (NSF) awards 2146568 and 2232915. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of ARO, DARPA, ARPA-H, or NSF.

Code Availability

The source code of MOOSE is available at https://github.com/OSUSecLab/MOOSE.

References

[1] Maat. https://maat.re/.

[2] Options that control optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options. Accessed: 2021-06-25.

[3] The performance between GCC optimization levels - Phoronix. https://www.phoronix.com/scan.php?page=article&item=gcc_47_optimizations&num=1.

[4] PyVEX. https://github.com/angr/pyvex.

[5] Tracer. https://github.com/angr/tracer. Accessed: 2021-06-25.

[6] G Balakrishnan, T Reps, D Melski, and T Teitelbaum. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems*, 32(6):1–84, Aug 2010.

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[8] Musard Balliu, Mads Dam, and Gurvan Le Guernic. Encover: Symbolic exploration for information flow security. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 30–44. IEEE, 2012.

[9] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 845–860, 2014.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[12] Cristian Cadar. Targeted program transformations for symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 906–909, 2015.

[13] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[14] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108, 2018.

[15] Hui Chen. The influences of compiler optimization on binary files similarity detection. In *the International Conference on Education Technology and Information System (ICETIS 2013)*, 2013.

[16] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. Learning to accelerate symbolic execution via code transformation. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[17] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[18] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 14, Newport Beach, CA, 2011.

[19] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 99–116, 2017.

[20] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[21] Shiyu Dong, Oswaldo Olivo, Lingming Zhang, and Sarfraz Khurshid. Studying the influence of standard compiler optimizations on symbolic execution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 205–215. IEEE, 2015.

[22] Joe W Duran and Simeon Ntafos. A report on random testing. In *ICSE*, volume 81, pages 179–183. Citeseer, 1981.

[23] Stefan Edelkamp. Symbolic exploration in two-player games: Preliminary results. In *The International Conference on AI Planning & Scheduling (AIPS), Workshop on Model Checking*, pages 40–48, 2002.

[24] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[25] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[26] Caroline Lemieux and Koushik Sen. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. *arXiv preprint arXiv:1709.07101*, 2017.

[27] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.

[28] Yan Lin and Debin Gao. When function signature recovery meets compiler optimization. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[29] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[30] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, page 12, Jun 2007.

[31] Anh Nguyen-Tuong, David Melski, Jack W Davidson, Michele Co, William Hawkins, Jason D Hiser, Derek Morris, Ducson Nguyen, and Eric Rizzi. Xandra: An autonomous cyber battle system for the cyber grand challenge. *IEEE Security & Privacy*, 16(2):42–51, 2018.

[32] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[33] David M Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 68–78, 2017.

[34] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don't interpret, compile! In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 181–198, 2020.

[35] Sebastian Poeplau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *Proceedings of the 2021 Network and Distributed System Security Symposium*, 2021.

[36] Sebastian Poeplau and Aurélien Francillon. Systematic comparison of symbolic execution systems: intermediate representation and its generation. In *Proceedings of the 35th Annual Computer Security Applications Conference*, page 163–176. ACM, Dec 2019.

[37] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[38] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. Unleashing the hidden power of compiler optimization on binary code difference: an empirical study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 142–157, 2021.

[39] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy*, page 317–331. IEEE, 2010.

[40] Paria Shirani, Lingyu Wang, and Mourad Debbabi. Binshape: Scalable and robust binary library function identification using function shape. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 301–324. Springer, 2017.

[41] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[42] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, 2016.

[43] Kevin R Wadleigh and Isom L Crawford. *Software optimization for high-performance computing*. Prentice Hall Professional, 2000.

[44] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. -overify: Optimizing programs for fast verification. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, number CONF, 2013.

[45] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.

[46] Shunfan Zhou, Zhemin Yang, Dan Qiao, Peng Liu, Min Yang, Zhe Wang, and Chenggang Wu. Ferry: State-aware symbolic execution for exploring state-dependent program paths. 2022.