Studying the Influence of Standard Compiler Optimizations on Symbolic Execution

Shiyu Dong[†], Oswaldo Olivo[§], Lingming Zhang[*], Sarfraz Khurshid[†]

[†]Department of Electrical and Computer Engineering, University of Texas at Austin, 78712, USA
{shiyud, khurshid}@utexas.edu
[§]Department of Computer Science, University of Texas at Austin, 78712, USA
olivo@cs.utexas.edu
[*]Department of Computer Science, University of Texas at Dallas, 75080, USA
lingming.zhang@utdallas.edu

***Abstract*—Systematic testing plays a vital role in increasing software reliability. A particularly effective and popular approach for systematic testing is symbolic execution, which analyzes a large number of program behaviors using symbolic inputs. Even though symbolic execution is among the most studied analyses during the last decade, scaling it to real-world applications remains a key challenge. This paper studies how a class of semantics-preserving program transformations, namely compiler optimizations, which are designed to enhance performance of standard program execution (using concrete inputs), influence traditional symbolic execution. As an enabling technology, the study uses KLEE, a well-known symbolic execution engine based on the LLVM compiler infrastructure, and focuses on 33 optimization flags of LLVM. Our specific research questions include: (1) how different optimizations influence the performance of symbolic execution for Unix Coreutils, (2) how the influence varies across two different program classes, and (3) how the influence varies across three different back-end constraint solvers. Some of our findings surprised us. For example, applying the 33 optimizations in a pre-defined order provides a slowdown (compared to applying no optimization) for a majority of the Coreutils when using the basic depth-first search with no constraint caching. The key finding of our work is that standard compiler optimizations need to be used with care when performing symbolic execution for creating tests that provide high code coverage. We hope our study motivates future research on harnessing the power of symbolic execution more effectively for enhancing software reliability, e.g., by designing program transformations specifically to scale symbolic execution or by studying broader classes of traditional compiler optimizations in the context of different search heuristics, memoization, and other strategies employed by modern symbolic execution tools.**

I. Introduction

Software testing and verification provide an effective means to improve software reliability. During the last decade, one of the most widely studied techniques for testing and verification is symbolic execution, a classic program analysis based on a systematic exploration of the program's execution paths (up to a bound on the search depth) [16], [37], [12], [25], [35], [51], [34], [44], [54], [59], [7]. While testing and verification were the original application areas of symbolic execution when it was first conceived over 30 years ago [16], [37], recent work has shown its application in areas as diverse as security [7] and software energy consumption estimation [31], as well as directly supporting software reliability [20], [21], [49].

The path-based analysis that symbolic execution performs to compute its results is powerful but time consuming since programs often have a very large number of paths. To reduce the cost of symbolic execution, researchers have designed a number of optimizations to speed it up, e.g., using composition [24], [50], parallelization [55], [52], differential analysis [48], memoization [12], [60], [57], and heuristics [9], [41]. Despite these advances, scaling symbolic execution to effectively handle real-world applications remains a key technical challenge [14].

In this paper we study a class of semantics-preserving program transformations that were originally designed to make *conventional* program execution (using concrete inputs) faster [6] and we also hope to provide a promising basis for designing new optimizations for symbolic execution. Specifically, we study *compiler optimizations* to empirically evaluate their effect on the performance of symbolic execution. The evaluation results show possible negative performance impact of the optimizations on symbolic execution – contrary to what one might assume given their well-known benefits for conventional execution. Our paper thus presents an important advance in how users of symbolic execution for reliability – whether through improved testing and verification or through direct application – may benefit most from this powerful analysis.

Traditional compiler optimizations form a well-studied area in the context of conventional execution. Modern compilers, such as GCC [1] and LLVM [39], support a number of *basic* as well as *aggressive* optimizations, and allow the users to manually select a suitable optimization *level* for their applications. Moreover, some recent research projects have addressed the problem of automatically identifying *combinations* of optimizations for given applications to achieve likely optimal benefits across a number of different axes, e.g., time, memory, and program size, using heuristics [46] [56].

However, the use of compiler optimizations in the context of symbolic execution is not so well-understood. A symbolic execution engine that explicitly supports their use is KLEE [12], which analyzes C programs. KLEE's foundation on the LLVM infrastructure [39] allows KLEE to directly access and apply (if the user so chooses) dozens of optimizations that the LLVM compiler provides.

Given the semantics preserving and performance optimizing nature of the program transformations that compiler optimizations perform by construction, it can be natural to simply reason that compiler optimizations offer obvious benefits for symbolic execution similar to their benefits for traditional execution [3]. However, such reasoning is complicated by the fact that symbolic execution relies on an SMT solver [23] that is used to check the satisfiability of each *path condition*, which represents conditions on inputs required to execute that path, such as *branch* conditions and *in-bounds load/store* conditions. The issue that complicates applying a compiler optimization to symbolic execution is that while it is easier to predict that eliminating instructions, removing redundant computations, or enabling other optimizations typically reduces program runtime in *standard* execution, a similar reduction is not so obvious in symbolic execution. The reason is that the main bottleneck for symbolic execution is the SMT solving time and the SMT solver is usually used as a *black-box*. In fact, some optimizations are simply irrelevant to the path conditions (and thus have no impact on SMT solving time), and others like transforming loop variables to promote further loop optimizations might even generate path conditions that are harder for the solvers to handle. To our knowledge, previous research has not rigorously studied the relation between standard compiler optimizations and symbolic execution.

Our goal in this paper is to study compiler optimizations in the context of *traditional* symbolic execution [37], [16], which can be implemented by using a deterministic (bounded) depth-first search [35], [12], [25], [51]. KLEE , with its industrial strength implementation and basis on the advanced compilation infrastructure of LLVM, provides the enabling technology for our study. KLEE has several options, e.g., search strategy and constraint caching, which users can select to control how symbolic execution is performed. To make KLEE's analysis faithful to traditional symbolic execution (which is the focus of our study), we use KLEE with two specific settings: depth-first search and caching disabled.

We present our study of 33 compiler optimizations implemented by the LLVM compiler and used by KLEE. Specifically, we study three core research questions. One, we study how different optimizations influence the performance of symbolic execution for Unix Coreutils. Coreutils are a set of programs that KLEE handles very well. The initial embodiment of KLEE found a number of previously unknown bugs in several programs in Coreutils [12]. Subsequently, KLEE was applied to test a number of more recent versions of Coreutils [42]. Two, we study how the influence of compiler optimizations varies across two different classes of programs: (1) Coreutils and (2) NECLA benchmarks [5]. Three, we study how the influence of compiler optimizations varies across three different back-end constraint solvers for symbolic execution: *STP* [23], *Z3* [19], and *Boolector(Btor)* [8]. The *STP* solver is the primary solver for KLEE . Recent work on KLEE [45] added support for *Z3* and *Btor*.

Our initial findings are:

- Certain compiler optimizations influence symbolic execution more than the other optimizations. On average, applying optimizations makes symbolic execution worse for Coreutils programs. Moreover, using a combination of optimizations makes symbolic execution even worse, and KLEE's default settings are in general not optimal for symbolic execution of Coreutils programs.
- The influence of compiler optimizations varies across different sets of subject programs. Specifically, optimizations help symbolic execution of the smaller NECLA benchmarks. Moreover, the optimizations that have the maximum influence on NECLA benchmarks are a subset of those that have the highest influence on Coreutils. Furthermore, applying more optimizations does not necessarily make symbolic execution faster, regardless of the order, even for NECLA benchmarks.
- The influence of compiler optimizations is similar across the three solvers. Moreover, the relative performance of the solvers is similar across different programs even in the presence of compiler optimizations; specifically *STP* performs better than *Z3* and *Btor*.

```
1int main() {
2  int a;
3  klee_make_symbolic(&a,
       sizeof(a), "a");
4  klee_assume(a > 0);
5  klee_assume(a < 51);
6  int x = 0;
7  int y = 0;
8  int i;
9  for (i=0;i<a+1;i++)
10   x = x + 3;
11 for (i=0;i<a+1;i++)
12   y = y + 4;
13 return x + y;
14}
```

```
1int main() {
2  int a;
3  klee_make_symbolic(&a,
       sizeof(a), "a");
4  klee_assume(a > 0);
5  klee_assume(a < 51);
6  int x = 0;
7  int y = 0;
8  int i;
9  for (i=0;i<a+1;i++) {
10   x = x + 3;
11   y = y + 4;
12 }
13 return x + y;
14}
```

Fig. 1. A compiler optimization example that reduces SMT queries for symbolic execution.

We hope our work will motivate further research in designing effective compiler optimizations specifically targeted to symbolic execution [58] as well as studies of compiler optimizations in wider settings for symbolic execution, e.g., using heuristics [47], different search strategies [12], [40], and memoization [12], [60], [57]. We believe symbolic execution has a vital role to play in software reliability and optimizing symbolic execution further will improve our ability to study and enhance the reliability of real-world software systems.

II. Motivating Examples

This section presents two small examples to demonstrate that compiler optimizations can both reduce and increase the number of SMT queries.

Our first example uses *loop fusion*, which consists of combining loops that have statements in common to avoid redundant computations across the loops. Figure 1 shows the example code before and after the loop fusion optimization. We mark the change in grey.

In this example loop fusion helps symbolic execution because the branching conditions that are present at the lower

```
int main() {
  int N;
  int i;
  klee_make_symbolic(&N,
      sizeof(N),"N");
  klee_assume(N>0);
  klee_assume(N<10);
  int a[10];
  for(i=0;i<N;++i) {
    a[i]=i;
  }
  for(i=0;i<N-3;++i) {
    a[i]=0;
  }
  int sum=0;
  for(i=0;i<N;++i)
    sum+=a[i];
  return sum;
}
```

```
int main() {
  int N;
  int i;
  klee_make_symbolic(&N,
      sizeof(N),"N");
  klee_assume(N>0);
  klee_assume(N<10);
  int a[10];
  for( i=N-3 ;i<N;++i)
      {
    a[i]=i;
  }
  for(i=0;i<N-3;++i) {
    a[i]=0;
  }
  int sum=0;
  for(i=0;i<N;++i)
    sum+=a[i];
  return sum;
}
```

Fig. 2. A compiler optimization example that increases SMT queries for symbolic execution.

level representation of the code are reduced as the second loop is removed. The number of queries sent to the solver is 208 before the optimization and 106 after the optimization, which is expected given that two similar loops are converted into one.

Next, we present an example that uses aggressive dead code elimination, which assumes all instructions are dead unless proven otherwise and tries to eliminate dead statements within loop computations. The code before and after the optimization is shown in Figure 2. We mark the change in grey.

In this case, symbolic execution before the optimization requires 63 queries whereas it requires 154 queries afterwards. This can be explained by the fact that the starting condition of the first loop gets more complicated after the optimization to avoid doing the redundant computations. Although this is favorable in terms of execution time, the resulting path conditions in the context of symbolic execution are harder to analyze.

III. Background

This section gives background on symbolic execution, the KLEE tool, and compiler optimizations.

A. Symbolic Execution

Symbolic execution [37], [16], [51], [12], [48], [55], [47], [61] treats user inputs as *symbolic* values instead of concrete values, so that the execution to be performed covers many potential concrete executions at the same time. To do so, the conditions to reach the different control points in the program are maintained together with sanity checks (such as array indices within bounds). Conceptually, branching instructions create a fork in the symbolic exploration, which considers the execution when the branch is false as well as when it is true. Ideally, if symbolic execution is performed for every feasible path of the program, the software should be very reliable. However, symbolic execution requires solving the path conditions to check their feasibility and avoid exploration of known infeasible paths. Thus, the complexity of the path conditions is in practice a key bottleneck for the scalability of symbolic execution.

B. KLEE and LLVM

KLEE is a symbolic execution engine built on top of the compilation framework LLVM [38]. LLVM grew as an academic project to focus efforts on implementing a strong compiler back-end that was independent of the front end. One of LLVM's best features is its well-defined intermediate representation(IR), over which the back-end operates. The main idea is that multiple front-ends can be plugged into the compiler as long as they produce the correct IR, and similarly, the optimizations and target-generation plugins can be developed independently as long as they can handle the input IR. Ultimately, it has become a widely used compiler, being competitive in performance to GCC. The IR is based on single-static assignment form (SSA) [18], which means that, when possible, variables are defined only once among any execution path. This allows for easier tracking of the values of variables, and hence code that is easier to analyze at compile-time, facilitating the application of optimizations.

KLEE supports test input generation with respect to given input *bounds*. It has been shown to provide excellent program coverage in general, over 90% in average of the Coreutils benchmarks, and has been able to find bugs in these programs that remained undetected for many years. KLEE's error reporting provides useful information for fault localization during debugging. The existing version of KLEE allows either the application of the entire set of compiler optimizations in LLVM or no application of optimizations.

C. Compiler Optimizations

Compiler optimizations transform the source program into a more efficient (i.e. faster, smaller, less power-consuming) target program to be executed. Since KLEE is built on top of LLVM, we study the effect of the LLVM optimizations in symbolic execution. These optimizations are mostly loop transformations, conversion of memory operations to register operations, simplifications of computed expressions and elimination of redundant instructions. These are all transformations that are well defined over lattices using the data flow analysis framework [36]. They have a natural implementation as a fix-point computation algorithm. These optimizations or similar ones have been described in traditional textbooks and dissertations [6][17][29].

IV. Experimental Study

In this section we will first define our research questions. Given these questions, we will then describe our design of a series of experiments, including independent and dependent variables. Then we will present and analyze the results and give our answers as well as explanations to the defined research questions in Section V.

A. Research Questions

We study the following research questions in this paper:

RQ1: How do LLVM compiler optimizations influence symbolic execution performance for Coreutil programs?
Given the above motivating examples, we would like to study more about how LLVM compiler optimizations influence symbolic execution with respect to the Coreutil programs. More specifically, first we want to know if compiler optimizations have positive or negative effects on symbolic execution for Coreutils. Also, we would like to find out whether there are any specific optimization flags that have great impact on symbolic execution. Moreover, we want to observe whether adding more optimization flags, or having a combination of optimization flags, leads to better or worse results, and whether KLEE's default compiler optimization settings are optimal for Coreutils or not.

RQ2: Is the influence of compiler optimizations on symbolic execution consistent across different program classes?
Different programs may behave differently even for the same setup. Therefore, it might be not very convincing if we only conduct experiments on Coreutil programs. We want to see if we could address something in common from benchmarks with different characteristics and different sizes, and therefore we choose another suite of benchmarks to compare the effect of compiler optimizations on symbolic execution and observe the variation.

RQ3: How do different solvers influence the performance of different optimizations?
Even if we apply the same optimizations on the original program before symbolic execution, different constraint solvers may also have influences on the result of symbolic execution and give different results. We would like to compare the performance of symbolic execution with different solvers after applying different optimizations to see if the behaviors of different optimizations are consistent among different solvers.

B. Experimental Setup

We modified KLEE so that it can take an extra flag specifying which optimization flag(s) to apply before symbolic execution. We use our modified version for all our experiments. For other options of KLEE we use the ones similar to KLEE documentation [4]. The main change that we have made for the setup is that we use the DFS search heuristic instead of KLEE's default random search heuristic in order to have more deterministic results. Also, we disable the caching of KLEE for all experiments in order to exclude the influence of caching on symbolic executions.

1) Independent Variables: We considered the following independent variables for our study:

Different flags. LLVM has a rich set of optimization flags, and the number is still increasing as LLVM is evolving. Among all of them we choose all 33 flags that KLEE uses inside its *–optimize* option. This option is the optimization option that comes with KLEE . It applies 33 different optimization flags provided by LLVM in a certain order. The set of these 33 flags contains the most commonly used flags in compiler optimization. We study the effect of each of them, and some of their combinations. In later part of this paper we will refer the *–optimize* option that comes with KLEE as the *ALL* optimization.

Program	ELOC	GLOC	Program	ELOC	GLOC
base64	3989	105	nice	4010	59
basename	4026	39	nl	10037	211
chcon	4343	195	od	4463	711
cksum	3983	62	paste	3837	187
comm	3997	98	pathchk	3857	132
cut	4195	296	printf	4251	257
dd	4734	561	readlink	4154	50
dircolors	4093	190	rmdir	3892	72
dirname	3889	31	setuidgid	3878	77
du	5790	302	sleep	4199	46
env	3937	45	split	4428	217
expand	3916	151	sum	4068	95
expr	9565	338	sync	3919	20
fold	3891	113	tee	3966	69
groups	4002	37	touch	4744	145
link	3829	28	tr	4150	659
logname	3902	25	tsort	3856	203
mkdir	4213	66	unexpand	3903	194
mkfifo	3959	47	unlink	3865	25
mknod	3840	80	wc	4075	262

TABLE I. LIST OF ALL COREUTIL PROGRAMS USED IN THIS STUDY.

Different programs. We study two different sets of programs, *Unix Coreutils 6.11*, and the *NECLA Static Analysis Benchmarks (necla-static-small)* [5]. Many researches on symbolic execution use KLEE as the symbolic execution tool to run their experiments against Coreutils. This experiment was originally proposed by the authors of KLEE [12]. We conduct studies with similar setup as the one used by most researchers and select 40 different Coreutils programs for experimental subjects, with the changes of search heuristic option mentioned above. Table I lists of all of the Coreutil programs studied in this work with their ELOC and GLOC. ELOC shows the size of the programs in terms of the number of executable lines of code [40], while GLOC shows the lines of code excluding library and head code, e.g., the lines of code directly traced by gcov, which is a tool used in conjunction with GCC to test code coverage in programs [2]. The NECLA benchmarks [5] are a traditional set of C benchmarks to test the performance of compilers. Normally they are small, loop intensive and perform operations with integer variables and arrays. We modified some of them by changing some variables inside them to symbolic variables, and adding some nondeterministic bounds to these variables inside the program to make them compatible with KLEE .

Different solvers. The latest version of KLEE supports three different types of solvers: *STP*, *Z3* and *Btor*[45]. The *STP* solver is the native solver integrated with KLEE, and the other two are recently supported. We would like to study and compare the effect of all of them together with different compiler optimizations.

2) Dependent Variables: For different experiments we would like to measure different dependent variables according to the property of each experiment design. Specifically there are two types of experiments, and we list the dependent variables for each of them.

Limited time. For all Coreutil programs, since they are normally large and complicated, we will limit the execution time and halt the execution when the specified time reaches. For different setups we will choose 5, 10, 20 or 30 minutes execution time. We measure both line and branch coverage after the execution stops. By comparing different coverage numbers using the same execution time but different optimizations, we could see the performance difference between different optimization flags.

Unlimited time. For the NECLA benchmarks, the program size is usually very small and they do not require complicated inputs such as the Coreutils. Therefore we let the programs finish their execution. In this case, we measure the time needed for execution and the number of solver calls. When the programs finish executing, they will typically give the same coverage. Therefore we can compare the execution time and also the number of solver calls. With the same coverage, shorter time and less number of solver calls indicates faster symbolic execution.

V. Result Analysis

A. RQ1: Influence of optimizations on symbolic execution

The main question we would like to answer is how compiler optimizations influence symbolic execution for Coreutil programs. Specifically, we would like to know whether compiler optimizations are generally beneficial or detrimental for symbolic execution. Also, we believe that among all optimization flags, there are some flags that are more influential than others, and we would like to verify if our assumption is correct. Moreover, we want to know whether a combination of optimization flags makes symbolic execution better or worse, and whether KLEE's optimization setting is optimal for symbolic execution of Coreutil programs.

With these questions, we design several experiments. We will present them in the following subsections.

1) Finding the determining flags: We combine all the 40 studied Coreutils programs with each of the 33 optimization flags mentioned in Section IV-B1 , plus no optimization (*NO*) and KLEE's optimization option (*ALL*) that comes with KLEE . We limit each run to 5 minutes and record the line coverage for each run. We consider it as a change if the line coverage after applying certain optimization is different from not applying any optimization. All flags that we use and the number of changes that each flag makes are shown in Table II.

From Table II we can see that among all 33 optimization flags and the *ALL* flag, many of them only make changes for less than 10 programs. From the actual raw data we can observe that there are some programs whose result will be changed after applying almost every single flag, and these programs contribute a lot to those changes with small numbers. However, there are certain flags that make significant changes to most programs. The *ALL* flag makes most of the changes to the program, because it applies all other optimization flags

Optimization	# Prog.	Optimization	# Prog.
ALL	34	LoopUnroll	5
InstructionCombining	29	ArgumentPromotion	4
IndVarSimplify	20	DeadStoreElimination	4
PromoteMemoryToRegister	19	DeadTypeElimination	4
ScalarReplAggregates	19	FunctionAttrs	4
LoopRotate	11	IPConstantPropagation	4
AggressiveDCE	8	LoopDeletion	4
GVN	8	MemCpyOpt	4
SCCP	8	PruneEH	4
LoopUnswitch	7	RaiseAllocation	4
StripDeadPrototypes	7	TailCallElimination	4
CondPropagation	6	CFGSimplification	3
FunctionInlining	6	DeadArgElimination	3
JumpThreading	6	GlobalDCE	3
ConstantMerge	5	GlobalOptimizer	3
LICM	5	Reassociate	3
LoopIndexSplit	5	SimplifyLibCalls	3

TABLE II. Number of programs with changed coverage when applying individual flags.

in a certain order. We also observe that *InstructionCombining(IR)* makes the second most changes to the programs. Also, *IndVarSimplify(IVS)*, *PromoteMemoryToRegister(PMTR)* and *LoopRotate(LR)* makes more than or about half of the programs to change.

We could further separate the above changes into two categories: the ones that make symbolic execution better and the ones that make symbolic execution worse. Figure 3 illustrates this separation. The x-axis represents different optimization flags. The bar above the x-axis represents the number of programs for which the optimizations have positive effects, and below means negative changes. From this figure we notice the very interesting observation that the majority of flags tend to make symbolic execution worse, since for most programs the light gray area below x-axis is larger than the dark area above x-axis. This observation indicates that, although compiler optimizations are usually beneficial for program execution, the same may not be true for symbolic execution in general.

We consider the above top five flags as the "determining flags" for our setup, since they contribute the most in making symbolic execution different from applying no optimization. We will further study the effect of these flags in later experiments.

2) Analyzing the determining flags: Using the five determining flags from Section V-A1, we study more on the effect of them on symbolic execution. We run KLEE using *NO*, *ALL* and these five determining flags one by one, on all 40 Coreutils listed in Table I, and limit the time to 5, 10, 20 and 30 minutes accordingly using the DFS heuristic. Again, we first apply *NO* for different time limits and get the line and branch coverage for each run as the "base". Then we apply either *ALL* optimization or a single optimization flag to get the line coverage and branch coverage after optimization. We show the box plots of the actual line and branch coverage for this set of experiments in Figure 4 and 5. In each figure, the four sub-figures are box plot of the coverage running KLEE for 5, 10, 20 and 30 minutes, and each box is the result of applying one of the optimization flags mentioned above for the corresponding time. Each box plot shows the average (a small square in the box), median (a line in the box), and upper/lower quartile values for the corresponding code coverage. We mark the results for the pre-defined KLEE flags (*NO* and *ALL*) as gray and the other five individual flags (*IVS*, *IC*, *LR*, *PMTR*

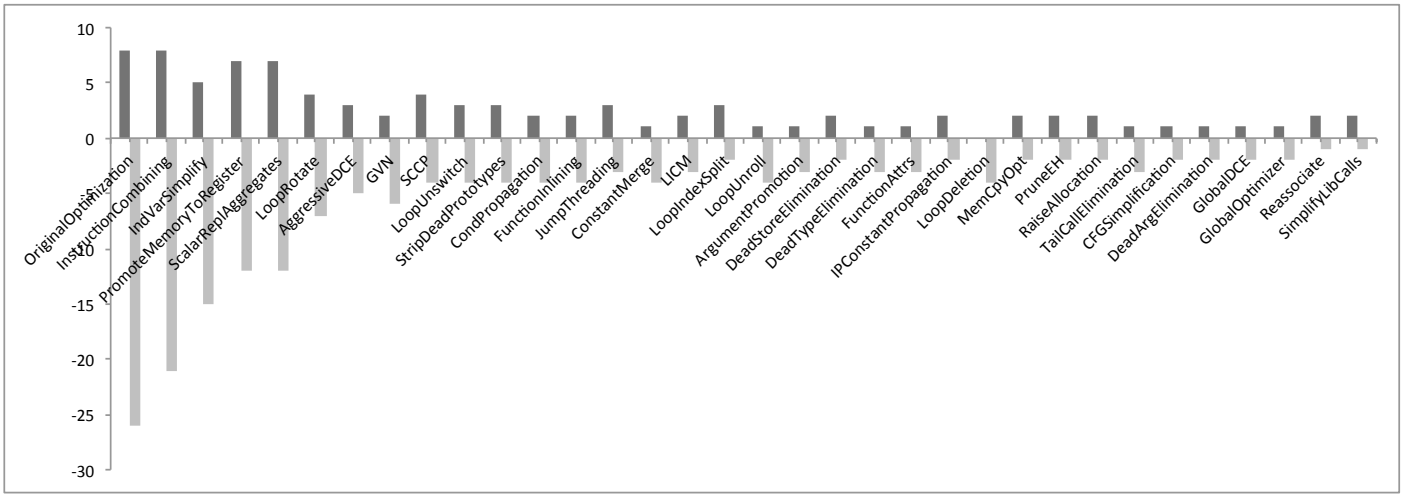


Fig. 3. The influence of different compiler optimization flags to programs under test.

and *SRA*) as red. From the corresponding box plots, we make the following observations:

First, although the performance for an individual optimization on symbolic execution varies from program to program according to the box plot, the average result shows the trend that compiler optimizations are in general making symbolic execution worse for Coreutils programs. To illustrate, *NO* (applying no optimizations) achieves better average coverage for all time limitations than all the other single compiler optimization flags and the *ALL* optimization. This result is quite surprising to us, since KLEE's documentation[3] says the following:

"We can help with that problem (and others) (note from author: low coverage) by passing the –optimize option to KLEE. This will cause KLEE to run the LLVM optimization passes on the bitcode module before executing it; in particular they will remove any dead code. When working with non-trivial applications, it is almost always a good idea to use this flag."

It seems that KLEE simply applies all compiler optimizations, but our experimental results show the negative effects of doing so.

Second, from the results we can observe that among all the optimization flags that we choose, the *ALL* flag performs worse than all single flags for both line and branch coverage, which is even more surprising. Take line coverage and 30 minutes as an example, the *ALL* flag (applying all optimizations) only achieves an average coverage of 52.6%, while the coverage for all single flags are greater than that. The worst single flag, *IC*, achieves an average coverage of 55.4%, and all the others achieve coverage greater than 60%. We could also easily make similar observations in the box plot since the boxes for *ALL* flag are always lower than the others. As mentioned above, KLEE is using a traditional order of optimization which in general helps with program execution. However, according to the experimental result the *ALL* optimization that KLEE uses is the worst optimization for symbolic execution in our settings. Therefore, we can conclude that KLEE's default setting for optimization is not optimal for symbolic execution, at least for our experimental setup. This observation leads us to wonder whether the combination of more optimization flags will make symbolic execution even worse. In the next section we will use a new set of experiments to explore this question.

Name	Flags
TRAD	*PMTR*, *SRA*, *IVS*, *LR*, *IC*, *PMTR*
EXTR	*PMTR*, *IC*, *SRA*, *IC*, *LR*, *IC*, *IVS*, *IC*, *SRA*, *IC*, *PMTR*
C3	*PMTR*, *IC*, *SRA*, *IC*
C4	*LR*, *IVS*, *PMTR*, *IC*, *SRA*, *IC*

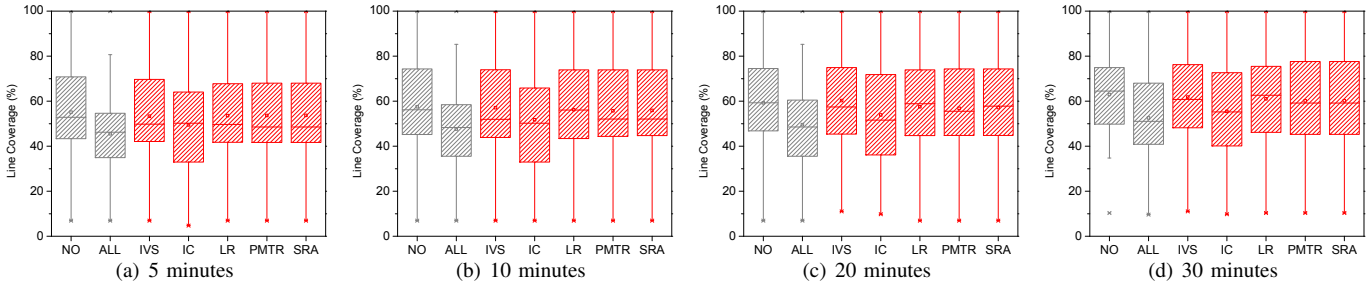
TABLE III. LIST OF DIFFERENT OPTIMIZATION COMBINATIONS.

3) Study of different optimization combinations: In order to study the effect of compiler optimization combinations on symbolic execution, we further come up with four different combinations based on our knowledge. We combine the determining flags that we found out from previous experiments in different orders. One flag may appear more than once.

The first combination(*TRAD*) is an arrangement according to the traditional compiler optimization order. The idea is from the famous compilers "Dragon Book" [6]. It suggests a normal order of applying optimizations that is good for program execution in general. The second combination (*EXTR*) is extracted from KLEE's original optimization only using the determining flags, with the same order and number of occurrences. The third and fourth combinations (*C3* and *C4*) are based on our understanding of compilers. Table III lists all of them.

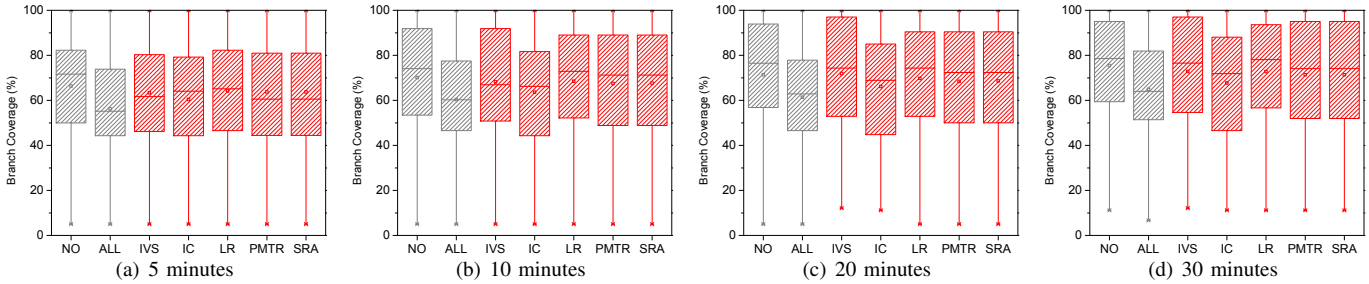
We first run KLEE using *NO* and *ALL*. After that we apply these combinations one by one. Each run in this section takes 10 minutes. Similar to the previous section, for the average line and branch coverage, we show box plots in Figure 6. We mark *NO* and *ALL* in gray, our four combinations *TRAD*, *EXTR*, *C3* and *C4* in red. To make further comparison, we also show the worst single flag *IC* and mark it in blue.

From Figure 6 we can observe that none of the optimizations are making symbolic execution any better. Also, comparing the box plot of *ALL* with the other combinations of



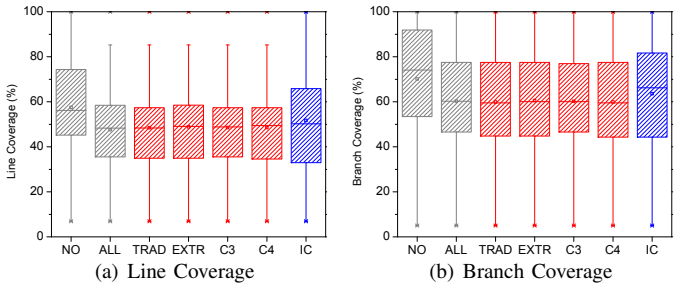
(a) 5 minutes (b) 10 minutes (c) 20 minutes (d) 30 minutes

Fig. 4. Line coverage of Coreutils programs using KLEE with individual optimizations.



(a) 5 minutes (b) 10 minutes (c) 20 minutes (d) 30 minutes

Fig. 5. Branch coverage of Coreutils programs using KLEE with individual optimizations.



(a) Line Coverage (b) Branch Coverage

Fig. 6. Line and branch coverage of Coreutils programs using KLEE with optimization combinations.

optimizations, we find that they perform almost the same. This observation shows the effectiveness of the chosen determining flags, that the combinations of the five determining flags have similar effect to *ALL*. Also, if we compare all the combinations including *ALL* with the worst single flag *IC*, we could observe that all of them are worse than the worst single flag, and even worse than *NO*. Therefore we can conclude that applying more optimization flags within a given time limit tends to make symbolic even worse than applying no optimization or a single optimization. To sum up, in this experimental setup we have the following findings:

First, on average, given limited time and using the DFS search heuristic, compiler optimizations are making symbolic execution for Coreutil programs worse in terms of line coverage and branch coverage.

Second, there are several determining flags which have greater influence on symbolic execution. According to our experimental setup the top five flags are *IC*, *IVS*, *PMTR*, *SRA* and *LR*. They have the most influence on symbolic execution among all optimization flags that we choose, and the combination of them provides similar result to the *ALL* flag.

Third, applying a combination of optimizations is not a good choice. According to our experiments, applying more optimizations tends to make symbolic execution even worse, and the most complicated *ALL* flag gives the worst result. Therefore, KLEE's default compiler optimization settings is not optimal for Coreutil programs.

B. RQ2: Result consistency across different program classes

In section V-A we studied the effect of compiler optimizations on symbolic execution based on the Coreutil programs, which is a set of real-world programs with usually complicated structures. In this section we will conduct similar experiments on another class of programs. We choose a small set of NECLA Static Analysis Benchmarks [5]. These benchmarks are C based programs often used to test the performance of compilers. All the programs are quite small in size and are less than 100 lines of code. Many of them are loop intensive and perform integer and array operations for most of the time. We used all the six NECLA programs that can be successfully executed with KLEE . For all the following experiments in this section we use similar options as those for Coreutil programs.

1) Single optimization flag: We first apply *NO*, *ALL*, and all 33 individual optimization flags with a similar setup as the previous experiments, and record the result for each of them. However, among all 33 optimization flags that KLEE has used, we notice that only three flags, *IC*, *PMTR* and *SRA*, make a change between the optimized program and original program in either execution time or number of solver calls. In order to save space and show only meaningful results, we only list the result of applying *NO*, applying *ALL*, and applying these three individual flags in Table IV, where "Time" shows the

	Flag	ex2	ex3	ex8	ex9	ex30	ex34
Time	NO	1.44	0.14	0.04	0.94	0.33	0.15
	ALL	0.2	0.02	0.05	0.14	0.07	0.09
	IC	0.06	0.12	0.05	0.07	0.06	0.09
	PMTR	1.34	0.13	0.03	0.96	0.32	0.15
	SRA	1.36	0.13	0.03	0.95	0.32	0.15
Queries	NO	528	42	16	273	122	19
	ALL	52	6	17	36	30	19
	IC	19	33	17	18	25	19
	PMTR	528	42	16	273	122	19
	SRA	528	42	16	273	122	19

TABLE IV. APPLYING INDIVIDUAL OPTIMIZATIONS ON NECLA BENCHMARKS.

	Flag	ex2	ex3	ex8	ex9	ex30	ex34
Time	I+P	0.07	0.13	0.06	0.11	0.08	0.12
	I+S	0.06	0.11	0.07	0.07	0.07	0.1
	P+I	0.06	0.13	0.05	0.06	0.07	0.09
	P+S	1.33	0.13	0.04	0.97	0.31	0.15
	S+I	0.07	0.19	0.05	0.07	0.06	0.11
	S+P	2.13	0.16	0.04	1.05	0.39	0.17
Queries	I+P	19	33	17	18	25	19
	I+S	19	33	17	18	25	19
	P+I	19	33	17	18	25	19
	P+S	528	42	16	273	122	19
	S+I	19	33	17	18	25	19
	S+P	528	42	16	273	122	19

TABLE V. APPLYING OPTIMIZATION COMBINATIONS ON NECLA BENCHMARKS.

symbolic execution time, and "Queries" shows the number of solver calls.

From Table IV we can observe that, unlike in the previous Coreutil experiments, the *ALL* optimization, in many of the programs, helps with symbolic execution by significantly reducing the execution time and number of solver calls while giving the same coverage. This result conflicts with our previous results that compiler optimization makes symbolic execution worse. However, we believe that both results are understandable because these two experiment are of two different program classes. All the NECLA programs are very small and are intentionally designed to test the performance of compilers, so there is a lot of space left for optimizations. However, for the real-life Coreutils, the program size is usually very large and the they do more realistic tasks other than array operations and they use more complicated data types and data structures. Therefore, the space left for optimization is very narrow and applying some optimizations might even make the program structure more complicated.

Another observation we can make is that the behavior of *IC* is very similar to *ALL*, which indicates that for this experimental setup *IC* might be the most significant determining flag. The other two optimizations slightly changed the symbolic execution time, while all the remaining flags that are not listed in Table IV do not change anything. Also, another very interesting observation is that, all the three flags that we list for this experiment are a subset of the determining flags that we get from the previous Coreutil experiment. This observation indicates that there might be several determining optimization flags in common across different program classes, which have more influence on symbolic execution than others.

2) Multiple optimization flags: Similar to the previous experiment, we also want to explore the effect of different flag combinations for the small benchmarks. Since this time we only have 3 determining flags, we simply combine every two of them in different order, and list the result in Table V. Note that in this table, for simplicity we use *I* for *IC*, *P* for *PMTR* and *S* for *SRA*.

Table V shows the results after combining two out of the three determining flags. There are two observations that we can make for this case. First, applying one more flag on the most significant determining flag (which is *IC* in our case) does not help anymore in symbolic execution. For example, in ex2 applying *I+P* gives almost the same result as applying *IC* only. Second, the order of flags does not change the execution results. Take ex2 as an example again, applying *IC* before *PMTR* gives the same result as applying *PMTR* before *IC*. We believe that all above behaviors are due to the fact that *IC* is very effective for our selected benchmarks, since they are share similar characteristics and it is possible that one optimization in our case will outperform all other flags for our selected benchmarks.

Since all these small programs are used to test the functionality of compilers and finding compiler bugs and their sizes are usually small, they do not resemble the real-life programs, and the conclusion we get for this experiment might not be applicable to larger-sized real-world programs. That is the main reason why some of the results in this experiment are not in accordance with the results from the Coreutils experiment. Nonetheless, even for the Coreutil experiments there are some cases where compiler optimizations help with symbolic execution. Therefore, we cannot yet claim that compiler optimizations are beneficial or detrimental for symbolic execution for a specific program, since the program itself is still an important factor. However, we think in future work if we could characterize different programs into different categories, and study the effect of different optimizations on each category, it is very possible that we could get some more concrete observation and generalize the effect of certain optimizations to the result of symbolic execution of certain kinds of programs.

C. RQ3: Influence of solvers on optimizations

All previous results are based on the *STP* solver which comes with KLEE . We also want to see the effect of different solvers working together with different optimization flags. Recently Palikareva et al. proposed a new infrastructure for KLEE which supports different constraint solvers [45]. This infrastructure provides us a good opportunity to study the effect of multiple solvers, together with compiler optimizations, on symbolic execution.

We use the same 11 Coreutils and same setup as the authors used in their paper for multi-solver support for KLEE [45]. Again, we use our modified version of KLEE so that it can take one or more optimization flags. Using the same determining flags that we mentioned above (*NO*, *ALL* and the five determining optimization flags), we execute KLEE for 10 minutes

	NO	ALL	IVS	IC	LR	PMTR	SRA
STP	62.06	50.52	58.51	50.55	63.69	63.50	63.50
Z3	61.02	49.71	51.30	49.51	60.62	61.13	61.47
Btor	56.49	45.28	54.40	45.76	56.82	58.66	58.66

TABLE VI. AVERAGE LINE COVERAGE OF EACH OPTIMIZATION SOLVER PAIR FOR ALL PROGRAMS.

	NO	ALL	IVS	IC	LR	PMTR	SRA
STP	74.05	60.85	69.87	60.94	74.59	75.24	75.24
Z3	73.97	59.65	62.12	59.83	73.37	73.20	74.01
Btor	65.32	50.91	62.44	52.42	65.59	67.89	67.89

TABLE VII. AVERAGE BRANCH COVERAGE OF EACH OPTIMIZATION SOLVER PAIR FOR ALL PROGRAMS.

for each run and record the line and branch coverage for each program-optimization-solver tuple. We disable caching in this experiment in order to get the result close to the traditional symbolic execution. One thing to point out is that these 11 programs are not the same set as the above Coreutil examples, therefore we may see different average coverage values against previous Coreutil experiments. We calculate the average line and branch coverage of all 11 programs, for each optimization-solver pair, and list them in Table VI and VII.

From Tables VI and VII we can see that the behavior of different optimization flags are slightly different working with different solvers, because the coverage for the same optimization can be different for different solvers. However, their relative behaviors are still the same. Similarly to our previous Coreutil experiments, if we horizontally compare the result for each optimization flag for different solvers, we can see that the *ALL* optimization still performs the worst compared with any single flags for branch coverage for all three solvers. Also, we cannot observe a specific optimization-solver pair that gives abnormally low or high coverage. Therefore, we can conclude that the performance of different optimizations is consistent among different solvers. Another finding is that the STP solver consistently outperforms the other two solvers using each compiler optimization setting. This indicates that the STP solver can be optimal for the KLEE symbolic execution technique no matter which compiler optimization is applied.

D. Threats to Validity

Threats to external validity. The main threat to external validity of our study is that our findings may not be generalizable for other subject programs, symbolic execution tools, or compiler optimizations. To reduce this threat, we studied two different set of subjects, each of which has been widely used in software testing and analysis research. In addition, to make our results replicable, we used the widely used KLEE tool with the deterministic DFS search heuristic, and the LLVM framework with 33 popular compiler optimizations. However, our results may still suffer from the threats to external validity. Further reduction of these threats to external validity requires additional studies using different symbolic execution tools with different search strategies, more compiler optimizations, as well as more experimental settings, e.g., using longer time for each symbolic execution run.

Threats to internal validity. The main threat to internal validity of our study is that there may be potential faults in the studied symbolic execution and compiler optimization techniques, as well as in our code for data analysis. To reduce this threat, we used the mature symbolic execution tool KLEE and the compiler optimization flags in the widely used LLVM framework. In addition, we reviewed all the code that we produced for our experiments before conducting the experiments.

Threats to construct validity. The main threat to construct validity is the metrics that we used to assess the efficiency of symbolic execution under different compiler optimizations. To reduce this threat, for our large subjects, we limit the experimentation time and then check the statement and branch coverage that symbolic execution with certain compiler optimizations can achieve. For our small subjects, we record the number of solver calls and time taken by each configuration of symbolic execution to achieve full branch and statement coverage.

VI. RELATED WORK

This section first reviews some related work on scaling symbolic execution for testing and verification using parallel, compositional, and incremental techniques, and then discusses compiler optimizations in the context of standard execution and symbolic execution.

A couple of recent research projects proposed *parallel* symbolic execution [55], [52], [53]. Static partitioning [55] uses an initial shallow run of symbolic execution to minimize the communication overhead during parallel symbolic execution; the key idea is to create pre-conditions using conjunctions of clauses on path conditions encountered during the shallow run and to restrict symbolic execution by each worker to program paths that satisfy the pre-condition for that worker's path exploration. ParSym [52] parallelizes symbolic execution dynamically by treating every path exploration as a unit of work and using a central server to distribute work between parallel workers. While this technique implements a direct approach for parallelization [27], [13], it requires communicating symbolic constraints for every branch explored among workers, which incurs a higher overhead. Ranged symbolic execution [53] represents the state of a symbolic execution run using a just concrete test input to provide a lightweight solution for work stealing.

Compositional techniques for symbolic execution, introduced by PREfix and PREfast [10], can handle large code bases but they do not handle properties of complex data, such as heap-allocated data structures. Godefroid et al.'s work on must summaries [28] also enables compositional symbolic execution [24] as well as compositional dynamic test generation [26] by statically validating symbolic test summaries against changes. These techniques use logical representations for the summaries for sequential code.

Incremental techniques for symbolic execution utilize previous runs of symbolic execution to optimize the current run. DiSE [48] uses a static analysis to determine the differences between two program versions and uses this information to guide the execution of symbolic paths towards exercising that difference. Memoise [60] builds a tree-based representation to store path conditions and path feasibility results during a run

of symbolic execution on one program version and enables reuse of those results during a future run of symbolic execution on a newer program version. The idea of caching constraints in symbolic execution was first introduced by KLEE [12]; doing so allows KLEE to achieve orders of magnitude speed-up because there are often many redundant constraints during symbolic path exploration. The recently developed Green solver [57] wraps every call to the solver to check if the result is already available in its database of previous solver calls.

While traditional work on compiler optimizations defined optimizations and studied their effect [6], a focus of some recent research projects on compiler optimizations has been the problem of selecting a likely optimal set of compiler optimizations for faster, smaller and less power-consuming object code. A multi-objective optimization formulation of the problem was proposed in [32]. A machine-learning approach using performance counters with machine learning that takes in consideration the underlying hardware was presented in [15]. Their work was enhanced to be included as part of the GCC compiler [22]. These are the so-called iterative compilation frameworks, where single flags are being added with the purpose of minimizing runtime of the benchmarks. The work in [32] tries to optimize many objectives simultaneously by exploring the search space instead of resorting to local search by adding flags iteratively. A manual mechanism to select the best set of flags for Java was presented in [33].

Our study of the influence of compiler optimizations on symbolic execution is driven by our overarching goal of making symbolic execution more efficient. To our knowledge, KLEE's *–optimize* flag is the first previous work that directly uses standard compiler optimizations in an attempt to achieve better performance for symbolic execution. The more recent Overify framework [58] introduces a suite of compiler optimizations specifically designed for making symbolic execution more efficient; the experimental results show Overify outperforms the standard optimization settings (i.e., $O0, \ldots, O3$) supported by KLEE (and LLVM). Our work differs from Overify in three basic ways: (1) our study is at a finer-grained level – we study how standard compiler optimizations influence symbolic execution *individually* (not just collectively as pre-defined optimization levels, such as $O2$); (2) our study uses three different back-end solvers (not just the default KLEE solver); (3) we study a new class of programs, namely NECLA benchmarks (not just the standard benchmarks of Coreutils). Recently, Cadar [11] independently reported similar findings with our paper. That work provided concrete examples and justifications for the adverse effect of some LLVM optimizations in KLEE. In contrast, our study presents a more rigorous empirical study involving different sets of real-world subject programs together with different constraint solver settings.

The idea of source-level program transformation to facilitate automated software testing was introduced by Harman et al. in their work on *testability transformation* [30], which focused on test input generation and test coverage criteria. They also introduced the idea of employing a transformation that may not be semantics preserving.

In the context of a declarative language, Marinov et al. presented transformations inspired by traditional compiler optimizations to optimize bounded-exhaustive analysis of declarative models written in the Alloy specification language and analyzed using the Alloy toolset's backend SAT solvers [43]. These transformations, while not being semantics preserving in general, produced an Alloy model that was equivalent to the original model with respect to the analysis bounds.

VII. Conclusion

The focus of our paper was to study the foundation of a promising approach for scaling symbolic execution – a classic program analysis with many traditional applications in software testing and verification, and most recently applied directly for software reliability. Specifically we studied the impact of compiler optimizations – which are designed for faster execution of programs on concrete inputs – on the performance of symbolic execution. As an enabling technology we used the KLEE symbolic execution engine and focused on 33 optimization flags of the LLVM compiler infrastructure that is used by KLEE. Specific research questions addressed were (1) how different optimizations influence the performance of symbolic execution for Unix Coreutils, (2) how the influence varies across two different program classes, and (3) how the influence varies across three different back-end constraint solvers. Some of our findings surprised us. For example, applying the 33 studied compiler optimizations in a pre-defined order can slow down symbolic execution (when using the basic depth-first search with constraint caching turned off) for Coreutils; moreover, in our experimental setup, applying no optimization performs better for many of the Coreutils. This is because existing compiler optimizations, which have been tailored to effectively generate faster code, display the inconvenient side-effect of altering its (typically branching) structure in a way that can make the program harder to analyze. Nevertheless, we believe compiler optimizations and techniques inspired by them have an important role to play in symbolic execution. We hope our work provides an effective foundation for better scalability of symbolic execution and using it more effectively for testing, verification, and reliability of real-world software systems.

Acknowledgments

This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-0845628 and CCF-1319688).

References

[1] GCC, the GNU compiler collection. http://gcc.gnu.org/.

[2] The GNU coverage tool. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[3] KLEE coreutils case study. http://klee.github.io/klee/TestingCoreutils.html.

[4] The KLEE symbolic virtual machine. http://klee.github.io/klee.

[5] NECLA static analysis benchmarks (necla-static-small). http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php#NECLA_Static_Analysis_Benchmarks.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, & tools*, volume 1009. Pearson/Addison Wesley, 2007.

[7] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Automatic exploit generation. In *NDSS*, 2011.

[8] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *TACAS*, pages 174–177. 2009.

[9] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446, 2008.

[10] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7), 2000.

[11] C. Cadar. Targeted program transformations for symbolic execution. In *FSE New Ideas Track*, pages 906–909, 9 2015.

[12] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, pages 209–224, 2008.

[13] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS*, pages 322–335, 2006.

[14] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *ICSE*, pages 1066–1071. Springer, 2011.

[15] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. OBoyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO*, pages 185–197, 2007.

[16] L. A. Clarke. A system to generate test data and symbolically execute programs. *TSE*, (3):215–222, 1976.

[17] K. D. Cooper and L. Torczon. *Engineering a Compiler*, volume 2. Morgan Kaufmann, 2011.

[18] R. Cytron, J. Ferrante, B. K. Rosen, Wegman, M. N., and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *TOPLAS*, volume 13, pages 451–490, 1991.

[19] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340. 2008.

[20] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *ICSE*, pages 622–631, 2013.

[21] A. Filieri, C. S. Pasareanu, and W. Visser. Reliability analysis in symbolic pathfinder: A brief summary. In *Software Engineering*, pages 39–40, 2014.

[22] G. Fursin, Y. Kashnikov, A. Wahid Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. K. I. Williams, and M. OBoyle. Milepost gcc: Machine learning enabled self-tuning compiler. In *International Journal of Parallel Programming*, volume 39, pages 296–327, 2011.

[23] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531, 2007.

[24] P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.

[25] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223, 2005.

[26] P. Godefroid, S. K. Lahiri, and C. Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*, pages 112–128, 2011.

[27] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, 2008.

[28] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, pages 43–56, 2010.

[29] C. F. Gross. *Machine Code Optimization - Improving Executable Object Code*, volume 1. Computer Science Department Technical Report No. 246. Courant Institute, New York University, 1986.

[30] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *TSE*, 30(1):3–16, 2004.

[31] T. Hönig, C. Eibel, R. Kapitza, and W. Schröder-Preikschat. Seep: Exploiting symbolic execution for energy-aware programming. In *Operating Systems Review*, volume 45, pages 58–62, 2011.

[32] K. Hoste and L. Eeckhout. Cole: compiler optimization level exploration. In *CGO*, volume 6, pages 165–174, 2008.

[33] K. Ishizaki, K. Kawachiya, T. T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and H. Nakatani. Efectiveness of cross-platform optimizations for a java just-in-time compiler. In *OOPSLA*, volume 3, pages 187–204, 2003.

[34] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jfuzz: A concolic whitebox fuzzer for java. In *NFM*, pages 121–125, 2008.

[35] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568. 2003.

[36] G. Kildall. A unified approach to global program optimization. In *POPL*, volume 1, pages 194–206, 1973.

[37] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.

[38] C. Lattner. *LLVM: An Infrastructure for Multi-Stage Optimization*. Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[39] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86, 2004.

[40] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. In *OOPSLA*, pages 19–32, 2013.

[41] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, pages 416–426, 2007.

[42] P. D. Marinescu and C. Cadar. Katch: high-coverage testing of software patches. In *FSE*, pages 235–245, 2013.

[43] D. Marinov, S. Khurshid, S. Bugrara, L. Zhang, and M. C. Rinard. Optimizations for compiling declarative models into boolean formulas. In *SAT*, volume 3569 of *LNCS*, pages 187–202, June 2005.

[44] D. Molnar, X. C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security*, pages 67–82, 2009.

[45] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In *CAV*, pages 53–68. Springer, 2013.

[46] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO*, pages 319–332, 2006.

[47] C. S. Păsăreanu and N. Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *ASE*, pages 179–180, 2010.

[48] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed Incremental Symbolic Execution. In *PLDI*, pages 504–515, 2011.

[49] Q.-S. Phan, P. Malacaria, C. S. Păsăreanu, and M. D'Amorim. Quantifying information leaks using reliability analysis. In *SPIN*, pages 105–108, 2014.

[50] R. Qiu, G. Yang, C. S. Pasareanu, and S. Khurshid. Compositional symbolic execution with memoized replay. In *ICSE*, 2015.

[51] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *FSE*, pages 263–272, 2005.

[52] J. H. Siddiqui and S. Khurshid. ParSym: Parallel symbolic execution. In *ICSTE*, pages V1–405–V1–409, Oct. 2010.

[53] J. H. Siddiqui and S. Khurshid. Scaling symbolic execution using ranged analysis. In *OOPSLA*, 2012.

[54] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. In *TOSEM*, volume 17, pages 1–10, 2008.

[55] M. Staats and C. Păsăreanu. Parallel Symbolic Execution for Structural Test Generation. In *ISSTA*, pages 183–194, 2010.

[56] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IPDPS*, pages 1–12, 2009.

[57] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *FSE*, 2012.

[58] J. Wagner, V. Kuznetsov, and G. Candea. Overify: optimizing programs for fast verification. In *HotOS*, pages 18–18, 2013.

[59] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *FSE*, pages 257–266, 2010.

[60] G. Yang, C. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *ISSTA*, pages 144–154, 2012.

[61] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *ICSM*, pages 1–10, 2010.