The Thesis Committee for Shiyu Dong
certifies that this is the approved version of the following thesis:

An Empirical Study of the Influence of Compiler Optimizations on Symbolic Execution

APPROVED BY

SUPERVISING COMMITTEE:

Sarfraz Khurshid, Supervisor

Dewayne Perry

An Empirical Study of the Influence of Compiler Optimizations on Symbolic Execution

by

Shiyu Dong

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2014

Dedicated to my father and mother, for all their support in the last 25 years.

Acknowledgments

I wish to thank the multitudes of people who helped me to complete my program of work. Without their help I would never have been able to finish my Master's degree.

First and foremost, I would like to express my sincere gratitude to my supervisor Prof. Sarfraz Khurshid for all his support for my study and research. Starting from December 2012, he offers tremendous help and guidance for both my study and research. His motivation, intelligence and immense knowledge really inspire me a lot. Also, as a supervisor he is very kind and always open to me. Every time I feel discouraged about the experiment result and afraid to face him he is always able to cheer me up. It is my great honor and pleasure to have an advisor like him. Besides my supervisor, I would like to thank my thesis reader Prof. Dewayne Perry. I really appreciate his time and effort in helping me with my thesis work.

Moreover, I would also like to thank for Lingming Zhang, for his generous help to my research. Lingming was one of the Ph.D students in our research group and he offers me continuous help during my research. His suggestions for doing empirical evaluation is of great value to me. Without his help it would be impossible for me to submit my first research paper.

My sincere thanks also goes to Oswaldo Olivo. He worked together with me and Lingming for our research paper. He is professional in compilers and constraint solving. I have learned a lot for him. His idea and knowledge is of invaluable help to our work.

I would like to thank UT-Austin and other funding agencies for supporting my degree. In particular, my work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-0845628, CCF-1319688, and CNS-0958231).

This thesis is in part based on a paper entitled "An Empirical Study of the Influence of Compiler Optimizations on Symbolic Execution" co-authored by Shiyu Dong, Oswaldo Olivo, Lingming Zhang and Sarfraz Khurshid, currently under submission for peer-review.

Last but not least, my thanks go for my father and mother. They are always on my back whenever I need whatever help that I need since I was born. I would not have been live a life as happy as I had without their love and all their selfless support.

An Empirical Study of the Influence of Compiler Optimizations on Symbolic Execution

Shiyu Dong, M.S.E.
The University of Texas at Austin, 2014

Supervisor: Sarfraz Khurshid

Compiler optimizations in the context of traditional program execution is a well-studied research area, and modern compilers typically offer a suite of optimization options. This thesis reports the first study (to our knowledge) on how standard compiler optimizations influence *symbolic* execution. We study 33 optimization flags of the LLVM compiler infrastructure, which are used by the KLEE symbolic execution engine. Specifically, we study (1) how different optimizations influence the performance of KLEE 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, KLEE's setting for applying the 33 optimizations in a pre-defined order provides sub-optimal performance for a majority of the Coreutils when using the basic depth-first search; moreover, in our experimental setup, applying no optimization performs better for many of the Coreutils.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Researchers have extensively studied *compiler optimizations*, i.e., semantics preserving program transformations that are designed to make program execution faster [1]. Modern compilers, such as `gcc` [2] and LLVM [3], support a number of *basic* as well as *aggressive* optimizations, and allow the users to manually select a suitable optimization *level* for their applications. 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 [4] [5].

While the area of compiler optimizations in the context of traditional program execution is well-studied, their use in the context of *symbolic* execution – a popular, systematic analysis technique for checking program behaviors using path exploration – has received much less attention. To our knowledge, the *KLEE* symbolic execution engine is the only tool to explicitly support compiler optimizations in the context of symbolic execution. *KLEE* 's foundation on the LLVM infrastructure [6] allows *KLEE* to directly access 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 [7] However, such reasoning is complicated by the fact that symbolic execution relies on an SMT solver [8] that is used to check for each path explored, the satisfiability of its *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 because the main bottleneck for symbolic execution is the SMT solving time and moreover, the SMT solver is 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 compiler optimizations and symbolic execution.

This thesis presents our study of 33 compiler optimizations implemented by the LLVM compiler and used by *KLEE* . Our choice of *KLEE* is driven by its industrial strength implementation and basis on the advanced compilation infrastructure of LLVM. Specifically, we study three core research questions. One, we study how different optimizations influence the performance of *KLEE* 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 [6]. Subsequently, *KLEE* was applied to test a number of more recent versions of Coreutils [9]. Two, we study how the influence of compiler optimizations varies across two different classes of programs: (1) Coreutils; and (2) NECLA benchmarks [10], which consist of smaller programs that are designed specifically to explore the strengths and weaknesses of static analyses. Three, we study how the influence of compiler optimizations varies across three different back-end constraint solvers that *KLEE* supports: *STP* [8], *Z3* [11], and *Boolector(Btor)* [12]. The *STP* solver is the primary solver for *KLEE* . Recent work on *KLEE* [13] added support for *Z3* and Btor. We conduct our study in the context of using depth-first search for symbolic execution, which is a deterministic search strategy employed by a number of standard symbolic execution tools, e.g., Symbolic PathFinder [14] for Java.

Our main findings are:

- Certain compiler optimizations influence symbolic execution more than the other optimizations. On average, applying optimizations makes symbolic executions worse for Coreutil 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 the two program classes. Specifically, optimizations help symbolic execution of 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 any better, 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*.

We hope our study provides a useful first step in motivating new research on investigating compiler optimizations for symbolic execution.

Chapter 2

Motivating Examples

This section presents two small examples to demonstrate that compiler optimizations can sometimes reduce and sometimes 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 2.1 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 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 2.2. We mark the change in grey.

In this case, symbolic execution before the optimization requires 63

```c
int main() {
  int a;
  klee_make_symbolic(&a,
      sizeof(a), "a");
  klee_assume(a > 0);
  klee_assume(a < 51);
  int x = 0;
  int y = 0;
  int i;
  for (i=0;i<a+1;i++)
    x = x + 3;
  for (i=0;i<a+1;i++)
    y = y + 4;
  return x + y;
}
```

```c
int main() {
  int a;
  klee_make_symbolic(&a,
      sizeof(a), "a");
  klee_assume(a > 0);
  klee_assume(a < 51);
  int x = 0;
  int y = 0;
  int i;
  for (i=0;i<a+1;i++) {
    x = x + 3;
    y = y + 4;
  }
  return x + y;
}
```

Figure 2.1: A compiler optimization example that reduces SMT queries for symbolic execution.

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.

```c
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;
}
```

```c
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;
}
```

Figure 2.2: A compiler optimization example that increases SMT queries for symbolic execution.

Chapter 3

Background

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

3.1 Symbolic Execution

Symbolic execution [15] 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 in 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. 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.

3.2 KLEE and LLVM

KLEE is a symbolic execution engine built on top of the compilation framework LLVM[16]. 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 with GCC. The IR is based on single-static assignment form (SSA)[17], 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, and benefiting 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 transformations.

3.3 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[18]. They have a natural implementation as a fix-point computation algorithm. These optimizations or similar ones have been described in traditional textbooks and dissertations [1][19][20].

Chapter 4

Experimental Study

In this section we will first define our research questions. Given these question we will then describe our design of a series of experiments, including independent and dependent variables. Then we will present and analysis the results and give our answer to the defined questions and our explanation in Section 5

4.1 Research Questions

We study the following research questions in this thesis:

RQ1: How do LLVM compiler optimizations influence KLEE's performance for Coreutil programs?

Given the above motivative 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 are generally good or bad to 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 if adding more optimization flags, or having a combination of optimization flags, leads to better or worse results, and if *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 solver 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.

RQ4: How robust are different solvers with respect to different com-

piler optimizations?

Another dimension that we want to study for constraint solvers is how robust the solvers are. In other words, we want to know if the best solver is always the best for different programs. A robust solver should perform good most of the time regardless of different optimizations, and we will compare the robustness for different solvers. Note that while RQ3 studies the influence of different solvers for compiler optimizations, here RQ4 focuses on the influence of different compiler optimizations for various solvers.

4.2 Experiment 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 [21]. The main changes that we have made for the setup is that we use 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.

4.2.1 Independent Variables

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

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 4.1: List of all Coreutil programs that we use in our experiment.

optimization option comes with *KLEE* . It applies 33 different optimization flags provided by LLVM in a certain order. The set of these 33 flags contains 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 thesis we will refer the *–optimize* option that comes with *KLEE* as *ALL* optimization.

Different programs. We study two different sets of programs, *Unix Coreutils 6.11*, and the *NECLA Static Analysis Benchmarks (necla-static-small)* [10]. 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* [6]. We conduct studies with similar setup as the one that used by most researchers and select 40 different Coreutils programs for experimental subjects, with the changes of search heuristic option mentioned above. Table 4.1 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 [22], 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 [23]. The NECLA benchmarks [10] are a traditional set of C benchmarks to test the performance of compilers. Normally they are of small sizes, loop intensive and they 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* support three different types of solvers: *STP*, *Z3* and *Btor*[13]. 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

4.2.2 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 complicate inputs as the Coreutils need. Therefore for these program we let the program finish execution. In this case, we measure number of instructions, time needed for execution and number of solver calls. When program finish execution, normally they will give the same coverage. Therefore we can compare the execution time and also the number of solver calls. With the same coverages shorter time and less number of solver calls means that symbolic execution performs better.

Chapter 5

Result Analysis

We design different experiment to address different research questions. We will describe them here in more detail, present the experiment results, and give our analysis and explanation.

5.1 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 good or bad for symbolic execution. Also, we believe that among all optimization flags, there are some flags which has more influences than the 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 if *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 sections.

5.1.1 Finding the determining flags

We combine all the 40 studied Coreutils programs with each of the 33 optimization flags mentioned in Section 4.2.1 , 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. Then, we simply divide the line coverage after each optimization by the line coverage of *NO*, and get a ratio. We consider it as a change if the ratio is not equal to 1, which means the line coverage after applying certain optimization is different from not applying any optimization. All flags we use and the number of changes that make are shown in Table 5.1, and the raw data is shown in Figure A.1 in Appendix A.

From Table 5.1 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 contributes 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 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 making symbolic execution better (ratio greater than 1) and the ones

Optimization	Changes	Optimization	Changes
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 5.1: Number of changes caused by applying individual flag.

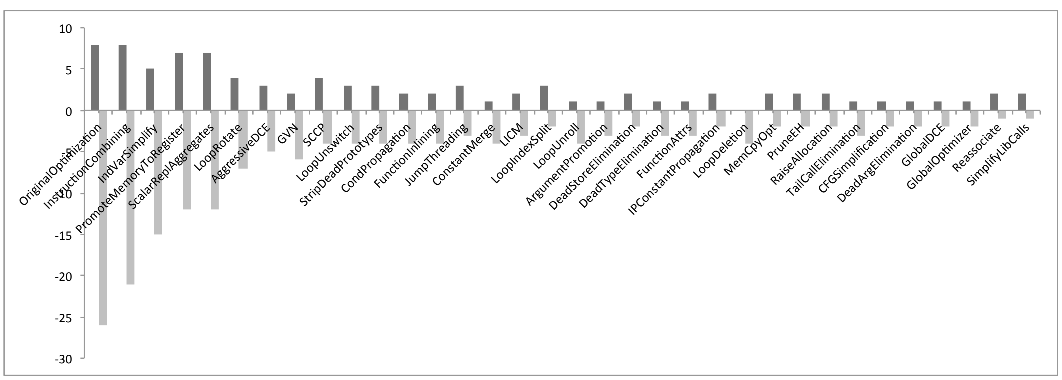


Figure 5.1: The influence of different compiler optimization flags to programs under test.

making symbolic execution worse (ratio smaller than 1). Figure 5.1 illustrates this separation. The x-axis represents different optimization flags. The bar above x-axis represents positive changes and below means negative changes. From this figure we could have a very interesting observation that a single flag tends to make symbolic execution worse rather than better, since for most programs the light gray area below x-axis is larger than the dark area above x-axis. This observation makes us think that, although compiler optimization is good for normal execution, it might be bad for symbolic execution in general even for a single flag. Then we designed more experiments to study if our assumption holds.

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

5.1.2 Analyzing the determining flags

Using the five determining flags from Section 5.1.1, we study more on the effect of them to symbolic execution. We run *KLEE* using *NO*, *ALL* and these five determining flags one by one, on all 40 Coreutils listed in Table 4.1, and limit the time to 5, 10, 20 and 30 minutes accordingly using 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

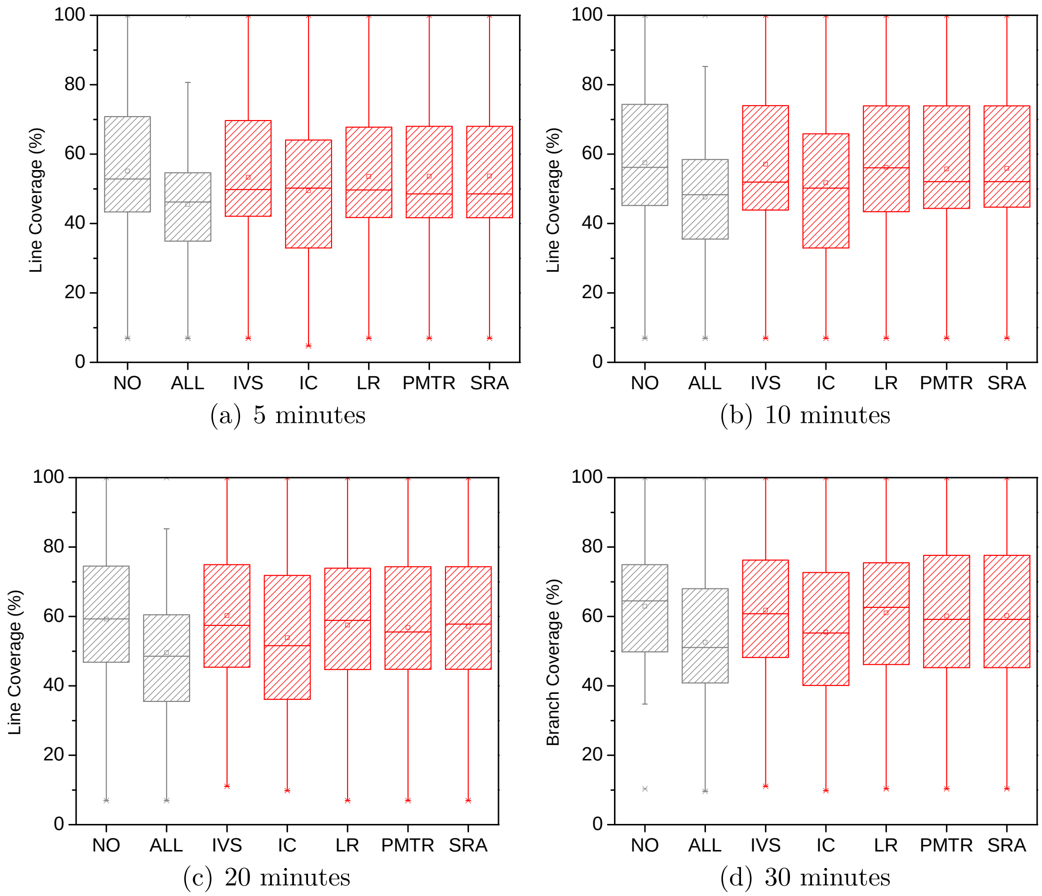
	5min	10min	20min	30min
ALL	0.826	0.827	0.837	0.835
IVS	0.968	0.993	1.017	0.982
IC	0.898	0.899	0.910	0.881
LR	0.972	0.976	0.971	0.969
PMTR	0.974	0.969	0.960	0.956
SRA	0.974	0.972	0.965	0.957

Table 5.2: Average line coverage ratio of each optimization flag.

optimization. Then we divide the new number by the "base" to get the ratio. All the raw data in this experiment is shown in Figure A.1-A.8 in Appendix A, and Table 5.2 and 5.3 list the average ratio of line and branch coverages for all Coreutil programs after applying each optimization flag. Here a number greater than 1 means it performs better than not applying any optimizations for a given time limit, and worse otherwise. We also list the the box plots of the actual line and branch coverage for this set of experiment in Figure 5.2 and 5.3. 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. We mark the results come with KLEE (*NO* and *ALL*) as gray and other five individual flags (*IVS*, *IC*, *LR*, *PMTR* and *SRA*) as red.

From the above two tables and corresponding box plots, we can make some interesting observations:

First, although the performance for an individual optimization on symbolic execution varies from program to program according to the box plot, the



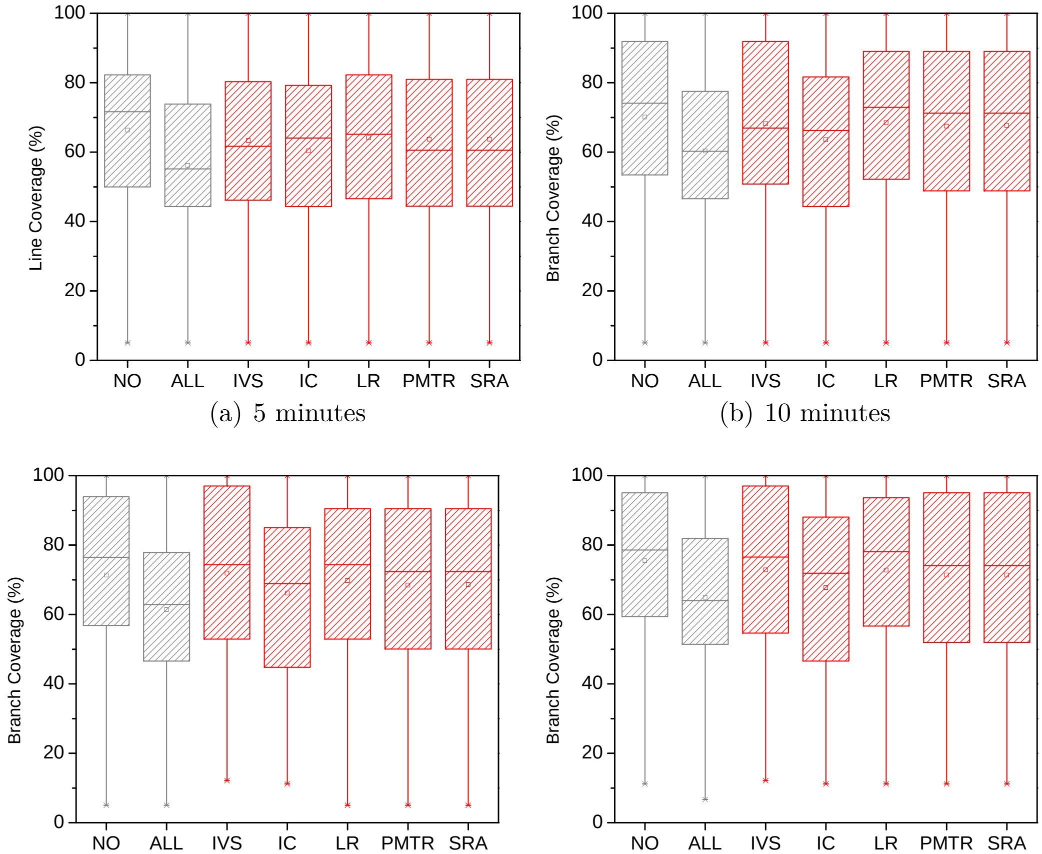
(a) 5 minutes

(b) 10 minutes

(c) 20 minutes

(d) 30 minutes

Figure 5.2: Box plot for line coverage of the Coreutil experiment.



(a) 5 minutes

(b) 10 minutes

(c) 20 minutes

(d) 30 minutes

Figure 5.3: Box plot for branch coverage of the Coreutil experiment.

	5min	10min	20min	30min
ALL	0.847	0.861	0.860	0.860
IVS	0.955	0.973	1.007	0.965
IC	0.910	0.908	0.927	0.897
LR	0.967	0.977	0.977	0.965
PMTR	0.960	0.963	0.959	0.946
SRA	0.960	0.965	0.962	0.946

Table 5.3: Average branch coverage ratio of each optimization flag.

average result shows the trend that compiler optimization is in general making symbolic execution worse for Coreutils programs. In both tables, most of the average ratios is smaller than 1, which indicates that the average coverage of applying optimizations is not as good as not applying any optimizations. Similar observations can be made in the box plots, where *NO* gives better average coverage for all time limitations than all the other single compiler optimization flags and the *ALL* optimization. This result is pretty surprising to us, since *KLEE* 's documentation[7] 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. Here are the results from running again with –optimize enabled"

It seems that the authors of *KLEE* simply applies compiler optimizations without a very deep understanding about how exactly those optimization will work for symbolic execution. In any case, our experiment result does not support the previous quote.

Second, from the result 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 only gives an average ratio of 0.835, while the ratio for all single flags are grater than it. The worst single flag, *IC*, gives a ratio 0.881, and all the others give a ratio greater than 0.9. 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 experiment 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 experiment setup. This observation leads us to think about maybe it is possible that 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.

5.1.3 Study on different optimization combinations

In order to study the effect of compiler optimization combination on symbolic execution, we first come up with four different combinations based on our knowledge. We combine the determining flags that we find out from

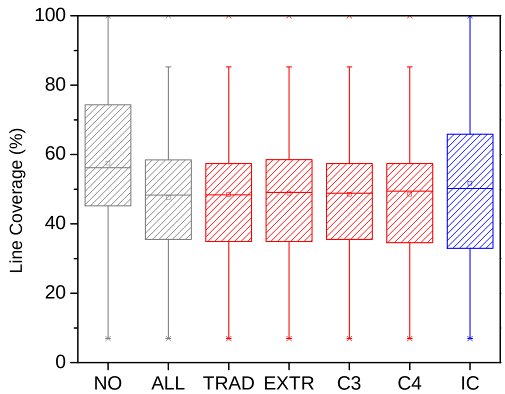
Name	Flags
TRAD	*PMTR*, *SRA*, *IVS*, *LR*, *IC*, *PMTR*
EXTR	*PMTR*, *IC*, *SRA*, *IC*, *LR*, *IC*, *IVS*, *IC*, *SRA*, *IC*, *PMTR*
C3	*PMTR*, *IC*, *SRA*, *IC*
C4	*LR*, *IVS*, *PMTR*, *IC*, *SRA*, *IC*

Table 5.4: List of different combinations.

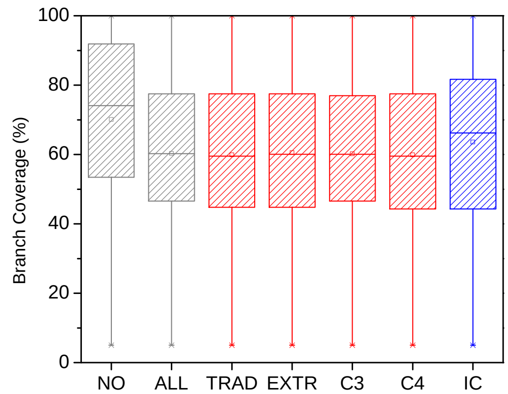
previous experiments in different orders. One flag may appear more than once.

The first combination(*TRAD*) is an arrangement according to the traditional compiler optimization order. The idea is from the famous "Dragon Book" of compiler[1]. 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 5.4 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 previous experiments, we calculate the ratio between the line coverage or branch coverage of the optimized execution to the original execution with *NO* and put the results in Table 5.5. We also put the result of the worst single flag, *IC*, in this table in order to make further comparison. For the average coverage line and branch coverage, we also draw similar box plots in Figure 5.4.



(a) Line Coverage



(b) Branch Coverage

Figure 5.4: Box plot for line and branch coverage of the Coreutil experiment for different combinations.

	Line Coverage	Branch Coverage
ALL	0.843	0.855
TRAD	0.849	0.864
EXTR	0.844	0.859
C3	0.843	0.855
C4	0.827	0.861
IC	0.899	0.908

Table 5.5: Average line and branch coverage ratio of each flag combination and each individual flag.

We mark *NO* and *ALL* that come with *KLEE* in gray, our four combinations *TRAD*, *EXTR*, *C3* and *C4* in red, and the worst single flag *IC* in blue. All the raw data for this experiment is shown in Figure A.9 and A.10 in Appendix A.

From Table 5.5 we can see that none of the optimization gives an average ratio greater than 1, which indicates that applying a combination of optimizations tends to make symbolic execution worse. From Figure 5.4 if we compare the box of *ALL* with the other combinations that we have, we could see that they are 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 experiment setup we have the following findings:

First, on average, given limited time and using DFS search heuristic, compiler optimizations are making symbolic execution for Coreutil programs worse if we measure line coverage and branch coverage.

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

Third, applying a combination of optimizations is not a good choice. According to our experiment, 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.

5.2 RQ2: Result consistency across different program classes

In section 5.1 we studied the effect of compiler optimizations on symbolic execution based on the Coreutil programs, which is a set of real-world programs and usually with complicated structure. In this section we will conduct similar experiments on another class of programs. We choose a small set of NECLA Static Analysis Benchmarks [10]. These benchmarks are C based programs that are often used to test the performance of compilers. Many of them are loop intensive and they perform integer and array operations for most of the time.

For all the following experiments in this section we use similar options as those for Coreutil programs. We use DFS search heuristic and we disable caching of *KLEE* . However, because NECLA are very simple programs that *KLEE* can finish symbolic execution soon and give the same coverage, in this experiment we let *KLEE* finish executing instead of specifying a time limit. We record the number of instructions, time for each execution, and number of solver calls. Again we use our modified version of *KLEE* so that it can take a certain optimization flag or flag combinations as an argument.

	Flag	ex2	ex3	ex8	ex9	ex30	ex34
Instrs	NO	12423	6077	5189	11511	6851	5381
	ALL	2174	1884	1888	2216	1945	1929
	IC	5188	5895	5153	5285	5234	5232
	PMTR	8505	4014	3334	8592	4369	3436
	SRA	8505	4014	3334	8592	4369	3436
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 5.6: Symbolic execution result for applying single optimization on several NECLA benchmarks.

5.2.1 Single optimization flag

We first apply *NO*, *ALL*, and all 33 individual optimization flags with 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, number of solver calls or number of instructions. 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 5.6.

From Table 5.6 we can observe that, different from the previous Coreutil experiment, 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 contradicts 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 jobs 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 experiment setup *IC* might be the most significant determining flag. The other two optimizations only change the number of instructions, while all the remaining flags that are not listed in Table 5.6 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 may indicate that there might be several determining optimization flags in common across different program classes, which have more influence on symbolic execution than the others. This observation could encourage us

	Flag	ex2	ex3	ex8	ex9	ex30	ex34
Instrs	I+P	3380	3926	3352	3465	3389	3386
	I+S	3380	3926	3352	3465	3389	3386
	P+I	3380	3926	3352	3465	3389	3386
	P+S	8505	4014	3334	8592	4369	3436
	S+I	3380	3926	3352	3465	3389	3386
	S+P	8505	4014	3334	8592	4369	3436
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 5.7: Symbolic execution result for two optimization flags on several NECLA benchmarks.

to do further study.

5.2.2 Multiple optimization flags

Similar to the previous experiment, we also want to explore the effect of different flags 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 5.7. Note that in this table, for simplicity we use *I* for *IC*, *P* for *PMTR* and *S* for *SRA*.

Table 5.7 shows the result after a combination two out of the three determining flags. There are two observations that we could make for this case. First, applying one more flag on the most significant determining flag (which is *IC* in our case) does not help any more 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 functionally 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 or might not be applicable to real life programs. That is the main reason why some of the results in this experiment is not in accordance to the results from the Coreutils experiment. Actually, even for the Coreutil experiments there are some cases where compiler optimizations help with symbolic execution. Therefore, we cannot yet claim that compiler optimization is good or bad for symbolic execution for a specific program, since the program itself is still an important factor. However, we think in future works if we could characterize different programs into different categories, and study the effect of different optimizations on each categories, 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.

5.3 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 make it to support different constraint solvers [13]. 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* [13]. 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 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 is not the same set as the above Coreutil examples, therefore we may see the difference average coverages against previous Coreutil experiments. The raw data is in Figure B.1 and B.2 in Appendix B.

First we want to study the influence of different solvers on different optimizations, and to see if the performance of different optimizations is consistent

	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 5.8: 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 5.9: Average branch coverage of each optimization solver pair for all programs.

across different solvers. From the data that we have, we calculate the average line and branch coverages of all 11 programs, for each optimization-solver pair, and list them in Table 5.8 and 5.9.

From Table 5.8 and 5.9 we can see that the behavior of different optimization flags are slightly different working with different solvers, because the coverages for the same optimization are different for different solvers. However, their relatively behavior are still the same. Same as 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 optimization is consistent among different solvers.

5.4 RQ4: Robustness of solvers to optimizations

Using the same data from section 5.3, we also want to study the performance of different solvers for symbolic execution, and to see whether the result of the solver changes a lot or not after applying different optimizations. In particular, we want to know how "robust" different solvers are. In other words, we want to know for the same program whether the best solver is always the best or not across different optimizations.

In Table 5.10 we calculate the average line and branch coverage for each program-solver pair for different optimization flags. We also take the average coverage for each solver and put that at the bottom of the table. From these two tables, if we take a look at both the average line and branch coverage, on average *STP* gives the best coverage compared with *Z3* and *Btor*. The results here are consistent with the findings mentioned in [13]. In their work they showed that *STP* performs the best, or is more robust, among the three solvers given unlimited time, and in our experiment we can show similar result given limited time. Therefore we can strengthen the result that *STP* is currently the known best solver for symbolic execution. Also, since there is one solver already performs good, if we could find out the inner interaction between symbolic execution and constraint solvers, it is very possible to design better solvers specifically for symbolic execution.

Another observation we can make here is that although *STP* performs

	Line Coverage			Branch Coverage		
Program	STP	Z3	Btor	STP	Z3	Btor
base64	45.58	43.13	41.90	50.16	45.08	43.49
chmod	60.45	58.96	57.89	70.63	69.83	68.49
comm	70.41	64.43	63.26	87.35	86.12	86.12
csplit	38.38	38.38	45.79	41.74	41.74	50.48
dircolors	73.76	63.91	34.51	83.57	70.71	28.57
echo	67.13	66.99	67.68	74.39	74.39	74.39
env	77.14	77.14	77.14	96.10	96.10	96.10
factor	66.10	67.38	58.21	85.71	88.09	50.00
join	17.49	14.25	33.04	20.37	17.24	34.75
ln	65.24	59.50	58.03	72.03	69.77	69.47
fifo	66.26	66.26	53.50	89.14	89.14	77.71
Average	58.90	56.39	53.72	70.11	68.02	61.78

Table 5.10: Average line and branch coverage of each program-solver pair for all optimizations.

the best on average, there are cases where other solvers performs better. For example, for *csplit*, *STP* and *Z3* are not as good as *Btor*. Again, the actual result of symbolic execution really depends on the program executed. Different programs have different characteristics and may result in different behaviors for a specific optimization or solver. Therefore, in the future if we could categorize programs according to their characteristics, and find the best optimization flags and solvers for those characteristics, it is very likely that we could have a more effective way to make symbolic execution performs much better for a certain class of programs.

5.5 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., longer time limitations 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.

Chapter 6

Discussions and Future Work

Our experimental study investigates the relationship between compiler optimization and symbolic execution. The following sections list the limitations for our current work as well as the dimensions that our current work can be extended.

6.1 Discussions

6.1.1 Search heuristics

All of our pervious experiments are based on the DFS search heuristic in *KLEE* . We choose that in order to remove the nondeterminism as much as possible. However, it turns out that DFS is not the best search heuristic for symbolic execution. For example, if a program contains a loop, DFS tends to go into the loop and symbolic execution might stuck inside the loop, and therefore gives a low coverage. In fact, *KLEE* by default uses random search heuristic instead of DFS. It is more practical and may give better coverage since it could avoid the problem that we have mentioned above. However, when we try the random search *KLEE* gives significantly different results every time even if we use exactly the same setup and arguments. There is a tradeoff between which heuristic to use, to get better coverage or less nondeterminism. Since we would like to study the difference between different compiler optimizations, the nondeterminism of each run must be as low as possible, and that is why we choose DFS for our experiment.

6.1.2 Execution time

Another possible thing to do is to increase the time that we choose to run symbolic execution. For each run, we use 5 minutes to find the determining flag, 10 minutes to get the result for combination of flags and different solvers, and for individual determining flags we use 5, 10, 20 and 30 minutes for all 40 coreutils. We have to admit that the time is not long enough for some cases. However, since the scale of our experiments is very large, we cannot afford an hour for each run given limited time. For example, if we want to explore all coreutils to get the result for one hour execution, it will take 40 hours, plus the time to generate test cases and outputs, for one single flag, and we have to do the same thing for each time interval and each individual flag. In the future, we plan to evaluate symbolic execution using more and longer time constraints.

6.1.3 Number of optimizations and programs

The optimization flags that we explored are from the source code of *KLEE* , and it is a list of all optimization flags that *KLEE* has used for its own optimizations. LLVM has more optimization flags than our list, and it is possible that some flags not included in our list will have some impact on symbolic execution as well. There are two reasons why we do not include a full list of optimizations. First, similar to what we mentioned above, the time constraint is a concern to us. More optimizations means more experiments, and due to the time limitation we cannot afford that. Also, some of the non-added optimization flags are rarely used even for modern compilers, or they have many preconditions in order to be applied. Given more time we could do the same experiment for the full optimization list. However, in the scope of this empirical study we prefer not to use all of them, and we think it is a reasonable tradeoff to choose. For the coreutil programs that we have chosen, the way we choose coreutils is based on related researches and we use the programs that other people uses. By doing this we can guarantee that our experiment result is not biased and we can also compare our result with previous ones as we did for the multi-solver study. Again, in the future we plan to explore all the coreutils that *KLEE* supports, but we think our current choice is enough for this study.

6.2 Future Work

6.2.1 Finding the best flag combination

As our experiment result shows, in general adding more optimization flags may make symbolic execution worse. However, that is only a average statistical observation and for certain programs there are some optimization combinations which could make the symbolic execution better. We have tried several algorithms to dynamically search for the best single individual flag, and tried to add more flags on the previous best one(s). However, these algorithms turn out to be not as effective as we expected if we use the DFS search algorithm given short execution time, and it will be too expensive for longer time. We think if we can find a way to perform a static analysis of the program, it is possible to get some information and generate a better combination of optimization for a certain program.

6.2.2 Mutation testing

Due to its capability of simulating real faults, mutation testing has been widely used to evaluate testing techniques. Since our execution time is relatively short and we are using DFS search algorithm, *KLEE* cannot generate a lot of test cases. We tried to perform mutation testing using the test cases generated after running *KLEE* using DFS for 20 minutes, but only few test cases kill the mutants. We think DFS is the main reason why *KLEE* cannot explore many paths of the program and generate effective test cases, and it is also possible that the mutation testing score will be better if we run *KLEE* long enough. In any case, in the future we would expand the dimension of this study to include mutation testing.

6.2.3 Deeper reasoning

All previous experiments are just empirical studies without knowledge about compiler optimization nor the program itself. The results give the trend of the impact compiler optimizations have on symbolic execution, without very deep reasoning. In the future for some cases where compiler optimization makes symbolic execution better or worse, we would try to reason why certain improvements or decrements happen by looking into the program and the optimization itself. If we could find something in common for similar behaviors, it is possible to draw some conclusion based on that. For example, loop intensive programs may behave very differently from program without loops, and we can summarize the characteristics for program with loops and try to find out the best optimization(s) for program with this characteristic. Furthermore, with more knowledge about these characteristics, we can even design a better solver or a better optimization for symbolic execution, since from our experiment the overall performance of different solvers are different. If any of the above two possibilities can be further explored, it is very likely that the performance of symbolic execution can be greatly increased.

Chapter 7

Related Work

While traditional work on compiler optimizations defined optimizations and studied their effect [1], 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[24]. A machine-learning approach using performance counters with machine learning that takes in consideration the underlying hardware was presented in[25]. Their work was enhanced to be included as part of the gcc compiler[26]. 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 [24] 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 [27]. We are not aware of any study of the effect of compiler flags in symbolic execution.

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 only previous work that uses compiler optimizations in an attempt to achieve better performance. There exist a number of other techniques for optimizing symbolic execution.

A couple of recent research projects proposed *parallel* symbolic execution [28–30]. Static partitioning [28] 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 [29] 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 [31, 32], it requires communicating symbolic constraints for every branch explored among workers, which incurs a higher overhead. Ranged symbolic execution [30] 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 [33], 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 [34] also enables compositional symbolic execution [35] as well as compositional dynamic test generation [36] 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 [37] 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 [38] 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 re-use 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 [39]; 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 interface [40] wraps every call to the solver to check if the result is already available in its database of previous solver calls.

Chapter 8

Conclusion

This thesis reported the first study (to our knowledge) on how standard compiler optimizations influence *symbolic* execution. We studied 33 optimization flags of the LLVM compiler infrastructure, which are used by the KLEE symbolic execution engine. Specifically, we studied (1) how different optimizations influence the performance of KLEE 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, KLEE's setting for applying the 33 optimizations in a pre-defined order provides sub-optimal performance for a majority of the Coreutils when using the basic depth-first search; 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 for decades to effectively generate faster code, display the inconvenient side-effect of altering its (typically branching) structure in a way that makes the program harder to analyze. Nevertheless, we believe compiler optimizations have an important role to play in symbolic execution. In future work, we plan to explore the problem of defining effective compiler optimizations that are designed in the specific context of symbolic execution and perhaps fine-tuned with respect to specific program constructs, such as branch conditions, and control-flow structures, such as loops.

Appendices

Appendix A

Raw data for Coreutil experiments

	NO	ALL	IVS	IC	LR	PMTR	SRA
base64	40	14.29	36.19	63.81	40	40	40
basename	79.49	79.49	79.49	79.49	79.49	79.49	79.49
chcon	43.59	10.77	44.62	12.31	43.59	43.59	43.59
cksum	80.65	80.65	80.65	80.65	80.65	80.65	80.65
comm	73.47	46.94	69.39	55.1	59.18	58.16	58.16
cut	48.65	48.99	47.64	48.65	48.65	48.65	48.65
dd	10.34	9.63	10.34	9.8	10.34	10.34	10.34
dircolors	73.68	56.84	70	67.89	73.68	73.68	73.68
dirname	48.39	48.39	48.39	74.19	48.39	48.39	48.39
du	49.34	55.3	49.34	41.72	49.34	59.6	59.6
env	77.78	51.11	82.22	82.22	82.22	82.22	82.22
expand	41.06	44.37	41.06	49.01	41.06	39.07	39.07
expr	41.12	40.83	41.12	41.42	41.12	40.83	40.83
fold	64.6	47.79	45.13	49.56	64.6	41.59	41.59
groups	81.08	62.16	81.08	59.46	81.08	81.08	81.08
link	64.29	35.71	64.29	64.29	64.29	64.29	64.29
logname	56	52	56	56	56	56	56
mkdir	57.58	34.85	54.55	34.85	50	50	50
mkfifo	68.09	36.17	74.47	55.32	68.09	68.09	68.09
mknod	30	35	27.5	23.75	40	40	40
nice	50.85	57.63	50.85	50.85	42.37	42.37	42.37
nl	51.18	43.6	44.08	54.5	51.18	41.71	41.71
od	50.21	34.04	50.21	29.96	31.93	31.65	31.65
paste	67.38	45.45	48.13	37.43	67.38	67.91	67.91
pathchk	62.88	31.06	59.85	31.06	47.73	47.73	47.73
printf	28.79	35.41	27.63	67.7	12.84	42.02	43.58
readlink	76	54	76	46	76	76	76
rmdir	43.06	27.78	43.06	27.78	43.06	43.06	43.06
setuidgid	38.96	23.38	23.38	23.38	32.47	32.47	32.47
sleep	45.65	43.48	45.65	45.65	45.65	45.65	45.65
split	44.7	35.02	37.33	42.86	44.7	44.7	44.7
sum	86.32	85.26	86.32	85.26	86.32	86.32	86.32
sync	100	100	100	100	100	100	100
tee	75.36	59.42	75.36	75.36	75.36	75.36	75.36
touch	54.48	51.72	52.41	30.34	54.48	57.93	57.93
tr	25.34	34.45	39.91	4.7	34.29	34.14	34.14
tsort	6.9	6.9	6.9	6.9	6.9	6.9	6.9
unexpand	46.91	48.45	48.45	56.7	46.91	44.85	44.85
unlink	60	60	60	60	60	60	60
wc	59.16	51.91	53.44	52.29	59.92	58.78	58.78

Table A.1: Line coverage of different optimizations on different Coreutils for 5 minutes.

	NO	ALL	IVS	IC	LR	PMTR	SRA
base64	44.76	14.29	36.19	73.33	44.76	44.76	44.76
basename	79.49	79.49	79.49	79.49	79.49	79.49	79.49
chcon	46.67	10.77	47.69	12.31	46.67	46.67	46.67
cksum	80.65	80.65	80.65	80.65	80.65	80.65	80.65
comm	74.49	54.08	73.47	55.1	62.24	61.22	62.24
cut	48.65	48.99	47.64	48.65	48.65	48.65	48.65
dd	10.34	9.63	11.05	9.8	10.34	10.34	10.34
dircolors	73.68	71.05	70	67.37	73.68	73.68	73.68
dirname	74.19	48.39	100	100	74.19	74.19	74.19
du	56.29	59.27	56.29	42.05	56.29	59.6	59.6
env	77.78	51.11	82.22	100	82.22	82.22	82.22
expand	41.06	44.37	43.71	49.01	41.06	39.07	39.07
expr	48.52	48.22	41.72	48.22	42.31	42.01	48.22
fold	64.6	46.9	45.13	49.56	64.6	41.59	41.59
groups	81.08	62.16	81.08	59.46	81.08	81.08	81.08
link	64.29	42.86	67.86	64.29	64.29	64.29	64.29
logname	56	52	56	56	56	56	56
mkdir	56.06	34.85	65.15	34.85	56.06	50	50
mkfifo	74.47	36.17	74.47	55.32	74.47	74.47	74.47
mknod	43.75	33.75	51.25	25	43.75	50	50
nice	50.85	57.63	50.85	50.85	42.37	42.37	42.37
nl	51.18	45.02	44.08	60.66	51.18	46.92	46.92
od	50.21	38.82	50.21	29.96	31.93	31.65	31.65
paste	67.91	45.45	48.13	37.43	67.91	68.98	68.98
pathchk	64.39	31.06	60.61	31.06	47.73	47.73	47.73
printf	38.52	64.98	39.69	71.6	42.41	43.97	42.8
readlink	78	54	78	46	78	78	78
rmdir	43.06	27.78	43.06	27.78	43.06	54.17	54.17
setuidgid	40.26	23.38	40.26	23.38	32.47	32.47	32.47
sleep	45.65	43.48	45.65	45.65	45.65	45.65	45.65
split	44.7	44.24	37.33	42.86	44.7	44.7	44.7
sum	86.32	85.26	86.32	85.26	86.32	86.32	86.32
sync	100	100	100	100	100	100	100
tee	75.36	59.42	75.36	75.36	75.36	75.36	75.36
touch	66.21	51.72	65.52	30.34	66.21	66.21	66.21
tr	28.07	34.45	39.91	23.98	34.29	34.14	34.14
tsort	6.9	6.9	6.9	6.9	6.9	6.9	6.9
unexpand	46.91	48.45	48.45	56.7	46.91	44.85	44.85
unlink	60	60	60	60	60	60	60
wc	59.16	51.53	52.67	52.29	59.92	58.78	58.78

Table A.2: Line coverage of different optimizations on different Coreutils for 10 minutes.

	NO	ALL	IVS	IC	LR	PMTR	SRA
base64	44.76	14.29	36.19	73.33	44.76	44.76	44.76
basename	79.49	79.49	79.49	79.49	79.49	79.49	79.49
chcon	46.67	10.77	48.21	12.31	46.67	46.67	46.67
cksum	80.65	80.65	80.65	80.65	80.65	80.65	80.65
comm	74.49	46.94	74.49	55.1	73.47	73.47	73.47
cut	48.65	48.99	47.64	48.65	48.65	48.65	48.65
dd	10.34	9.63	11.05	9.8	10.34	10.34	10.34
dircolors	73.68	71.58	70.53	67.89	73.68	76.32	76.32
dirname	100	100	100	100	74.19	74.19	74.19
du	58.61	55.3	58.61	45.7	58.61	59.6	59.6
env	77.78	51.11	82.22	100	82.22	82.22	82.22
expand	41.06	44.37	43.71	49.01	41.06	39.07	39.07
expr	50.3	60.95	60.95	50	60.95	50	60.65
fold	64.6	46.9	45.13	49.56	64.6	41.59	41.59
groups	81.08	62.16	81.08	59.46	81.08	81.08	81.08
link	64.29	42.86	67.86	64.29	64.29	64.29	64.29
logname	56	52	56	56	56	56	56
mkdir	71.21	34.85	71.21	34.85	65.15	65.15	65.15
mkfifo	74.47	36.17	74.47	55.32	74.47	74.47	74.47
mknod	55	33.75	56.25	25	55	55	55
nice	50.85	57.63	50.85	50.85	42.37	42.37	42.37
nl	51.66	45.02	44.08	60.66	51.66	46.92	46.92
od	50.21	41.21	50.21	29.96	31.93	31.65	31.65
paste	67.91	48.66	48.13	37.43	67.91	68.98	68.98
pathchk	64.39	31.06	60.61	31.06	48.48	48.48	48.48
printf	39.3	66.54	39.3	71.6	42.41	45.14	45.14
readlink	78	58	78	46	78	78	78
rmdir	43.06	27.78	43.06	27.78	43.06	54.17	54.17
setuidgid	49.35	23.38	40.26	23.38	32.47	32.47	32.47
sleep	45.65	43.48	45.65	45.65	45.65	45.65	45.65
split	44.7	44.24	37.33	42.86	44.7	44.7	44.7
sum	86.32	85.26	86.32	85.26	86.32	86.32	86.32
sync	100	100	100	100	100	100	100
tee	75.36	75.36	75.36	75.36	75.36	75.36	75.36
touch	66.9	51.72	66.21	30.34	66.9	66.21	66.21
tr	28.07	34.45	39.91	26.1	34.29	34.14	34.14
tsort	6.9	6.9	89.66	73.4	6.9	6.9	6.9
unexpand	46.91	48.45	48.45	56.7	46.91	44.85	44.85
unlink	60	60	68	72	60	60	60
wc	59.92	50.76	52.67	52.29	59.16	58.78	59.54

Table A.3: Line coverage of different optimizations on different Coreutils for 20 minutes.

	NO	ALL	IVS	IC	LR	PMTR	SRA
base64	44.76	14.29	36.19	73.33	44.76	44.76	44.76
basename	79.49	79.49	79.49	79.49	79.49	79.49	79.49
chcon	46.67	10.77	48.21	12.31	46.67	46.67	46.67
cksum	80.65	80.65	80.65	80.65	80.65	80.65	80.65
comm	73.47	46.94	73.47	55.1	75.51	82.65	82.65
cut	50.68	51.01	49.32	50.68	50.68	48.65	48.65
dd	10.34	9.63	11.05	9.8	10.34	10.34	10.34
dircolors	73.68	71.58	70.53	67.89	73.68	77.89	77.89
dirname	100	100	100	100	100	100	100
du	70.2	55.3	70.2	45.7	70.2	59.6	59.6
env	77.78	51.11	82.22	100	82.22	82.22	82.22
expand	41.06	44.37	43.71	49.01	41.06	39.07	39.07
expr	60.95	60.95	60.95	59.76	60.95	60.65	60.65
fold	64.6	46.9	45.13	49.56	64.6	41.59	41.59
groups	81.08	62.16	81.08	62.16	81.08	81.08	81.08
link	64.29	42.86	64.29	64.29	64.29	64.29	64.29
logname	56	52	56	56	56	56	56
mkdir	71.21	34.85	65.15	34.85	65.15	65.15	65.15
mkfifo	74.47	36.17	74.47	55.32	74.47	74.47	74.47
mknod	57.5	33.75	56.25	25	56.25	55	56.25
nice	50.85	57.63	50.85	50.85	42.37	42.37	42.37
nl	54.98	45.02	44.08	63.51	54.98	46.92	46.92
od	50.21	38.82	50.21	29.96	31.93	31.65	31.65
paste	67.91	48.66	48.13	37.43	67.91	68.98	68.98
pathchk	64.39	64.39	60.61	64.39	48.48	48.48	48.48
printf	73.93	73.93	73.54	76.65	57.59	50.97	51.36
readlink	78	58	78	46	78	78	78
rmdir	43.06	27.78	54.17	27.78	43.06	54.17	54.17
setuidgid	49.35	23.38	40.26	23.38	32.47	32.47	32.47
sleep	45.65	43.48	45.65	45.65	45.65	45.65	45.65
split	44.7	44.24	37.33	42.86	44.7	44.7	44.7
sum	86.32	85.26	86.32	85.26	86.32	86.32	86.32
sync	100	100	100	100	100	100	100
tee	75.36	75.36	85.51	75.36	75.36	75.36	75.36
touch	66.9	51.72	65.52	30.34	66.9	66.21	66.21
tr	34.75	34.45	40.21	28.53	34.29	34.14	34.14
tsort	77.34	73.4	93.1	76.85	77.34	77.34	77.34
unexpand	46.91	48.45	48.45	56.7	46.91	44.85	44.85
unlink	68	72	68	72	68	68	68
wc	59.16	51.53	52.67	53.05	59.16	58.78	58.78

Table A.4: Line coverage of different optimizations on different Coreutils for 30 minutes.

	NO	ALL	IVS	IC	LR	PMTR	SRA
base64	42.22	6.67	44.44	66.67	42.22	42.22	42.22
basename	79.49	79.49	79.49	79.49	79.49	79.49	79.49
chcon	43.59	10.77	44.62	12.31	43.59	43.59	43.59
cksum	80.65	80.65	80.65	80.65	80.65	80.65	80.65
comm	73.47	46.94	69.39	55.1	59.18	58.16	58.16
cut	48.65	48.99	47.64	48.65	48.65	48.65	48.65
dd	10.34	9.63	10.34	9.8	10.34	10.34	10.34
dircolors	81.25	63.75	80	78.75	81.25	81.25	81.25
dirname	80	80	80	100	80	80	80
du	67.37	70.53	67.37	58.95	67.37	71.58	71.58
env	100	72.73	100	100	100	100	100
expand	57.14	55.1	53.06	65.31	57.14	51.02	51.02
expr	53.59	53.59	53.59	53.59	53.59	53.59	53.59
fold	83.56	57.53	57.53	67.12	83.56	54.79	54.79
groups	89.47	89.47	89.47	78.95	89.47	89.47	89.47
link	83.33	66.67	83.33	83.33	83.33	83.33	83.33
logname	75	75	75	75	75	75	75
mkdir	80.65	45.16	80.65	45.16	74.19	74.19	74.19
mkfifo	92	48	100	76	92	92	92
mknod	45.45	54.55	36.36	40	52.73	52.73	52.73
nice	50	65	50	50	40	40	40
nl	60.34	46.55	46.55	66.38	60.34	43.1	43.1
od	59.76	48.05	59.76	42.44	42.2	42.2	42.2
paste	84.25	59.84	62.99	40.16	84.25	84.25	84.25
pathchk	77.89	44.21	77.89	44.21	56.84	56.84	56.84
printf	28.97	34.58	28.97	68.22	7.94	44.39	44.39
readlink	100	100	100	92	100	100	100
rmdir	56.52	39.13	56.52	39.13	56.52	56.52	56.52
setuidgid	50	22.73	22.73	22.73	36.36	36.36	36.36
sleep	44.44	44.44	44.44	44.44	44.44	44.44	44.44
split	62.86	41.43	41.43	62.86	62.86	62.86	62.86
sum	100	100	100	100	100	100	100
sync	100	100	100	100	100	100	100
tee	95.74	80.85	95.74	95.74	95.74	95.74	95.74
touch	70.08	70.08	66.93	48.03	70.08	73.23	73.23
tr	27.45	36.6	45.75	5.01	36.17	36.17	36.17
tsort	5	5	5	5	5	5	5
unexpand	59.7	55.22	55.22	73.13	59.7	52.24	52.24
unlink	80	80	80	80	80	80	80
wc	73.18	59.22	60.34	59.22	73.18	73.18	73.18

Table A.5: Branch coverage of different optimizations on different Coreutils for 5 minutes.

	NO	ALL	IVS	IC	LR	PMTR	SRA
base64	46.67	6.67	44.44	86.67	46.67	46.67	46.67
basename	100	100	100	100	100	100	100
chcon	54.17	11.11	54.17	23.61	54.17	54.17	54.17
cksum	100	100	100	100	100	100	100
comm	94.29	71.43	94.29	77.14	88.57	88.57	88.57
cut	51.61	52.42	51.61	52.42	51.61	51.61	51.61
dd	11.17	11.17	12.14	11.17	11.17	11.17	11.17
dircolors	81.25	81.25	80	77.5	81.25	81.25	81.25
dirname	100	80	100	100	100	100	100
du	72.63	71.58	72.63	60	72.63	71.58	71.58
env	100	72.73	100	100	100	100	100
expand	57.14	55.1	55.1	65.31	57.14	51.02	51.02
expr	60.22	60.22	53.59	60.22	53.59	53.59	60.22
fold	83.56	60.27	57.53	67.12	83.56	54.79	54.79
groups	89.47	89.47	89.47	78.95	89.47	89.47	89.47
link	83.33	66.67	83.33	83.33	83.33	83.33	83.33
logname	75	75	75	75	75	75	75
mkdir	80.65	45.16	87.1	45.16	80.65	74.19	74.19
mkfifo	100	48	100	76	100	100	100
mknod	52.73	54.55	70.91	40	52.73	70.91	70.91
nice	50	65	50	50	40	40	40
nl	60.34	48.28	46.55	70.69	60.34	46.55	46.55
od	59.76	56.59	59.76	42.44	42.2	42.2	42.2
paste	84.25	59.84	62.99	40.16	84.25	85.83	85.83
pathchk	82.11	44.21	77.89	44.21	56.84	56.84	56.84
printf	38.79	70.09	38.79	71.03	46.26	44.39	44.39
readlink	100	100	100	92	100	100	100
rmdir	56.52	39.13	56.52	39.13	56.52	60.87	60.87
setuidgid	50	22.73	50	22.73	36.36	36.36	36.36
sleep	44.44	44.44	44.44	44.44	44.44	44.44	44.44
split	62.86	62.86	41.43	62.86	62.86	62.86	62.86
sum	100	100	100	100	100	100	100
sync	100	100	100	100	100	100	100
tee	95.74	80.85	95.74	95.74	95.74	95.74	95.74
touch	77.95	70.08	76.38	48.03	77.95	77.95	77.95
tr	27.89	36.6	45.75	23.31	36.17	36.17	36.17
tsort	5	5	5	5	5	5	5
unexpand	59.7	55.22	55.22	73.13	59.7	52.24	52.24
unlink	80	80	80	80	80	80	80
wc	73.18	59.22	60.34	59.22	73.18	73.18	73.18

Table A.6: Branch coverage of different optimizations on different Coreutils for 10 minutes.

	NO	ALL	IVS	IC	LR	PMTR	SRA
base64	46.67	6.67	44.44	86.67	46.67	46.67	46.67
basename	100	100	100	100	100	100	100
chcon	54.17	11.11	54.17	23.61	54.17	54.17	54.17
cksum	100	100	100	100	100	100	100
comm	94.29	62.86	94.29	77.14	91.43	91.43	91.43
cut	51.61	52.42	51.61	52.42	51.61	51.61	51.61
dd	11.17	11.17	12.14	11.17	11.17	11.17	11.17
dircolors	81.25	81.25	80	78.75	81.25	86.25	86.25
dirname	100	100	100	100	100	100	100
du	73.68	70.53	73.68	62.11	73.68	71.58	71.58
env	100	72.73	100	100	100	100	100
expand	57.14	55.1	55.1	65.31	57.14	51.02	51.02
expr	60.22	67.96	67.96	60.22	67.96	60.22	67.96
fold	83.56	60.27	57.53	67.12	83.56	54.79	54.79
groups	89.47	89.47	89.47	78.95	89.47	89.47	89.47
link	83.33	66.67	83.33	83.33	83.33	83.33	83.33
logname	75	75	75	75	75	75	75
mkdir	93.55	45.16	93.55	45.16	87.1	87.1	87.1
mkfifo	100	48	100	76	100	100	100
mknod	78.18	54.55	78.18	40	78.18	78.18	78.18
nice	50	65	50	50	40	40	40
nl	60.34	48.28	46.55	70.69	60.34	46.55	46.55
od	59.76	57.8	59.76	42.44	42.2	42.2	42.2
paste	84.25	62.99	62.99	40.16	84.25	85.83	85.83
pathchk	82.11	44.21	77.89	44.21	56.84	56.84	56.84
printf	42.52	75.7	42.52	71.03	46.26	49.07	49.07
readlink	100	100	100	92	100	100	100
rmdir	56.52	39.13	56.52	39.13	56.52	60.87	60.87
setuidgid	59.09	22.73	50	22.73	36.36	36.36	36.36
sleep	44.44	44.44	44.44	44.44	44.44	44.44	44.44
split	62.86	62.86	41.43	62.86	62.86	62.86	62.86
sum	100	100	100	100	100	100	100
sync	100	100	100	100	100	100	100
tee	95.74	95.74	95.74	95.74	95.74	95.74	95.74
touch	77.95	70.08	77.95	48.03	77.95	77.95	77.95
tr	27.89	36.6	45.75	26.14	36.17	36.17	36.17
tsort	5	5	98.33	81.67	5	5	5
unexpand	59.7	55.22	55.22	73.13	59.7	52.24	52.24
unlink	80	80	100	100	80	80	80
wc	73.18	59.22	60.34	59.22	73.18	73.18	73.18

Table A.7: Branch coverage of different optimizations on different Coreutils for 20 minutes.

	NO	ALL	IVS	IC	LR	PMTR	SRA
base64	46.67	6.67	44.44	86.67	46.67	46.67	46.67
basename	100	100	100	100	100	100	100
chcon	54.17	11.11	54.17	23.61	54.17	54.17	54.17
cksum	100	100	100	100	100	100	100
comm	94.29	62.86	94.29	77.14	91.43	94.29	94.29
cut	53.23	55.65	52.42	55.65	53.23	51.61	51.61
dd	11.17	11.17	12.14	11.17	11.17	11.17	11.17
dircolors	81.25	81.25	80	78.75	81.25	88.75	88.75
dirname	100	100	100	100	100	100	100
du	78.95	70.53	78.95	62.11	78.95	71.58	71.58
env	100	72.73	100	100	100	100	100
expand	57.14	55.1	55.1	65.31	57.14	51.02	51.02
expr	67.96	67.96	67.96	65.75	67.96	67.96	67.96
fold	83.56	60.27	57.53	67.12	83.56	54.79	54.79
groups	89.47	89.47	89.47	89.47	89.47	89.47	89.47
link	83.33	66.67	83.33	83.33	83.33	83.33	83.33
logname	75	75	75	75	75	75	75
mkdir	93.55	45.16	87.1	45.16	87.1	87.1	87.1
mkfifo	100	48	100	76	100	100	100
mknod	78.18	54.55	78.18	40	78.18	78.18	78.18
nice	50	65	50	50	40	40	40
nl	63.79	48.28	46.55	70.69	63.79	46.55	46.55
od	59.76	56.59	59.76	42.44	42.2	42.2	42.2
paste	84.25	62.99	62.99	40.16	84.25	85.83	85.83
pathchk	82.11	82.11	77.89	82.11	56.84	56.84	56.84
printf	76.64	78.5	76.64	74.77	59.81	53.74	53.74
readlink	100	100	100	92	100	100	100
rmdir	56.52	39.13	60.87	39.13	56.52	60.87	60.87
setuidgid	59.09	22.73	50	22.73	36.36	36.36	36.36
sleep	44.44	44.44	44.44	44.44	44.44	44.44	44.44
split	62.86	62.86	41.43	62.86	62.86	62.86	62.86
sum	100	100	100	100	100	100	100
sync	100	100	100	100	100	100	100
tee	95.74	95.74	95.74	95.74	95.74	95.74	95.74
touch	77.95	70.08	76.38	48.03	77.95	77.95	77.95
tr	40.09	36.6	46.19	26.14	36.17	36.17	36.17
tsort	83.33	81.67	98.33	81.67	83.33	83.33	83.33
unexpand	59.7	55.22	55.22	73.13	59.7	52.24	52.24
unlink	100	100	100	100	100	100	100
wc	73.18	59.22	60.34	59.22	73.18	73.18	73.18

Table A.8: Branch coverage of different optimizations on different Coreutils for 30 minutes.

	NO	ALL	TRAD	EXTR	C3	C4	IC
base64	44.76	14.29	14.29	14.29	14.29	14.29	73.33
basename	79.49	79.49	79.49	79.49	79.49	79.49	79.49
chcon	46.67	10.77	10.77	10.77	10.77	10.77	12.31
cksum	80.65	80.65	80.65	80.65	80.65	80.65	80.65
comm	74.49	54.08	47.96	55.1	48.98	55.1	55.1
cut	48.65	48.99	48.65	48.65	48.65	48.65	48.65
dd	10.34	9.63	9.63	9.63	9.63	9.63	9.8
dircolors	73.68	71.05	71.58	71.05	71.58	71.58	67.37
dirname	74.19	48.39	100	100	100	100	100
du	56.29	59.27	55.3	55.3	55.3	55.3	42.05
env	77.78	51.11	51.11	51.11	51.11	51.11	100
expand	41.06	44.37	45.03	45.03	43.71	45.03	49.01
expr	48.52	48.22	48.22	48.22	48.22	49.41	48.22
fold	64.6	46.9	46.9	46.02	46.02	46.9	49.56
groups	81.08	62.16	62.16	62.16	59.46	62.16	59.46
link	64.29	42.86	42.86	42.86	39.29	39.29	64.29
logname	56	52	52	52	52	52	56
mkdir	56.06	34.85	34.85	34.85	34.85	28.79	34.85
mkfifo	74.47	36.17	36.17	36.17	36.17	36.17	55.32
mknod	43.75	33.75	33.75	33.75	33.75	33.75	25
nice	50.85	57.63	50.85	57.63	50.85	50.85	50.85
nl	51.18	45.02	50.24	50.24	50.24	50.24	60.66
od	50.21	38.82	41.21	41.21	41.21	41.21	29.96
paste	67.91	45.45	45.45	45.45	45.99	45.99	37.43
pathchk	64.39	31.06	31.06	31.06	31.06	31.06	31.06
printf	38.52	64.98	68.87	68.87	64.2	69.65	71.6
readlink	78	54	54	54	54	54	46
rmdir	43.06	27.78	27.78	27.78	27.78	27.78	27.78
setuidgid	40.26	23.38	23.38	23.38	23.38	23.38	23.38
sleep	45.65	43.48	43.48	43.48	43.48	43.48	45.65
split	44.7	44.24	35.02	35.02	47.47	35.02	42.86
sum	86.32	85.26	85.26	85.26	85.26	85.26	85.26
sync	100	100	100	100	100	100	100
tee	75.36	59.42	59.42	59.42	59.42	59.42	75.36
touch	66.21	51.72	51.72	51.72	51.72	51.72	30.34
tr	28.07	34.45	34.45	34.45	34.14	34.14	23.98
tsort	6.9	6.9	6.9	6.9	6.9	6.9	6.9
unexpand	46.91	48.45	48.45	49.48	49.48	49.48	56.7
unlink	60	60	60	60	60	60	60
wc	59.16	51.53	50.76	50.76	50.76	50.76	52.29

Table A.9: Line coverage of different optimization combinations on different Coreutils for 10 minutes.

	NO	ALL	TRAD	EXTR	C3	C4	IC
base64	46.67	6.67	6.67	6.67	6.67	6.67	86.67
basename	100	100	100	100	100	100	100
chcon	54.17	11.11	11.11	11.11	11.11	11.11	23.61
cksum	100	100	100	100	100	100	100
comm	94.29	71.43	62.86	71.43	65.71	71.43	77.14
cut	51.61	52.42	52.42	52.42	52.42	52.42	52.42
dd	11.17	11.17	11.17	11.17	11.17	11.17	11.17
dircolors	81.25	81.25	81.25	81.25	81.25	81.25	77.5
dirname	100	80	100	100	100	100	100
du	72.63	71.58	73.68	73.68	70.53	70.53	60
env	100	72.73	72.73	72.73	72.73	72.73	100
expand	57.14	55.1	55.1	55.1	55.1	55.1	65.31
expr	60.22	60.22	60.22	60.22	60.22	60.22	60.22
fold	83.56	60.27	60.27	60.27	60.27	60.27	67.12
groups	89.47	89.47	89.47	89.47	78.95	89.47	78.95
link	83.33	66.67	66.67	66.67	66.67	66.67	83.33
logname	75	75	75	75	75	75	75
mkdir	80.65	45.16	45.16	45.16	45.16	38.71	45.16
mkfifo	100	48	48	48	48	48	76
mknod	52.73	54.55	54.55	54.55	54.55	54.55	40
nice	50	65	50	65	50	50	50
nl	60.34	48.28	50	50	50	50	70.69
od	59.76	56.59	57.8	57.8	57.8	57.8	42.44
paste	84.25	59.84	59.84	59.84	59.84	59.84	40.16
pathchk	82.11	44.21	44.21	44.21	44.21	44.21	44.21
printf	38.79	70.09	72.9	72.9	70.09	72.9	71.03
readlink	100	100	100	100	100	100	92
rmdir	56.52	39.13	39.13	39.13	39.13	39.13	39.13
setuidgid	50	22.73	22.73	22.73	22.73	22.73	22.73
sleep	44.44	44.44	44.44	44.44	44.44	44.44	44.44
split	62.86	62.86	41.43	41.43	65.71	41.43	62.86
sum	100	100	100	100	100	100	100
sync	100	100	100	100	100	100	100
tee	95.74	80.85	80.85	80.85	80.85	80.85	95.74
touch	77.95	70.08	71.65	71.65	71.65	71.65	48.03
tr	27.89	36.6	36.6	36.6	36.6	36.6	23.31
tsort	5	5	5	5	5	5	5
unexpand	59.7	55.22	53.73	55.22	53.73	53.73	73.13
unlink	80	80	80	80	80	80	80
wc	73.18	59.22	59.22	59.22	59.22	59.22	59.22

Table A.10: Branch coverage of different optimization combinations on different Coreutils for 10 minutes.

Appendix B

Raw Data for Multi-slover Experiment

Program	Solver	NO	ALL	IVS	IC	LR	PMTR	SRA
base64	STP	48.57	14.29	46.67	63.81	48.57	48.57	48.57
base64	Z3	44.76	14.29	40.95	63.81	44.76	44.76	48.57
base64	Btor	44.76	14.29	36.19	63.81	44.76	44.76	44.76
chmod	STP	70.52	41.04	58.96	41.04	70.52	70.52	70.52
chmod	Z3	68.79	41.04	57.23	41.04	67.05	68.79	68.79
chmod	Btor	67.05	41.04	56.65	41.04	68.79	65.32	65.32
comm	STP	74.49	46.94	74.49	55.1	80.61	80.61	80.61
comm	Z3	74.49	46.94	74.49	55.1	67.35	66.33	66.33
comm	Btor	73.47	54.08	73.47	55.1	62.24	62.24	62.24
csplit	STP	48.26	60.18	11.19	4.22	48.26	48.26	48.26
csplit	Z3	48.26	60.18	11.19	4.22	48.26	48.26	48.26
csplit	Btor	52.48	51.74	52.84	4.22	52.48	53.39	53.39
dircolors	STP	73.68	74.21	73.68	65.26	73.68	77.89	77.89
dircolors	Z3	73.68	71.58	13.68	67.37	73.68	73.68	73.68
dircolors	Btor	39.47	40	10	33.68	39.47	39.47	39.47
echo	STP	66.99	67.96	66.99	66.99	66.99	66.99	66.99
echo	Z3	66.99	66.99	66.99	66.99	66.99	66.99	66.99
echo	Btor	66.99	67.96	67.96	66.99	67.96	67.96	67.96
env	STP	77.78	51.11	82.22	100	82.22	82.22	82.22
env	Z3	77.78	51.11	82.22	100	82.22	82.22	82.22
env	Btor	77.78	51.11	82.22	82.22	82.22	82.22	82.22
factor	STP	65.67	64.18	71.64	64.18	65.67	65.67	65.67
factor	Z3	71.64	61.19	71.64	64.18	71.64	65.67	65.67
factor	Btor	58.21	58.21	58.21	58.21	58.21	58.21	58.21
join	STP	5.44	61.45	5.44	26.98	12.24	5.44	5.44
join	Z3	5.44	59.18	5.44	13.38	5.44	5.44	5.44
join	Btor	20.86	45.35	20.41	11.79	28.57	52.15	52.15
ln	STP	76.8	38.14	77.84	30.93	77.32	77.84	77.84
ln	Z3	64.95	38.14	65.98	30.93	64.95	75.77	75.77
ln	Btor	64.95	38.14	65.98	30.93	64.95	70.62	70.62
mkfifo	STP	74.47	36.17	74.47	55.32	74.47	74.47	74.47
mkfifo	Z3	74.47	36.17	74.47	55.32	74.47	74.47	74.47
mkfifo	Btor	55.32	36.17	74.47	55.32	55.32	48.94	48.94

Table B.1: Line coverage for each program-solver-flag tuple for 10 minutes.

Program	Solver	NO	ALL	IVS	IC	LR	PMTR	SRA
base64	STP	55.56	6.67	55.56	66.67	55.56	55.56	55.56
base64	Z3	46.67	6.67	46.67	66.67	46.67	46.67	55.56
base64	Btor	46.67	6.67	44.44	66.67	46.67	46.67	46.67
chmod	STP	85.05	46.73	64.49	46.73	85.05	83.18	83.18
chmod	Z3	85.05	46.73	62.62	46.73	81.31	83.18	83.18
chmod	Btor	83.18	46.73	62.62	46.73	85.05	77.57	77.57
comm	STP	94.29	62.86	94.29	77.14	94.29	94.29	94.29
comm	Z3	94.29	62.86	94.29	77.14	91.43	91.43	91.43
comm	Btor	94.29	71.43	94.29	77.14	88.57	88.57	88.57
csplit	STP	52.22	63.7	13.33	6.3	52.22	52.22	52.22
csplit	Z3	52.22	63.7	13.33	6.3	52.22	52.22	52.22
csplit	Btor	56.67	58.89	57.04	6.3	56.67	58.89	58.89
dircolors	STP	81.25	86.25	82.5	76.25	81.25	88.75	88.75
dircolors	Z3	81.25	81.25	11.25	77.5	81.25	81.25	81.25
dircolors	Btor	32.5	33.75	6.25	30	32.5	32.5	32.5
echo	STP	74.39	74.39	74.39	74.39	74.39	74.39	74.39
echo	Z3	74.39	74.39	74.39	74.39	74.39	74.39	74.39
echo	Btor	74.39	74.39	74.39	74.39	74.39	74.39	74.39
env	STP	100	72.73	100	100	100	100	100
env	Z3	100	72.73	100	100	100	100	100
env	Btor	100	72.73	100	100	100	100	100
factor	STP	83.33	88.89	94.44	83.33	83.33	83.33	83.33
factor	Z3	94.44	83.33	94.44	83.33	94.44	83.33	83.33
factor	Btor	50	50	50	50	50	50	50
join	STP	7.54	71.48	7.54	27.54	13.44	7.54	7.54
join	Z3	7.54	68.85	7.54	14.1	7.54	7.54	7.54
join	Btor	19.02	49.84	19.02	13.44	25.9	58.03	58.03
ln	STP	80.95	47.62	82.01	35.98	80.95	88.36	88.36
ln	Z3	77.78	47.62	78.84	35.98	77.78	85.19	85.19
ln	Btor	77.78	47.62	78.84	35.98	77.78	84.13	84.13
mkfifo	STP	100	48	100	76	100	100	100
mkfifo	Z3	100	48	100	76	100	100	100
mkfifo	Btor	84	48	100	76	84	76	76

Table B.2: Branch coverage for each program-solver-flag tuple for 10 minutes.

Bibliography

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*, volume 1009. Pearson/Addison Wesley, 2007.

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

[3] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[4] Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pages 12–pp. IEEE, 2006.

[5] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[6] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted

and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

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

[8] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, pages 519–531. Springer, 2007.

[9] Paul Dan Marinescu and Cristian Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 235–245. ACM, 2013.

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

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

[12] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[13] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Computer Aided Verification*, pages 53–68. Springer, 2013.

[14] Corina S Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 179–180. ACM, 2010.

[15] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

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

[17] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Wegman, Mark N., and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *ACM Transactions on Programming Languages and Systems*, volume 13, pages 451–490, 1991.

[18] G. Kildall. A unified approach to global program optimization. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, volume 1, pages 194–206, 1973.

[19] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*, volume 2. Morgan Kaufmann, 2011.

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

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

[22] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. In *OOPSLA*, pages 19–32, 2013.

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

[24] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *CGO*, volume 6, pages 165–174, 2008.

[25] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP OB́oyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 185–197, 2007.

[26] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael OB́oyle. Milepost gcc: Machine learning enabled self-tuning compiler. In *International Journal of Parallel Programming*, volume 39, pages 296–327, 2011.

[27] 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 *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 3, pages 187–204, 2003.

[28] M. Staats and C. Păsăreanu. Parallel Symbolic Execution for Structural Test Generation. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 183–194, 2010.

[29] Junaid Haroon Siddiqui and Sarfraz Khurshid. ParSym: Parallel symbolic execution. In *Proc. 2nd International Conference on Software Technology and Engineering (ICSTE)*, pages V1–405–V1–409, San Juan, PR, October 2010.

[30] Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. In *Proc. 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2012.

[31] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated Whitebox Fuzz Testing. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2008.

[32] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *13th ACM conference on Computer and communications security*, pages 322–335, 2006.

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

[34] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *Proc. 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 43–56, 2010.

[35] Patrice Godefroid. Compositional dynamic test generation. In *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54, 2007.

[36] Patrice Godefroid, Shuvendu K. Lahiri, and Cindy Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *Proc. 18th International Conference on Static Analysis*, pages 112–128, 2011.

[37] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed Incremental Symbolic Execution. In *PLDI*, pages 504–515, 2011.

[38] G. Yang, C. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 144–154, 2012.

[39] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unas-

sisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, pages 209–224, 2008.

[40] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proc. ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, 2012.

Vita

Shiyu Dong was born in Tianjin, China. After completing his work in Tianjin No.1 Middle School in Tianjin, China, in 2008, he entered Shanghai Jiao Tong University in Shanghai, China. He earned the Bachelor of Science in Engineering degree in Electrical and Computer Engineering there on August 2012, before he entered the Graduate School at the University of Texas at Austin.

Permanent address: 1300 Crossing Place
Austin, Texas 78741

This thesis was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.