Learning to Accelerate Symbolic Execution via Code Transformation

Junjie Chen
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
chenjunjie@pku.edu.cn

Wenxiang Hu
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
huwx@pku.edu.cn

Lingming Zhang
Department of Computer Science, University of Texas at Dallas, 75080, USA
lingming.zhang@utdallas.edu

Dan Hao[1]
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
haodan@pku.edu.cn

Sarfraz Khurshid
Department of Electrical and Computer Engineering, University of Texas at Austin, 78712, USA
khurshid@ece.utexas.edu

Lu Zhang
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, EECS, Peking University, Beijing, 100871, China
zhanglucs@pku.edu.cn

Abstract

Symbolic execution is an effective but expensive technique for automated test generation. Over the years, a large number of refined symbolic execution techniques have been proposed to improve its efficiency. However, the symbolic execution efficiency problem remains, and largely limits the application of symbolic execution in practice. Orthogonal to refined symbolic execution, in this paper we propose to accelerate symbolic execution through semantic-preserving code transformation on the target programs. During the initial stage of this direction, we adopt a particular code transformation, compiler optimization, which is initially proposed to accelerate program concrete execution by transforming the source program into another semantic-preserving target program with increased efficiency (e.g., faster or smaller). However, compiler optimizations are mostly designed to accelerate program *concrete* execution rather than *symbolic* execution. Recent work also reported that unified settings on compiler optimizations that can accelerate symbolic execution for any program do not exist at all. Therefore, in this work we propose a machine-learning based approach to tuning compiler optimizations to accelerate symbolic execution, whose results may also aid further design of specific code transformations for symbolic execution. In particular, the proposed approach LEO separates source-code functions and libraries through our program-splitter, and predicts individual compiler optimization (i.e., whether a type of code transformation is chosen) separately through analyzing the performance of existing symbolic execution. Finally, LEO applies symbolic execution on the code transformed by

[1] corresponding author.

© Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang;
licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein; Article No. 6; pp. 6:1–6:27

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

compiler optimization (through our local-optimizer). We conduct an empirical study on GNU Coreutils programs using the KLEE symbolic execution engine. The results show that LEO significantly accelerates symbolic execution, outperforming the default KLEE configurations (i.e., turning on/off all compiler optimizations) in various settings, e.g., with the default training/testing time, LEO achieves the highest line coverage in 50/68 programs, and its average improvement rate on all programs is 46.48%/88.92% in terms of line coverage compared with turning on/off all compiler optimizations.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases Symbolic Execution, Code Transformation, Machine Learning

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.6

Funding This work is partially supported by the National Key Research and Development Program of China (Grant No. 2017YFB1001803), and NSFC 61672047, 61529201, 61522201, 61861130363. This work is also partially supported by NSF Grant No. CCF-1566589, UT Dallas start-up fund, Google, Huawei, and Samsung. This work is also partially supported by NSF Grant No. CCF-1704790.

1 Introduction

Symbolic execution is a systematic analysis methodology to explore program behaviors, and has been widely used in test input generation [29, 15, 30]. In particular, symbolic execution takes test inputs as symbolic values instead of concrete values so as to generate test inputs by solving the constraints for program paths. Although symbolic execution facilitates test generation to a large extent, it is widely recognized to suffer from the efficiency problem due to the exponential number of paths and constraint solving cost. To relieve the efficiency problem of symbolic execution, various optimization techniques have been proposed, e.g., compositional symbolic execution [41, 72], incremental symbolic execution [79, 88], and parallel symbolic execution [78, 76]. However, symbolic execution remains one of the most expensive testing methodologies [18].

Instead of refining symbolic execution techniques, in this paper, we aim to accelerate symbolic execution via another orthogonal dimension – transforming the programs under test. Intuitively, if a program under test can be transformed into a semantic-preserving but easy-to-analyze program, the efficiency of symbolic execution will be improved. Moreover, all the refined symbolic execution techniques will be also further improved because of the orthogonality. That is, semantic-preserving code transformation rules for symbolic execution are needed. However, few semantic-preserving code transformation rules studied in the literature targets at symbolic execution, and designing such rules is a complex process and will be a long-term project. During the initial stage of this direction, we borrow code transformation rules for concrete execution to learn code transformation rules for symbolic execution, because of the substantial knowledge accumulated over 30 years in the field of concrete execution as well as the similarity between concrete execution and symbolic execution. In particular, we borrow compiler optimization, which is one of the most mature code transformation approaches to transforming the source program into another semantic-preserving target program with increased efficiency (e.g., faster or smaller) and has been widely recognized by its effectiveness on accelerating *concrete* execution [2, 32, 38, 27].

Since compiler optimizations are specially designed for compilers to optimize program *concrete* execution, they may reduce the efficiency of *symbolic* execution due to the difference between concrete and symbolic execution. As reported by recent work [33, 14], some compiler optimizations indeed largely accelerate symbolic execution for some programs, but some compiler optimizations even make symbolic execution much slower for some programs. Moreover, there is no unified configuration on the compiler optimizations guaranteeing the efficiency of symbolic execution for all programs. If we can learn how to utilize compiler optimizations to accelerate symbolic execution for each individual program, it will become a very light-wight approach to accelerating symbolic execution via code transformation, and is also helpful in designing specific effective code transformation rules for symbolic execution. Therefore, in this paper we focus on *learning to tune compiler optimizations* to accelerate symbolic execution.

In particular, we propose the first machine-**LE**arning-based approach to tuning compiler **O**ptimizations for symbolic execution (abbreviated as **LEO**). LEO tunes compiler optimizations for each code portion (e.g., each function) of a program individually rather than for the whole program, because compiler optimizations transform different code portions in different ways. More specifically, for any program under test, LEO first divides it into source-code portions and libraries used in the program, and then learns their settings on compiler optimizations separately. Library optimizations can be directly applied with the corresponding compiler. To enable different code portions with different settings on compiler optimizations, we design and implement two components. The first one is program-splitter, which splits a program into multiple files so that each file contains only one source-code portion (e.g., function). The second one is local-optimizer, which optimizes each preceding file by its learnt compiler optimization settings. With these tools, LEO integrates the optimized files and optimized libraries into a fine-optimized program using the LLVM linker. Such fine-optimized program is semantically equivalent with the original program, and is treated as inputs of symbolic execution engines instead of the original one, so as to accelerate symbolic execution.

To evaluate LEO, we conduct an empirical study on KLEE using the widely used GNU Coreutils programs [57, 86, 33, 15]. Our experimental study shows that compared with two default settings of KLEE (i.e., symbolic execution without any code transformations – turning off all compiler optimizations, and symbolic execution turning on all compiler optimizations), LEO achieves the highest line coverage in 50/68 programs, indicating its great performance on accelerating symbolic execution. In particular, compared with symbolic execution without any code transformations (i.e., turning off all compiler optimizations), the average improvement rate of LEO on all programs is 88.92% in terms of line coverage, demonstrating that code transformation is indeed a promising direction to accelerate symbolic execution. Moreover, compared with symbolic execution turning on all compiler optimizations, the average improvement rate of LEO on all programs is 46.48% in terms of line coverage, indicating that effectively tunning compiler optimizations is a successful exploration in this direction and our machine-learning based approach is able to predict better compiler-optimization settings for accelerating symbolic execution. Furthermore, the compiler optimizations recommended by LEO with some specified training symbolic execution time (e.g., the default 10-minute) can always significantly outperform the default settings of KLEE in most cases even when the testing symbolic execution time increases.

```
1 int fun2(int N, int h[10]){
2    int i;
3 ---for(i=0;i<N-2;++i) {
4 ---   if(i%2==0) h[i] = 1;
5 ---   else h[i]=0;
6 ---}
7    for(i=0;i<N;++i) {
8       h[i]=2*i;
9    }
10   int sum=0;
11   for(i=0;i<N;++i)
12      sum+=h[i];
13   return sum;
14 }
```

(a) Acceleration.

```
1 int fun1(int M, int g[10]) {
2    int i;
3 ---for(i=0;i<M;++i) {
  +++for(i=M-2;i<M;++i) {
4       g[i]=i*i;
5    }
6    for(i=0;i<M-2;++i) {
7       g[i]=0;
8    }
9    int sum=0;
10   for(i=0;i<M;++i)
11      sum+=g[i];
12   return sum;
13 }
```

(b) Deceleration.

Figure 1 Motivating examples.

The contributions of this paper are summarized as follows.

- The first approach to accelerating symbolic execution via machine-learning based compiler optimization tuning for code transformation.
- An implementation of the proposed approach, including program-splitter and local-optimizer components, enabling the learnt compiler optimization settings for different code portions.
- An extensive study on GNU Coreutils programs demonstrating the performance of LEO on accelerating symbolic execution as well as the contributions of various components of LEO.

2 Motivation

In this section, we use two examples of aggressive dead code elimination (ADCE) to illustrate the motivation of this work, i.e., tunning compiler optimization can accelerate symbolic execution. ADCE is a compiler optimization that assumes all instructions are dead unless they are proven not and tries to eliminate dead statements within loop computations. This optimization can accelerate program concrete execution but has different impacts on symbolic execution. The first example is shown in Figure 1a, where the code with marks is the code transformed through the compiler optimization. The transformation removes the first redundant loop as marked, and accordingly simplifies path conditions, which facilitates symbolic execution. As a result, symbolic execution after optimization requires only 11 queries[2], while symbolic execution before optimization requires 54 queries.

Figure 1b presents another example on ADCE. Contradictory to the observation in Figure 1a, the optimization used in Figure 1b decelerates symbolic execution. More specifically, the transformation tries to avoid redundant computations by complicating the starting condition of the first loop (i.e., at Line 3). That is, turning on this compiler optimization increases the complexity of the path conditions, which enhances the difficulty of constraint solving in symbolic execution. As a result, symbolic execution before optimization requires

[2] Query is a concept of SMT constraint solving. More queries tend to decrease the efficiency of symbolic execution.

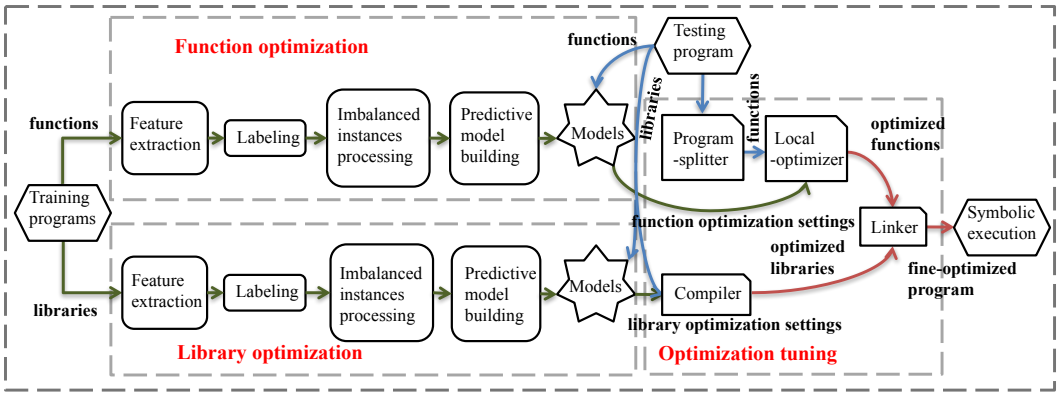


Figure 2 Overview of LEO.

48 queries, while it requires 107 queries after optimization, significantly aggravating the efficiency problem of symbolic execution. Combining the observations from Figures 1a and 1b, a compiler optimization can behave differently, e.g., accelerate or decelerate symbolic execution, making it not proper to give a unified compiler optimization setting for all programs. Therefore, this paper targets learning how to tune these compiler optimizations for each individual program to accelerate symbolic execution via code transformation.

From Figures 1a and 1b, the transformation performed by compiler optimizations actually occurs on some code portions rather than the whole program. For example, the transformation in Figure 1a occurs at Lines 3-6, and the transformation in Figure 1b occurs at Lines 3. That is, the transformation actually occurs at fine granularities (e.g., statements and functions) rather than at coarse granularities (e.g., the whole program). If a compiler optimization is uniformly set at coarse granularities, it is hard to guarantee the efficiency of symbolic execution. For example, if a large program consists of the two functions in Figure 1a and Figure 1b, it is hard to tell whether the optimization, ADCE, accelerates the symbolic execution of the whole program because such an optimization has opposite influence on the two functions. That is, to accelerate symbolic execution, compiler optimizations should be tuned at fine granularities, e.g., the function level, rather than at coarse granularities. On the other side, it is costly to tune compiler optimizations at much finer granularities (e.g., the statement level) due to the extra efforts on compiler optimization tuning. Therefore, *in this paper, we use the function level as a compromise and tune compiler optimizations for symbolic execution at the* ***function*** *level.*

3 Approach

To accelerate symbolic execution via code transformation, we propose the first approach to tuning compiler optimizations at the function level based on machine learning. The key insight of our approach is that program code portions with certain features (e.g., structure or complexity features) or combinations of features are inherently more likely to be transformed to easy-to-analyze programs by certain compiler optimizations. Besides the implemented source functions, a program may use API functions of some libraries, and thus it is necessary to learn how to set compiler optimizations for these libraries as well. However, the libraries are usually so large that splitting libraries into functions and tuning compiler optimizations for each library function incur huge costs, and thus LEO predicts the settings of compiler optimizations for libraries in a way different from what it does for functions. That is, LEO

divides a program into source-code functions and libraries, and predicts their settings of compiler optimizations separately.

Figure 2 presents the overview of LEO. It first trains a predictive model for each optimization to predict whether the compiler optimization should be turned on for a function (see Section 3.1), and then trains a predictive model for each compiler optimization to predict whether the compiler optimization is turned on for libraries (see Section 3.2). Based on the prediction results, LEO tunes the settings of compiler optimizations for the program under test, and implements the program-splitter and local-optimizer components to facilitate compiler optimization settings for each code portion (see Section 3.3). Note that although the general idea of LEO applies to various symbolic execution engines and compilers, in this work, we present LEO based on the KLEE symbolic execution engine [15] and its underlying LLVM compiler infrastructure [55].

3.1 Function Optimization

In function optimization, LEO first collects a set of training instances from source functions by extracting their features and identifying their labels, and then builds a predictive model based on these training data for each compiler optimization.

3.1.1 Feature Extraction

To predict whether a compiler optimization can facilitate symbolic execution for a function, the identified features from source functions should characterize how compiler optimizations influence the efficiency of symbolic execution. Therefore, we identify features from two aspects: path exploration and constraint solving, which are main reasons for the efficiency problem of symbolic execution [18]. From the aspect of path exploration, we use a group of features relevant to *program structure*, e.g., the number of basic blocks with one/two/more than two successor(s), the number of edges in the control flow graph, and the number of conditional branches. From the aspect of constraint solving, we use a group of features relevant to *program complexity*, e.g., the number of references (def/use) of static/extern/local variables, the number of instructions that do pointer arithmetic, and the number of indirect references via pointers. In particular, prior work on compiler optimizations for program concrete execution [38] has already recognized some characteristics of a program that are related to compiler optimizations. Here we use all these characteristics as the features of LEO because these features are relevant to either path exploration or constraint solving. Details about our features can be found in the homepage of LEO.

For each function, LEO extracts the values of these features, which are represented by a vector whose elements are numeric. As these features may have different value ranges, LEO normalizes each element's value into the range [0,1] using the min-max normalization [48] so as to adjust values measured on different scales into a common scale. Supposed that the set of training instances (i.e., functions) is denoted as $P = \{p_1, p_2, \ldots, p_n\}$, the set of vectors extracted from P is denoted as $V = \{v_1, v_2, \ldots, v_n\}$, the set of elements in a vector is denoted as $E = \{e_1, e_2, \ldots, e_m\}$, and the value of the element e_j in the vector v_i before normalization is denoted as x_{ij}, then the value of the element e_j in the vector v_i after normalization is denoted as x^*_{ij}, Formula 1 presents the min-max normalization on x_{ij}, where $1 \leq i \leq n$ and $1 \leq j \leq m$.

$$x^*_{ij} = \frac{x_{ij} - min(\{x_{kj} | 1 \leq k \leq n\})}{max(\{x_{kj} | 1 \leq k \leq n\}) - min(\{x_{kj} | 1 \leq k \leq n\})} \quad (1)$$

3.1.2 Labeling

LEO is designed to build a predictive model for a compiler optimization, characterizing how to accelerate symbolic execution through the compiler optimization setting. Therefore, the label for each training instance is defined as the setting of a compiler optimization that accelerates symbolic execution. In other words, a label of a training instance (i.e., function) refers to whether a compiler optimization should be turned on or off.

For any training instance (i.e., function), LEO labels based on the comparison between its symbolic execution efficiency with the compiler optimization turned on and turned off. Same as existing work [33, 15, 87], symbolic execution efficiency is measured by line coverage achieved by the generated test inputs within time limit. That is, within time limit, if line coverage achieved when turning on this compiler optimization is higher than that when turning off it, the instance label for this compiler optimization is "turning on". Otherwise, the label is "turning off".

It is hard to learn whether a compiler optimization should be turned on or off for a *function*, since symbolic execution takes the whole program rather than each function as input. To relieve this issue, LEO estimates the label of each function by analyzing the line coverage of the whole program instead. More specifically, LEO first collects line coverage of the whole program, i.e., which line of code is covered by the test inputs generated through symbolic execution, then determines the line coverage of each function by analyzing the distribution of line coverage. Finally, for each function, LEO compares its line coverage between symbolic execution with the optimization on and that with the optimization off to set the label.

3.1.3 Imbalanced Instance Processing

Through the steps introduced by Sections 3.1.1 and 3.1.2, we collect a set of training instances with features and labels. Based on the prior work [33], some compiler optimizations help accelerate symbolic execution in most cases but some other compiler optimizations make symbolic execution slower in most cases. That is, for a compiler optimization, its number of training instances whose labels are turned on may be greatly different from its number of training instances whose labels are turned off, which can incur the imbalanced data problem.

As the imbalanced problem may have serious impact on the accuracy of classification [23, 21], LEO uses over-sampling strategy to relieve the impact of imbalanced instances in optimization prediction. Here we choose over-sampling strategy rather than other strategies (e.g., under-sampling) because it is costly to collect a large number of training instances[3]. In particular, LEO uses SMOTE [22], because SMOTE over-samples the minority class by creating synthetic examples rather than by over-sampling with replacement [22]. More specifically, for each instance in the minority class, SMOTE creates synthetic examples along the line segments joining the instance and its k nearest neighbors by regarding all instances as points in space. Based on the amount of over-sampling required, neighbors are randomly chosen from the k nearest neighbors, and then one instance is created on each line segment.

[3] Collecting a training instance requires feature extraction and labeling. Moreover, to label each instance, each program has to be executed twice in symbolic execution, including turning on the compiler optimization and turning off the compiler optimization.

3.1.4 Predictive Model

For each compiler optimization, LEO builds a predictive model through machine learning. In particular, LEO adopts the SMO algorithm, which is used to solve the quadratic programming problem in the training of Support Vector Machines (abbreviated as SVM) [69] and regarded as the fastest for linear SVM and sparse data sets. Note that although LEO is implemented based on SMO, it is not specific to this machine learning algorithm and we investigate the impact of machine learning algorithms in Section 4.6.4.

3.2 Library Optimization

As the libraries are usually so large that splitting libraries into functions and tuning compiler optimizations for each library function incur huge cost. Therefore, we predict compiler optimizations for libraries in a different way. More specifically, LEO regards the functions of libraries as a whole by building a predictive model for libraries used in a program (rather than each function). Similar to function optimization prediction, LEO predicts compiler optimizations for libraries as follows.

First, LEO defines a set of new features that characterize how compiler optimizations accelerate symbolic execution for libraries. Since library functions are relatively fixed in implementation and repeatedly used by various client code, it is not necessary to collect detailed features about each library function separately. Instead, knowing how compiler optimizations impact programs that used a library function before, can help predict how compiler optimizations impact the current program using that library function. Therefore, LEO directly uses whether each individual library function is called by a program as features of library optimization. That is, for each training instance (i.e., a program), LEO identifies the called library functions and uses 1/0 to represent a library function is/isn't called. As the values of these features are all 0 or 1, normalization is not necessary.

Second, LEO labels each training instance. An instance label is whether a compiler optimization should be turned on or off for the libraries used in a program. Similar to the process of function optimization prediction, LEO determines a label by comparing the line coverage of the whole program achieved when turning on the compiler optimization and that achieved when turning off the compiler optimization within time limit[4].

Finally, based on the collected training data, LEO builds a predictive model for each compiler optimization using also SMO. Note that LEO also uses SMOTE to filter the impact of the imbalanced problem in library optimization prediction.

3.3 Optimization Tuning

Following Sections 3.1 and 3.2, LEO learns the settings of all compiler optimizations for a program, including each source-code function and the related libraries. However, as symbolic execution engines do not support various settings on different source-code functions of a program, LEO provides such fine-granularity optimization tuning by implementing two components: program-splitter and local-optimizer.

LEO first adopts the learnt settings of compiler optimizations for libraries by compiling the libraries individually. Then LEO splits the whole program into multiple files, each of which is only a function of the program, and adopts the learnt settings of compiler

[4] As the features of a training instance are directly related to libraries and optimized libraries also contribute to the line coverage of the program, LEO approximately uses whether the line coverage of the program is improved when turning on a compiler optimization as the label for library optimization prediction, to save the cost of labeling.

optimizations for each file (i.e., function). Finally, LEO integrates the multiple optimized files and optimized libraries into a fine-optimized program using the LLVM linker, and analyzes this program rather than the original target program through symbolic execution. Due to program complexity, implementing the program-splitter and local-optimizer is an important technical challenge in LEO. In the following subsections, we first present the details on how to split a program into multiple function-level files in Section 3.3.1, and then present the details on how to optimize these files using learnt settings and integrate these optimized files and libraries in Section 3.3.2.

3.3.1 Program-Splitter

For any given program denoted as P_A, LEO splits it into function-level files $P_B = \{p_{b1}, p_{b2}, \ldots, p_{bn}\}$, where p_{bk} refers to a function-level file ($1 \leq k \leq n$ and n is the total number of functions in P_A), via two stages: preprocessing stage and splitting stage. In the preprocessing stage, our approach preprocesses P_A and prepares the necessary materials, and in the splitting stage our approach splits P_A into function-level files based on these materials.

In the preprocessing stage, LEO first expands macro and removes comments to expediently transform P_A to P_B, and then prepares the following materials for the splitting stage:

- A common-symbol table, which contains the symbols of all common variables and functions in P_B[5], so as to solve the duplicate-name issue in link-time.
- A type-definition table, which records all definitions of existing types (e.g., structs) in P_B.
- A dependent table for each function in P_B, which records the declarations of its dependent functions and global variables.

In the splitting stage, our approach generates an individual file (denoted as p_{bk}) for each function (denoted as M_k) in P_B by the following steps:

- Putting the declarations of dependent functions and global variables into p_{bk} and using "extern" as their modifier, based on the dependent table of this function;
- Modifying the scope of the dependent functions and global variables, i.e., removing the "static" modifier, so that they can be used by the other files;
- Putting M_k into the file p_{bk};
- Putting all needed type definitions into p_{bk} referring to the type-definition table, based on all declarations in p_{bk}.

In particular, our approach records all global variables in an individual file so that all other files can use them.

3.3.2 Local-Optimizer

The optimizer of the KLEE symbolic execution engine applies all compiler optimizations together, but does not allow to turn on one compiler optimization or a subset of compiler optimizations. Therefore, we implement a local-optimizer by setting an interface that appoints which compiler optimizations are turned on. That is, we regard the names of compiler optimizations as parameters that are passed to the optimizer by the interface. Finally, LEO integrates all optimized files and libraries into a fine-optimized program using

[5] In this paper, common variables and functions refer to the global variables and functions without "static" modifier.

the LLVM linker. When linking libraries, some symbols may have duplicate names, which will incur link errors. To solve this problem, our approach utilizes the common-symbol table generated in the program-splitter to remove the symbols whose scope is the current file from the symbol table of the executable.

4 Experimental Study

Our study addresses the following four research questions:

- **RQ1:** How does LEO perform on accelerating symbolic execution via code transformation?
- **RQ2:** How do different training and testing time limits impact LEO?
- **RQ3:** Do both function optimization and library optimization contribute to LEO?
- **RQ4:** How do different machine learning algorithms impact LEO?

Note that in our study, there are two types of time limits: training time limit and testing time limit. The former refers to the symbolic-execution time used for collecting training instances, and the latter refers to the symbolic-execution time used to analyze the programs under test.

4.1 Tools and Libraries

In our study, we use KLEE [15], one of the most widely used symbolic execution engines [15, 33, 87, 57]. KLEE is implemented using C++ based on the LLVM infrastructure, whose compiler provides dozens of compiler optimizations. The same as prior work [87, 33], we build KLEE with LLVM 2.9, which has 30 compiler optimizations integrated by KLEE. These optimizations are turned on through the command "–optimize" and turned off through the command "–disable-opt", which are two default configurations of KLEE. In our study, we use similar KLEE options as the prior work [15]. Following the prior work [33, 30], we use the DFS search heuristic in KLEE so as to acquire more deterministic results, and disable the caching of KLEE since the caching contents can be different for different strategies, making it hard to check the actual impacts of different strategies. More discussion on the impact of search heuristics and caching can be found in Section 5.2.

We implement LEO's machine-learning component by using the SMO algorithm provided by Weka 3.6.12[6], whose Puk kernel is set with $omega = 3$ and $sigma = 1$ in this study based on a preliminary study on a small dataset.

In our study, we measure the performance with code coverage and fault detection rates achieved by the test inputs generated within the given testing time limit. For code coverage, we use line coverage, which is widely used to measure the effectiveness of symbolic execution [33, 15, 87, 86]. For fault detection rates, since real faults are usually small in number in practice and mutation faults have been widely recognized as suitable for simulating real faults in software testing experimentation [5, 51, 26, 25], following prior work [86, 57], we use mutation testing to simulate real faults and check the mutation scores, i.e., the proportion of killed mutants in all generated mutants. When collecting code coverage and mutation scores, we use widely-used and mature tools gcov[7] and mutGen [5]. When calculating mutation scores, following prior work [86, 57], we regard the console outputs of the original program as test oracles. If there is any difference between the console outputs of the original program and the console outputs of a mutant for the same test inputs, we regard a mutant as killed.

[6] `http://www.cs.waikato.ac.nz/ml/weka/`.
[7] `http://ltp.sourceforge.net/coverage/gcov.php`

4.2 Subjects

Following previous work on symbolic execution [15, 33, 87, 57, 86, 62], we also use GNU Coreutils C programs as subjects, which implement different tools for Unix-like operating systems [15, 57]. In particular, we use 76 GNU Coreutils 6.11 programs[8], whose total lines of source code (SLOC)[9] are 39,752 linked with an internal library size of 49,710 SLOC and an external library size of 223,147 SLOC.

Furthermore, when measuring mutation scores, we use 40 programs in GNU Coreutils because the rest of programs cannot produce outputs under the study environment or their outputs are related to environment/context information (e.g., system time). Following prior work [86], for each program, we randomly select 100 mutants. If the total number of generated mutants is less than 100, we use all generated mutants instead.

4.3 Experimental Setup

We consider the following independent variables:

Compared Approaches. LEO is the first automated approach to tuning compiler optimizations for accelerating symbolic execution. Therefore, we compare LEO with only the default optimization configurations of KLEE, i.e., all compiler optimizations off (abbreviated as NO) and all compiler optimizations on (abbreviated as ALL). Here, NO is regarded as the baseline, representing the original symbolic execution without any code transformations for programs under test whereas ALL is regarded as a compared approach applying all available compiler optimizations to accelerate symbolic execution.

Time limits. As the GNU Coreutils programs are normally large and complicated, all paths of a program cannot be fully explored by symbolic execution during the acceptable period of time. Therefore, similar as prior work [15, 33], in the experiment we also limit the maximum execution time of KLEE and halt its execution when reaching the time limit. In particular, for the testing time limit, we set it to 10, 15, 20, 25, and 30 minutes, to investigate whether LEO always performs well regardless of testing time limits. We set the default training time limit to be 10 minutes in LEO. Moreover, we also study the impact of different training time limits on LEO. Due to the high cost of training, we first set the training time limit to be 10 minutes to 30 minutes with the step of 10 minutes. Then, we also add a 5-minute training time limit to better understand the trend of the impact of training time limit. That said, we set the training time limit to be 5, 10, 20, and 30 minutes.

Variants of LEO. To explore whether each component of LEO (i.e., function optimization and library optimization) contributes to LEO on accelerating symbolic execution, we adapt LEO by removing each component and compare the performance of the adapted LEO and the original LEO. In particular, LEO has four variants through such adaption, which are (1) LEO with all compiler optimizations for libraries turned on (denoted as LEO-Lall), (2) LEO with all compiler optimizations for libraries turned off (LEO-Lno), (3) LEO with all compiler optimizations for functions turned on (LEO-Fall), and (4) LEO with all compiler optimizations for functions turned off (LEO-Fno). That is, LEO-Lall and LEO-Lno are variants of LEO without library optimization prediction, LEO-Fall and LEO-Fno are variants of LEO without function optimization prediction.

[8] We remove some programs from GNU Coreutils mainly because they can destroy our experimental data by generating dangerous test inputs.

[9] Following prior work [87, 61, 54, 60], the SLOC in this paper are measured by cloc, which is accessible at `https://github.com/AlDanial/cloc`.

Machine learning algorithms. To investigate the impact of machine learning algorithms on LEO, we consider other five typical machine learning algorithms besides SMO – Alternating Decision Tree (abbreviated as ADT) [36], Bayesian Logistic Regression (BLR) [40], Multinomial Logistic Regression (MLR) [56], LogitBoost (LB) [37], and Random Forests (RF) [12]. In particular, we also use their implementations provided by Weka.

Following prior work on machine learning [20, 6], LEO is evaluated through leave-one-out cross-validation. That is, for each subject, we use the instances collected from the *remaining* 75 programs as the training data to build predictive models for compiler optimizations respectively, and use these predictive models to learn the settings of compiler optimizations for the specific subject. The training is conducted offline, and not included as overhead. Note that we use 68 of 76 programs as the testing programs in turn because the other 8 programs incur KLEE errors when using some predicted compiler optimization settings.

The dependent variables considered are line coverage and mutation scores, which have been widely used in prior studies on symbolic execution [57, 86, 33, 15, 87].

4.4 Verifiability

The experimental study is conducted on a workstation with eight-core Intel Xeon E5620 CPU (2.4GHz) with 24G memory, and Ubuntu 15.04 operating system. For ease of experiment replication, we release the tools and implementation used in our experiment as well as all the experimental data at the homepage of LEO[10]. The detailed results in the homepage allow for verification without running the experiment again. The open-source tools, the implementation of our experiment (including the source code and readme files), and the subjects and mutants are available, so that one can easily reproduce our experiment.

4.5 Threats to Validity

The threats to internal validity mainly lie in the tool supports and our own implementations. To reduce the threat from tool supports, we use the widely-used KLEE symbolic execution engine [15] and the LLVM compiler infrastructure [55]. Since LEO predicts compiler optimizations for each source-code function, it may discount the effect of inter-procedural compiler optimizations. In the future, we plan to utilize LEO to predict optimizations for *a subset of* functions rather than *a single* function to reduce this threat. It can also bring an additional benefit, i.e., reducing the cost of LEO for optimization prediction. Also, we use the mature tools, i.e., gcov and mutGen [5], to collect line coverage and generate mutants, respectively. To avoid implementation errors, the first two authors review the source code and experimental scripts, and we adopt the mature implementations of those machine learning algorithms used in our study, which are provided by Weka.

The threat to external validity mainly lies in the studied subjects. Although we use the widely-used GNU Coreutils programs [15, 33, 87, 57], they may not be representative of other programs. To reduce this threat, we will use more and larger subjects in the future. Note that our current subjects do not suffer from overfitting. The reason is that GNU coreutils was created by merging a lot of earlier GNU packages; even within the same package, programs differ in their implementation, creation time, and functionalities. Moreover, LEO optimizes at the function rather than program level. Regarding to the libraries, LEO predicts

[10] `https://github.com/JunjieChen/leo`.

optimizations of a library for a target program based on the actual library portions invoked and does not necessarily produce the same prediction results on the same libraries of different target programs, which is confirmed by our experimental data. Therefore, the library code does not have overfitting concerns as well.

The threats to construct validity lie in the measurement, the time limits, and the compared approaches. In this study, we measure the performance of LEO through only its acceleration effectiveness rather than its cost because LEO has little overhead[11]. In particular, we choose the mostly used line coverage and mutation scores. The second threat comes from the time limit, including the training and testing time limits. To reduce these threats, we will repeat the experiment by using other time limits. The third threat lies in the compared approaches. As the first work on optimization prediction for symbolic execution, we use the state-of-the-art symbolic execution work KLEE and its compiler optimization support as the compared approaches (i.e., ALL and NO). There are also some other approaches that may be compared in the study, such as statically applying a subset of compiler optimizations for all the programs [84, 33]. However, according to the existing work [33], there is no unified compiler-optimization configuration guaranteeing the efficiency of symbolic execution for all programs. In particular, the experimental results in the existing work [33] have shown that the four different subsets of KLEE compiler optimizations that are designed based on their knowledge for symbolic execution perform almost the same as turning on all the optimizations (i.e., ALL). Therefore, statically applying a subset of compiler optimizations may not outperform LEO.

4.6 Results and Analysis

4.6.1 RQ1: Performance Comparison

Performance on line coverage. Table 1 lists the line coverage achieved by LEO and ALL/NO under the 10-minute testing time limit[12], where (✔), (❍), (✘) represent that the approach achieves the highest, medium, lowest line coverage on the corresponding subject among LEO, ALL and NO, respectively. In particular, the last two rows of this table present the number of subjects where each approach achieves the best, medium, and worst results. From this table, the number of subjects where LEO achieves the best performance (i.e., 50) is much larger than that of ALL (i.e., 26) and NO (i.e., 13), and the number of subjects where LEO achieves the worst performance (i.e., 5) is much smaller than that of ALL (i.e., 13) and NO (i.e., 44). Based on these results, LEO is more effective than the baseline NO, demonstrating code transformation is indeed a promising direction to accelerate symbolic execution. Also, LEO is more effective than ALL, indicating that effectively tuning compiler optimizations is a good exploration in this direction and our machine-learning based approach indeed predicts better compiler-optimization settings specific to symbolic execution.

Figure 3 further shows the comparison results between LEO and ALL/NO under the 10-minute testing time limit, where we calculate the difference between the line coverage achieved by LEO and that achieved by ALL/NO, as the coverage improvement using LEO. In this figure, the y axis represents the coverage improvement using LEO and the x axis sorts the

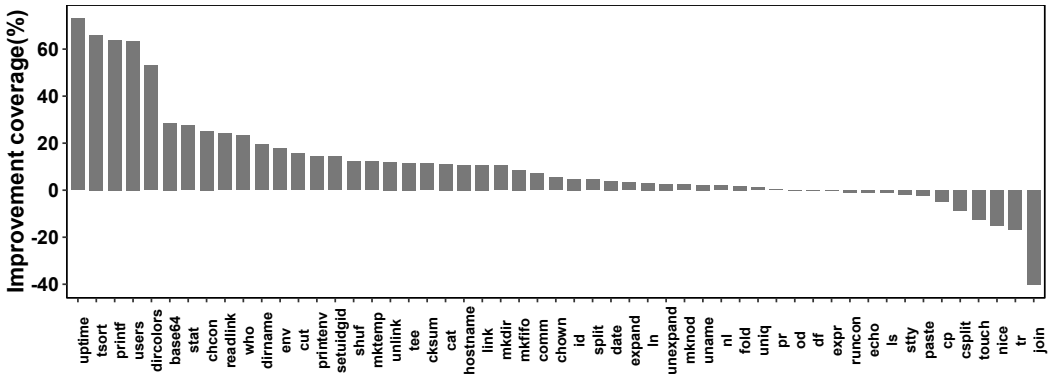
[11] Predictive models are built offline and they predict each optimization for each function or libraries very quickly (in seconds).

[12] If no otherwise specified, all training symbolic execution runs of LEO use the default 10-minute training time limit in the remainder of this paper.

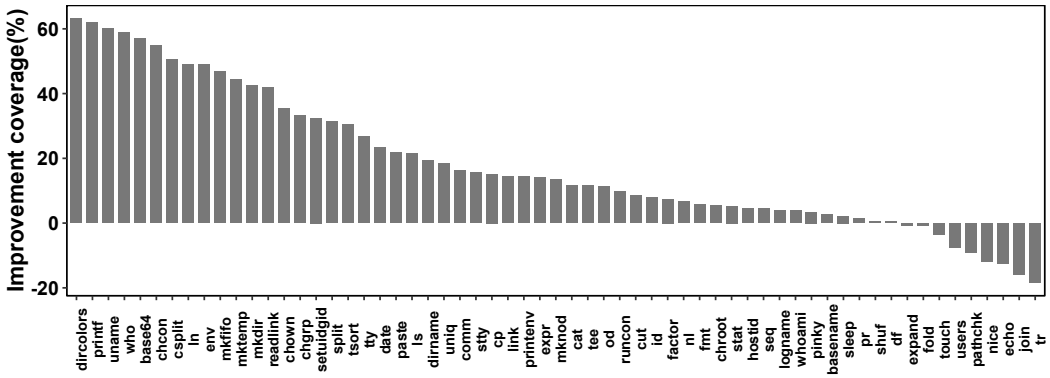
Table 1 Line coverage achieved by LEO/ALL/NO within 10-minute testing time limit

Sub	LEO	ALL	NO	Sub	LEO	ALL	NO
base64	71.43(✔)	42.86(❍)	14.29(✘)	basename	79.49(✔)	79.49(✔)	76.92(✘)
cat	66.38(✔)	55.17(❍)	54.74(✘)	chcon	70.77(✔)	45.64(❍)	15.90(✘)
chgrp	67.78(✔)	67.78(✔)	34.44(✘)	chown	65.59(✔)	60.22(❍)	30.11(✘)
chroot	62.16(✔)	62.16(✔)	56.76(✘)	cksum	91.94(✔)	80.65(✘)	91.94(✔)
comm	78.57(✔)	71.43(❍)	62.24(✘)	cp	41.46(❍)	46.34(✔)	26.29(✘)
csplit	53.76(❍)	62.57(✔)	3.30(✘)	cut	64.53(✔)	48.99(✘)	56.08(❍)
date	48.05(✔)	44.16(❍)	24.68(✘)	df	64.15(❍)	64.42(✔)	63.61(✘)
dircolors	73.16(✔)	20.00(❍)	10.00(✘)	dirname	93.55(✔)	74.19(❍)	74.19(❍)
echo	27.18(✘)	28.16(❍)	39.81(✔)	env	100.00(✔)	82.22(❍)	51.11(✘)
expand	42.38(❍)	39.07(✘)	43.05(✔)	expr	48.22(❍)	48.52(✔)	34.02(✘)
factor	71.64(✔)	71.64(✔)	64.18(✘)	fmt	65.83(✔)	65.83(✔)	60.19(✘)
fold	43.36(❍)	41.59(✘)	44.25(✔)	hostid	63.64(✔)	63.64(✔)	59.09(✘)
hostname	67.86(✔)	57.14(✘)	67.86(✔)	id	32.03(✔)	27.34(❍)	24.22(✘)
join	12.93(✘)	53.06(✔)	28.80(❍)	link	75.00(✔)	64.29(❍)	60.71(✘)
ln	78.35(✔)	75.26(❍)	29.38(✘)	logname	56.00(✔)	56.00(✔)	52.00(✘)
ls	44.24(❍)	45.33(✔)	22.70(✘)	mkdir	77.27(✔)	66.67(❍)	34.85(✘)
mkfifo	82.98(✔)	74.47(❍)	36.17(✘)	mknod	56.10(✔)	53.66(❍)	42.68(✘)
mktemp	88.89(✔)	76.77(❍)	44.44(✘)	nice	61.02(✘)	76.27(✔)	72.88(❍)
nl	48.82(✔)	46.92(❍)	42.18(✘)	nohup	77.63(✔)	77.63(✔)	77.63(✔)
od	40.65(❍)	40.79(✔)	29.25(✘)	paste	66.84(❍)	68.98(✔)	44.92(✘)
pathchk	46.97(❍)	46.97(❍)	56.06(✔)	pinky	83.33(✔)	83.33(✔)	79.91(✘)
pr	38.08(✔)	37.86(❍)	36.64(✘)	printenv	77.14(✔)	62.86(❍)	62.86(❍)
printf	74.32(✔)	10.51(✘)	12.45(❍)	pwd	20.34(✔)	20.34(✔)	20.34(✔)
readlink	96.00(✔)	72.00(❍)	54.00(✘)	runcon	54.37(❍)	55.34(✔)	44.66(✘)
seq	53.04(✔)	53.04(✔)	48.62(✘)	setuidgid	55.84(✔)	41.56(❍)	23.38(✘)
shuf	59.88(✔)	47.67(✘)	59.30(❍)	sleep	45.65(✔)	45.65(✔)	43.48(✘)
split	45.62(✔)	41.01(❍)	14.29(✘)	stat	37.05(✔)	9.47(✘)	31.75(❍)
stty	29.43(❍)	31.32(✔)	13.77(✘)	tee	86.96(✔)	75.36(❍)	75.36(❍)
touch	56.25(✘)	68.75(✔)	59.72(❍)	tr	22.15(✘)	39.15(❍)	40.52(✔)
tsort	72.91(✔)	6.90(✘)	42.36(❍)	tty	76.67(✔)	76.67(✔)	50.00(✘)
uname	79.55(✔)	77.27(❍)	19.32(✘)	unexpand	47.42(✔)	44.85(✘)	47.42(✔)
uniq	64.32(✔)	63.24(❍)	45.95(✘)	unlink	72.00(✔)	60.00(✘)	72.00(✔)
uptime	91.03(✔)	17.95(✘)	91.03(✔)	users	90.38(❍)	26.92(✘)	98.08(✔)
who	83.09(✔)	59.71(❍)	24.10(✘)	whoami	53.85(✔)	53.85(✔)	50.00(✘)
Best(✔)	50	26	13	**Med**(❍)	13	29	11
Worst(✘)	5	13	44	–	–	–	–

subjects in their coverage improvement by removing those with zero coverage improvement. That is, any bar above 0 represents a subject whose LEO result is better than ALL or NO, whereas any bar below 0 represents a subject whose LEO result is worse. Besides, LEO achieves the same line coverage as ALL in 14 subjects, and as NO in 6 subjects. From this figure, the vast majority of bars are above 0. That is, LEO makes symbolic execution more efficient than both ALL and NO in most cases. Moreover, the improvement of LEO is usually larger than its decrement. In particular, the increased coverage for *uptime*, *tsort*, *printf* and *users* on ALL, as well as *dircolors*, *printf* and *uname* on NO are even more than 60%.



(a) LEO-ALL



(b) LEO-NO

Figure 3 Coverage improvement within 10-minute testing time limit

This is another empirical evidence that LEO does effectively accelerate symbolic execution on most subjects.

To further quantitatively measure the performance of LEO on accelerating symbolic execution, similar to previous work [24], we calculate the improvement rate of LEO in terms of line coverage for each program. It is calculated via Formula 2, where $Cov(LEO)$ represents the line coverage achieved by LEO and $Cov(ALL(orNO))$ represents the line coverage achieved by ALL or NO. The average line-coverage improvement rate of LEO compared with ALL on all subjects is *46.48%* and that compared with NO is *88.92%*, demonstrating the significant acceleration performance of LEO in terms of line coverage.

$$Rate_{Cov} = \frac{Cov(LEO) - Cov(ALL(orNO))}{Cov(ALL(orNO))} * 100\% \quad (2)$$

Note that in some cases LEO decelerates symbolic execution, e.g., *nice* and *tr*. We try to analyze the possible reasons and find that some compiler optimizations have coupling effect in fact. For instance, based on the comments of LLVM, the optimization "IndvarSimplify"[13] should be performed after all the desired loop optimizations (e.g., the optimization "LoopRotation"). Currently, LEO learns each predictive model for each compiler optimization

[13] This optimization analyzes and transforms the induction variables into simpler forms suitable for subsequent analysis and transformation.

Table 2 Mutation scores achieved by LEO/ALL/NO within 10-minute testing time limit.

Sub	LEO	ALL	NO	Sub	LEO	ALL	NO
base64	17.00 (✔)	11.00 (❍)	4.00(✘)	basename	44.00 (✔)	44.00 (✔)	44.00(✔)
chcon	39.00 (✔)	18.00 (❍)	7.00(✘)	cksum	10.00 (❍)	9.00 (✘)	14.00(✔)
comm	20.00 (❍)	14.00 (✘)	21.00(✔)	cut	19.00 (❍)	18.00 (✘)	25.00(✔)
dircolors	46.00 (✔)	12.00 (✘)	45.00(❍)	dirname	74.12 (❍)	98.82 (✔)	74.12(❍)
env	100.00 (✔)	62.20 (❍)	48.78(✘)	expand	3.00 (❍)	3.00 (❍)	13.00(✔)
expr	100.00 (✔)	99.00 (❍)	8.00(✘)	fold	3.00 (❍)	3.00 (❍)	9.00(✔)
hostid	60.87 (✔)	60.87 (✔)	34.78(✘)	link	37.00 (✘)	46.00 (✔)	39.00(❍)
ln	99.00 (✔)	42.00 (❍)	11.00(✘)	logname	62.50 (✔)	62.50 (✔)	32.50(✘)
mkfifo	51.28 (❍)	50.00 (✘)	100.00(✔)	mknod	50.00 (✔)	46.00 (❍)	26.00(✘)
nice	29.00 (✘)	48.00 (✔)	40.00(❍)	nl	0.00 (❍)	0.00 (❍)	7.00(✔)
nohup	39.00 (✔)	39.00 (✔)	39.00(✔)	od	25.00 (❍)	28.00 (✔)	3.00(✘)
paste	11.00 (❍)	10.00 (✘)	23.00(✔)	pathchk	20.00 (❍)	18.00 (✘)	24.00(✔)
printf	25.00 (✔)	4.00 (❍)	1.00(✘)	pwd	7.00 (✔)	7.00 (✔)	7.00(✔)
readlink	66.67 (✔)	38.10 (✘)	47.62(❍)	runcon	32.00 (✔)	27.00 (❍)	25.00(✘)
setuidgid	27.00 (✔)	20.00 (❍)	13.00(✘)	sleep	32.00 (✔)	21.00 (✘)	30.00(❍)
split	12.00 (❍)	16.00 (✔)	11.00(✘)	tee	31.00 (✔)	20.00 (✘)	29.00(❍)
touch	23.00 (✘)	27.00 (✔)	25.00(❍)	tr	4.00 (✘)	12.00 (❍)	15.00(✔)
tsort	4.00 (✘)	9.00 (✔)	7.00(❍)	tty	46.30 (✔)	44.44 (❍)	24.07(✘)
unexpand	3.00 (❍)	3.00 (❍)	12.00(✔)	unlink	98.61 (✔)	58.33 (✘)	98.61(✔)
users	100.00 (✔)	100.00 (✔)	100.00(✔)	whoami	69.70 (✔)	69.70 (✔)	36.36(✘)
Best(✔)	22	14	16	**Med**(❍)	13	15	9
Worst(✘)	5	11	15	–	–	–	–

individually. Neglect of such couple effects may impact the performance of LEO. Therefore, in the future we plan to improve LEO by learning predictive models considering the coupling effect of compiler optimizations, which can be learned/inferred through source code and documentation of these optimizations.

Performance on mutation score. Besides line coverage, Table 2 further shows the comparison of mutation scores. Similar with Table 1, in this table, (✔), (❍), (✘) represent the approach achieves the highest, medium, lowest mutation scores. From this table, similarly, the number of subjects where LEO achieves the best mutation scores (i.e., 22) is larger than that of ALL (i.e., 14) and NO (i.e., 16), and the number of subjects where LEO achieves the worst mutation scores (i.e., 5) is smaller than that of ALL (i.e., 11) and NO (i.e., 15). This finding further confirms the performance of LEO in enhancing symbolic execution.

Similarly, to further quantitatively measure its performance, we also calculate the improvement rate of LEO in terms of mutation score for each program. It is calculated via the similar formula – Formula 3, where $Mut(LEO)$ represents the mutation score achieved by LEO and $Mut(ALL(orNO))$ represents the mutation score achieved by ALL or NO. The average mutation-score improvement rate of LEO compared with ALL on all subjects is *33.88%* and that compared with NO is *149.11%*, further demonstrating the significant acceleration performance of LEO in terms of mutation score.

$$Rate_{Mut} = \frac{Mut(LEO) - Mut(ALL(orNO))}{Mut(ALL(orNO))} * 100\% \quad (3)$$

Table 3 Comparison within various testing time limits under the default training time limit.

Time (minutes)	#Best			#Worst		
	LEO	ALL	NO	LEO	ALL	NO
10	50	26	11	5	13	44
15	51	29	10	5	12	46
20	46	33	8	5	10	47
25	44	38	8	5	9	45
30	50	34	7	6	9	45

Table 4 Statistics analysis on LEO and ALL/NO within various testing time limits under the default training time limit (α=0.05).

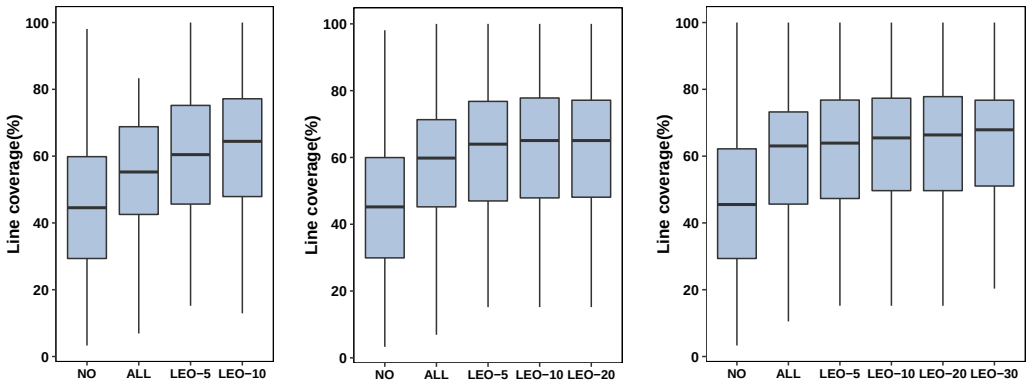
Time (minutes)		10	15	20	25	30
LEO v.s. ALL	**Imp. rate(%)**	46.48	39.01	37.80	23.85	18.39
	p-value	0.000(∗)	0.001(∗)	0.008(∗)	0.065	0.029(∗)
LEO v.s. NO	**Imp. rate(%)**	88.92	86.70	87.26	79.47	89.60
	p-value	0.000(∗)	0.000(∗)	0.000(∗)	0.000(∗)	0.000(∗)

4.6.2 RQ2: Impact of Training and Testing Time Limits

Although label collection in LEO needs to fix time limit, in practical usage symbolic execution may set various time limits (i.e., testing time limit) based on different requirements. It is quite necessary to investigate whether LEO always works no matter which testing time limit is set. Therefore, we first explore the performance of LEO through symbolic execution in different testing time limits by using the predictive models learnt in default 10-minute training time limit, whose results are shown in Table 3. In this table, the first column lists various testing time limits and Columns 2-4 and 5-7 represent the number of subjects where the corresponding approach achieves the best and worst performance, respectively. From Table 3, the number of subjects where LEO achieves best performance is always much larger than that of ALL and NO, and the number of subjects where LEO achieves worst performance is always much smaller than that of ALL and NO. That is, LEO accelerates symbolic execution regardless of testing time limits.

To learn whether LEO outperforms ALL and NO significantly at various testing time limits, we further perform statistical analysis on their results. First, we analyze the population on line coverage achieved by each approach and find that the population of each approach follows the normal distribution by Kolmogorov-Smirnov test [63], which is the precondition of the paired sample T test. Then, we perform a paired sample T test (whose significant level α is 0.05), and the results are shown in Rows 3 and 5 of Table 4, where "*" demonstrates significant difference between the compared approaches. Moreover, in Table 4, Rows 2 and 4 refer to the average line-average improvement rates on all subjects. From this table, LEO significantly outperforms ALL in 4/5 testing time limit comparisons with line-coverage improvement rates ranging from 18.39% to 46.48%, and always significantly outperforms NO with line-coverage improvement rates ranging from 79.47% to 89.60%.

Besides, from Tables 3 and 4, the smaller the gap between the default training time limit and the used testing time limit, the better LEO tends to perform compared with ALL and NO (especially ALL). This observation is as expected due to two possible reasons. First, given sufficient time, symbolic execution can always achieve high line coverage for a subject.



(a) 10min testing **(b)** 20min testing **(c)** 30min testing

Figure 4 Trend of performance of LEO with different training time limits.

30 minutes may be already long enough for symbolic execution of individual GNU Coreutils programs, especially the transformed programs using compiler optimizations. Therefore, symbolic execution will achieve similar (high) line coverage and become saturate eventually, no matter what the settings of compiler optimization are (especially for LEO and ALL). Second, based on the theory of machine learning [82], the time limit in training and testing staying consistent tends to achieve the best effectiveness. In our study, even though the training time limit of LEO is inconsistent with the testing time limit, LEO still improves the efficiency of symbolic execution. That said, if LEO sets training time limit longer than 10 minutes, its accelerating effectiveness may be more obvious when being applied to symbolic execution with these longer testing time limits.

Therefore, we further explore the impact of different training time limits on LEO. More specifically, we study whether the performance of LEO within these longer testing time limits becomes better when using longer training time limits. Figure 4 shows the performance trends of LEO whose training time limit is gradually close to the testing time limit, where the line in each box represents the median line coverage and LEO−X (i.e., X = 5, 10, 20, and 30) refers to LEO with X-minute *training* time limit. Figures 4a, 4b, and 4c present the trends in 10-minute, 20-minute, and 30-minute *testing* time limit, respectively. From each subfigure in Figure 4, with the training time limit being closing to the specific testing time limit, LEO indeed achieves better performance, confirming our hypothesis. Also, we find that LEO always performs better than ALL and NO no matter which training time limit is used.

Furthermore, Table 5 further shows more details about the impacts of training time limits. The comparison results include the number of subjects where LEO with various training time limits achieve best and worst performance, and the average line-coverage improvement rates compared with ALL and NO. For each row in this table, reading values from left to right, we find that with the training time limit being close to the specific testing time limit, the number of subjects where LEO achieve best performance becomes larger, the number of subjects where LEO achieves worst performance becomes smaller, and the average line-coverage improvement rates of LEO mostly becomes better. That is, when the training time limit in LEO is close to the testing time limit, LEO achieves better performance on accelerating symbolic execution.

Overall, LEO mostly significantly accelerates symbolic execution in various testing time limits with our default training time limit. Furthermore, when having more sufficient training time, LEO tends to perform better for longer testing time limits.

Table 5 Comparison within various training time limits.

Training time (minutes)		5	10	20	30
Testing-10min	#Best	46	50	—	—
	#Worst	6	5	—	—
	Imp. rate(%) (v.s. ALL)	39.31	46.48	—	—
	Imp. rate(%) (v.s. NO)	79.18	88.92	—	—
Testing-20min	#Best	45	46	47	—
	#Worst	6	5	5	—
	Imp. rate(%) (v.s. ALL)	32.04	37.80	38.23	—
	Imp. rate(%) (v.s. NO)	81.66	87.26	87.92	—
Testing-30min	#Best	47	50	51	51
	#Worst	8	6	6	4
	Imp. rate(%) (v.s. ALL)	15.98	18.39	18.51	18.51
	Imp. rate(%) (v.s. NO)	83.51	89.60	89.81	89.69

Table 6 Comparison between LEO and its variants.

Approach	LEO v.s.			
	LEO-Lall	LEO-Lno	LEO-Fall	LEO-Fno
#Win	36	50	7	7
#Lose	17	11	0	2
Avg. Improvement (%)	9.89	16.51	3.89	1.28

4.6.3 RQ3: Contribution of Function/Library Optimization

Table 6 shows the comparison between LEO and its four variants. In this table, the second/third row presents the number of subjects where LEO achieves higher/lower line coverage than its variants within 10-minute testing time limit. The last row presents the average coverage improvement through LEO compared with its variants. Note that we do not list the number of subjects that the two compared approaches perform equally, i.e., achieve the same line coverage. From this table, LEO performs much better than its four variants since the number of subjects LEO performing better is always larger than the number of subjects it performing worse. Therefore, both function optimization and library optimization are indispensable.

Furthermore, the difference between "#Win" and "#Lose" in the second and third columns (i.e., results of LEO without library optimization prediction) is much larger than the following two columns (i.e., results of LEO without function optimization prediction). Moreover, the average coverage improvement of LEO compared with LEO without library optimization prediction (i.e., LEO-Lall and LEO-Lno) is also larger than that of LEO compared with LEO without function optimization prediction (i.e., LEO-Fall and LEO-Fno). That is, library optimization is more important than function optimization for LEO in accelerating symbolic execution. The main reason is that the studied GNU Coreutils programs usually use a large portion of library code (Section 4.2).

These results tell us another promising direction for further improving LEO. We have known that library optimization is more important for LEO. Currently, LEO achieves such great performance through simply taking all the libraries used in the program as a whole by predicting unified settings on compiler optimizations for them. If library optimization can be

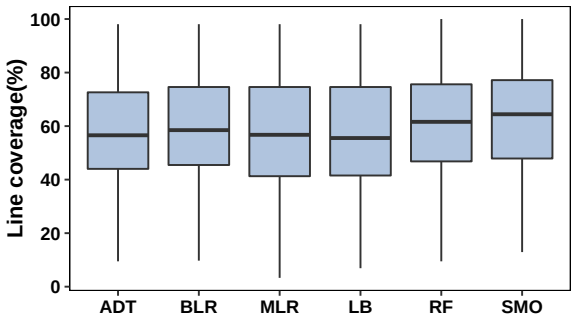


Figure 5 Comparison on machine learning algorithms.

better dealt with, LEO are quite likely to be improved on accelerating symbolic execution. In the future, we plan to learn how to further fine-tune compiler optimizations for libraries, e.g., splitting libraries into several sets of functions (which is more coarse granularity than each single function but makes it more efficient), or analyzing open-source repositories such as Github. Since libraries are widely used in software development, we believe that, like existing library researches (e.g., building a summary for library code to accelerating the analysis of client code [80]), researches on better transforming libraries for accelerating symbolic execution are also worthy and should attract more attentions.

4.6.4 RQ4: Impact of Machine Learning Algorithms

Figure 5 shows the results of LEO with various machine learning algorithms, to investigate the impact of machine learning algorithms on LEO. Actually, our approach using any of machine learning algorithms outperforms ALL and NO. From this figure, our approach using SMO performs slightly better than our approach using other machine learning algorithms, e.g., the top, median and bottom of SMO in box-plot are all higher than those of all other algorithms. Therefore, LEO achieves stably good acceleration performance for all studied machine learning algorithms, and SMO is a better choice.

5 Discussion

In this section, we first discuss the new direction that LEO opens for symbolic execution acceleration, and then discuss the impact of search heuristics and caching on LEO.

5.1 Promising Direction

Our work demonstrates the significant performance of code transformation on accelerating symbolic execution, indicating a promising direction for accelerating symbolic execution. Moreover, learning to tune existing compiler optimizations is a good exploration to accelerate symbolic execution in this direction. It can be further studied from the following aspects:

First, it is promising to design new code transformations specific to symbolic execution. As code transformations designed for symbolic execution scarcely exist, and it is very difficult to design such code transformations due to lack of knowledge in this direction, in this paper LEO accelerates symbolic execution by borrowing the knowledge of code transformation for *concrete* execution. Besides, LEO can be used as a light-weight approach to accelerating symbolic execution via code transformation. The results of LEO (e.g., the predictive model)

Table 7 Line coverage achieved by LEO/ALL/NO with random search heuristic and caching within 10-minute testing time limit.

Sub	LEO	ALL	NO	Sub	LEO	ALL	NO
chown	83.87(✔)	82.80(❍)	82.80(❍)	cp	48.24(✘)	49.05(✔)	48.78(❍)
csplit	68.62(❍)	64.59(✘)	75.60(✔)	date	85.06(✔)	80.52(✘)	82.47(❍)
echo	87.38(✔)	79.61(❍)	79.61(❍)	fmt	71.16(❍)	66.77(✘)	75.86(✔)
id	60.94(✔)	60.16(❍)	60.16(❍)	ln	82.99(✔)	76.80(✘)	77.84(❍)
ls	53.93(✔)	50.34(✘)	53.46(❍)	nice	96.61(✔)	94.92(✘)	96.61(✔)
nl	86.26(✔)	83.89(❍)	78.20(✘)	od	86.08(✔)	86.08(✔)	84.11(✘)
paste	92.51(✔)	92.51(✔)	92.51(✔)	pr	60.93(❍)	60.60(✘)	61.15(✔)
printenv	100.00(✔)	100.00(✔)	100.00(✔)	pwd	20.34(✔)	20.34(✔)	20.34(✔)
runcon	66.99(✔)	66.99(✔)	66.99(✔)	stat	63.23(✔)	57.38(✘)	62.40(❍)
touch	76.39(✘)	77.08(✔)	77.08(✔)	tr	55.24(❍)	55.69(✔)	54.32(✘)
Best(✔)	14	8	9	**Med(❍)**	4	4	8
Worst(✘)	2	8	3	–	–	–	–

can also provide knowledge to facilitate the design of code transformation specific to symbolic execution. That is, LEO can be regarded as a necessary step to code transformation for symbolic execution. Furthermore, when code transformations for symbolic execution are available, we can also use LEO to tune these transformations to achieve best acceleration performance due to the generality of our machine-learning based approach.

Second, code transformation can be combined with other symbolic execution optimization techniques (e.g., parallel and incremental symbolic execution). Code transformation manipulates the program under test by regarding symbolic execution as a black box, and thus it is orthogonal to other symbolic execution optimization techniques. For instance, through code transformation, a program is transformed into an easy-to-analyze target program, and then various optimized symbolic execution techniques can be applied to this new target program, making the analysis more easier. That is, LEO can further improve all the existing refined symbolic execution techniques because of the orthogonality.

5.2 Impact of Random Search Heuristic and Caching

The default KLEE random search heuristic and caching is disabled in our experimental study because the random search heuristic can bring much randomness and non-determinism and the caching contents can be different for different strategies, making it hard to evaluate the effectiveness of LEO. However, as random search heuristic and caching are widely used for symbolic execution, it is still interesting to know the performance of LEO with random search heuristic and caching on. Therefore, we conduct a preliminary study on 20 randomly-chosen GNU Coreutils programs with the similar setting as Section 4, but using random search heuristic and caching on. Table 7 shows the line coverage achieved by LEO, ALL, and NO with random search heuristic and caching. In particular, we repeat the experiments 5 times to reduce the impact of nondeterminism. From this table, with random search heuristic and caching, the number of subjects where LEO achieves the best performance (i.e., 14) is still larger than that of ALL (i.e., 8) and NO (i.e., 9), and the number of subjects where LEO achieves the worst performance (i.e., 2) is also still smaller than that of ALL (i.e., 8) and NO (i.e., 3). Furthermore, most subjects keep the same rankings with the results in Table 1. For example, LEO always performs the best for *chown* no matter whether using random search heuristic and caching. In particular, the main difference between using and not using random

search heuristic and caching is that when using them, the difference of LEO, ALL, and NO becomes smaller than that when not using them, no matter which technique performs the best. This is because random search heuristic and caching make symbolic execution more efficient and thus LEO, ALL, and NO achieve the similar (high) line coverage under the default 10-minute testing time limit. Therefore, *the setting with random search heuristic and caching has the orthogonal impact on accelerating symbolic execution with LEO.* That is, it further confirms that LEO, which optimizes *the code under test*, accelerates symbolic execution from an orthogonal dimension with techniques optimizing *symbolic execution itself*.

6 Related Work

In this section, we present the related work on both the symbolic execution and code transformation areas.

6.1 Symbolic Execution

Symbolic execution [29, 52, 17], is a systematic technique for generating program test inputs based on exploring all possible program paths, which has been recognized as one of the most costly testing methodologies. To improve the efficiency and effectiveness of symbolic execution, a huge amount of research effort [74, 81, 15, 28, 45, 50, 43, 70, 71, 34, 65, 58, 68, 79, 67, 83, 13, 44, 41, 85, 49, 11, 75, 3, 73, 67, 39, 59, 10, 35, 72] has been dedicated to the area, and more details on symbolic execution can be found in a recent survey [18]. To reduce the cost of symbolic execution, variants of symbolic execution have been proposed, e.g., concolic execution [42, 74] and execution-based testing [15, 16], which combine concrete execution with symbolic execution. Some researchers also proposed various techniques to accelerate the path exploration in symbolic execution. One specific approach is distributed symbolic execution where the path exploration is distributed among different workers [78, 77]. Since real-world programs usually consist of various sub-modules, a number of techniques have also been proposed to use the compositional approach to speed up symbolic execution [13, 44, 43]. Furthermore, Researchers have also proposed techniques to prune the search space of symbolic execution [79, 88].

Different from the above previous work on symbolic execution, our work accelerates symbolic execution via another orthogonal dimension – manipulating the programs under test via code transformation. Here we discuss closely related work applying code transformation to symbolic execution. Anand et al. [4] applied code transformation based on type-dependence analysis to help users identify problematic cases for symbolic execution and then the users can manually solve the problem. Dong et al. [33] showed that compiler optimizations could be harmful for symbolic execution via empirical study. Cadar [14] pointed out the potential direction of transforming program under test for better symbolic execution. Perry et al. [66] proposed code transformation rules specific to array operations to simplify constraints involving arrays for symbolic execution. Wagner et al. [84] and Converse et al. [31] mainly focused on reducing path exploration by simplifying program control-flow, but the generated tests may fail to cover the original program paths. In contrast, our work provides a general and fully automated machine-learning-based solution for accelerating symbolic execution based on all possible transformations.

6.2 Code Transformation

Compiler optimization is a typical and mature approach of code transformation [2, 32]. Besides, testability transformation [46], another code transformation approach, has been proposed to speed up search-based test generation [8, 53, 7], but cannot be directly applied to symbolic execution. Here we mainly review the work on compiler optimization [2, 32, 38, 19, 47, 1, 64, 9] since our work accelerates symbolic execution through this type of code transformation. Traditional work on compiler optimization focused on defining new optimizations and exploring their impacts on program concrete execution [2], whereas recently researchers focused on choosing the most suitable set of optimizations for general or specific target programs on concrete execution. More specifically, iterative compilation [32, 38, 19], a search-based approach that explores the compiler optimization space by iteratively compiling for single optimization objective (e.g., performance or code size) or multiple objectives [47]. Different from previous work searching for optimal compiler optimizations for program *concrete* execution, this paper presents the first work on predicting optimal compiler optimizations for *symbolic* execution based on machine learning.

7 Conclusion

Compiler optimization is a typical code transformation approach, which is firstly proposed to accelerate program concrete execution. In this paper, we present LEO, the first machine-learning-based approach to accelerating symbolic execution through tuning compiler optimizations. More specifically, LEO predicts compiler optimizations for source-code functions and libraries separately and applies the learnt optimization settings by program-splitter and local-optimizer. From our empirical study, compared with the default turning on/off all compiler optimizations in KLEE, LEO achieves the best acceleration performance in 50/68 GNU Coreutils programs and its average improvement rate on all programs is 46.48%/88.92% in terms of line coverage, with the default training/testing time limit. Furthermore, LEO consistently outperforms KLEE default settings with various training/testing time limits, and tends to perform the best when training and testing time limits are close.

References

1 F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO*, pages 295–305, 2006.

2 Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Pearson Education, Inc., 1986.

3 Aws Albarghouthi, Arie Gurfinkel, Ou Wei, and Marsha Chechik. Abstract analysis of symbolic executions. In *CAV*, pages 495–510, 2010.

4 Saswat Anand, Alessandro Orso, and Mary Jean Harrold. Type-dependence analysis and program transformation for symbolic execution. In *TACAS*, pages 117–133, 2007.

5 J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411, 2005.

6 Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *ISSTA*, pages 177–188, 2016.

7 André Baresel, David Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *ISSTA*, pages 108–118, 2004.

8 David W. Binkley, Mark Harman, and Kiran Lakhotia. Flagremover: A testability transformation for transforming loop-assigned flags. *TOSEM*, 20(3):12:1–12:33, 2011.

9 François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *PFDC*, 1998.

10 Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *FSE*, pages 411–421, 2013.

11 Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Symbolic execution of programs with heap inputs. In *FSE*, pages 602–613, 2015.

12 Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

13 William R Bush, Jonathan D Pincus, and David J Sielaff. A static analyzer for finding dynamic programming errors. *SPE*, 30(7):775–802, 2000.

14 Cristian Cadar. Targeted program transformations for symbolic execution. In *FSE*, pages 906–909, 2015.

15 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

16 Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *TISSEC*, 12(2):10, 2008.

17 Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *ICSE*, pages 1066–1071, 2011.

18 Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *CACM*, 56(2):82–90, 2013.

19 John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO*, pages 185–197, 2007.

20 Gavin C Cawley and Nicola LC Talbot. Efficient leave-one-out cross-validation of kernel fisher discriminant classifiers. *Pattern Recognition*, 36(11):2585–2592, 2003.

21 Nitesh V Chawla. Data mining for imbalanced datasets: An overview. In *Data Min. Knowl. Discov.*, pages 853–867. Springer, 2005.

22 Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *JAIR*, pages 321–357, 2002.

23 Nitesh V. Chawla, Nathalie Japkowicz, and Aleksander Kotcz. Editorial: Special issue on learning from imbalanced data sets. *SIGKDD Explor*, 6(1):1–6, 2004.

24 Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. Learning to prioritize test programs for compiler testing. In *ICSE*, pages 700–711, 2017.

25 Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, and Bing Xie. How do assertions impact coverage-based test-suite reduction? In *ICST*, pages 418–423, 2017.

26 Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. Supporting oracle construction via static analysis. In *ASE*, pages 178–189, 2016.

27 Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *ICSE*, pages 180–190, 2016.

28 Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *ICSE*, pages 144–155, 2016.

29 Lori A. Clarke. A system to generate test data and symbolically execute programs. *TSE*, SE-2(3):215–222, 1976. doi:10.1109/TSE.1976.233817.

30 Hayes Converse, Oswaldo Olivo, and Sarfraz Khurshid. Non-semantics-preserving transformations for high-coverage test generation using symbolic execution. In *ICST*, pages 241–252, 2017.

31 Hayes Elliott Converse, Oswaldo Olivo, and Sarfraz Khurshid. *Non-semantics-preserving transformations for higher-coverage test generation using symbolic execution*. PhD thesis, The University of Texas at Austin, 2017.

32 Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steve Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Exploring the structure of the space of compilation sequences using randomized search algorithms. *J Supercomput*, 36(2):135–151, 2006.

33 Shiyu Dong, Oswaldo Olivo, Lingming Zhang, and Sarfraz Khurshid. Studying the influence of standard compiler optimizations on symbolic execution. In *ISSRE*, pages 205–215, 2015.

34 Ikpeme Erete and Alessandro Orso. Optimizing constraint solving to better support symbolic execution. In *ICSTW*, pages 310–315, 2011.

35 Antonio Filieri, Corina S. Păsăreanu, Willem Visser, and Jaco Geldenhuys. Statistical symbolic execution with informed sampling. In *FSE*, pages 437–448, 2014.

36 Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *ICML*, pages 124–133, 1999.

37 Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *Ann. Stat.*, 28(2):337–407, 2000.

38 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 O'Boyle. Milepost gcc: Machine learning enabled self-tuning compiler. *IJPP*, 39(3):296–327, 2011.

39 Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *ISSTA*, pages 166–176, 2012.

40 Alexander Genkin, David D Lewis, and David Madigan. Large-scale bayesian logistic regression for text categorization. *Technometrics*, 49(3):291–304, 2007.

41 Patrice Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.

42 Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223, 2005.

43 Patrice Godefroid, Shuvendu K. Lahiri, and Cindy Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 112–128, Berlin, Heidelberg, 2011. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=2041552.2041564`.

44 Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, pages 43–56, 2010.

45 Shengjian Guo, Markus Kusano, and Chao Wang. Conc-ise: Incremental symbolic execution of concurrent software. In *ASE*, pages 531–542, 2016.

46 Mark Harman, André Baresel, David Binkley, Robert Hierons, Lin Hu, Bogdan Korel, Phil McMinn, and Marc Roper. Testability transformation–program transformation to improve testability. In *Formal methods and testing*, pages 320–344. Springer, 2008.

47 Kenneth Hoste and Lieven Eeckhout. Cole: Compiler optimization level exploration. In *CGO*, pages 165–174, 2008.

48 Y Kumar Jain and Santosh Kumar Bhandare. Min max normalization based data perturbation method for privacy protection. *IJRCCT*, 2(8):45–50, 2011.

49 Konrad Jamrozik, Gordon Fraser, Nikolai Tillmann, and Jonathan De Halleux. Augmented dynamic symbolic execution. In *ASE*, pages 254–257, 2012.

50 Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. Synthesizing framework models for symbolic execution. In *ICSE*, pages 156–167, 2016.

51 René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE*, pages 654–665, 2014.

52 James C King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.

53 Bogdan Korel, Mark Harman, S. Chung, P. Apirukvorapinit, R. Gupta, and Q. Zhang. Data dependence based testability transformation in automated test generation. In *ISSRE*, pages 245–254, 2005.

54 Tomasz Kuchta, Cristian Cadar, Miguel Castro, and Manuel Costa. Docovery: Toward generic automatic document recovery. In *ASE*, pages 563–574, 2014.

55 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86, 2004.

56 Saskia Le Cessie and Johannes C Van Houwelingen. Ridge estimators in logistic regression. *Applied statistics*, pages 191–201, 1992.

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

58 Daniel Liew, Cristian Cadar, and Alastair F. Donaldson. Symbooglix: A symbolic execution engine for boogie programs. In *ICST*, 2016.

59 Kasper Luckow, Corina S. Păsăreanu, Matthew B. Dwyer, Antonio Filieri, and Willem Visser. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *ASE*, pages 575–586, 2014.

60 P. D. Marinescu and Cristian Cadar. KATCH: High-coverage testing of software patches. In *FSE*, pages 235–245, 2013.

61 Paul Dan Marinescu and Cristian Cadar. High-coverage symbolic patch testing. In *SPIN*, pages 7–21, 2012.

62 Paul Dan Marinescu and Cristian Cadar. Make test-zesti: A symbolic execution solution for improving regression testing. In *ICSE*, pages 716–726, 2012.

63 Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *JASA*, 46(253):68–78, 1951.

64 Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO*, pages 319–332, 2006.

65 Corina S Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *ASE*, pages 179–180, 2010.

66 David M Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. Accelerating array constraints in symbolic execution. In *ISSTA*, pages 68–78, 2017.

67 Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *FSE*, pages 226–237, 2008.

68 Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.

69 John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical Report MSR-TR-98-14, Microsoft Research, April 1998.

70 Rui Qiu, Sarfraz Khurshid, Corina S Păsăreanu, Junye Wen, and Guowei Yang. Using test ranges to improve symbolic execution. In *NFM*, pages 416–434. Springer, 2018.

71 Rui Qiu, Corina S Păsăreanu, and Sarfraz Khurshid. Certified symbolic execution. In *ATVA*, pages 495–511. Springer, 2016.

72 Rui Qiu, Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. Compositional symbolic execution with memoized replay. In *ICSE*, pages 632–642, 2015.

73 Eric F. Rizzi, Mathew B. Dwyer, and Sebastian Elbaum. Safely reducing the cost of unit level symbolic execution through read/write analysis. *SEN*, 39(1):1–5, 2014.

74 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *FSE*, pages 263–272, 2005.

75 Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *FSE*, pages 842–853, 2015.

76 Junaid Haroon Siddiqui and Sarfraz Khurshid. ParSym: Parallel symbolic execution. In *ICSTE*, pages V1–405–V1–409, 2010.

77 Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. In *OOPSLA*, volume 47, pages 523–536, 2012.

78 Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *ISSTA*, pages 183–194, 2010.

79 Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. express: guided path exploration for efficient regression test generation. In *ISSTA*, pages 1–11, 2011.

80 Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *POPL*, volume 50, pages 83–95, 2015.

81 Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for .net. In *TAP*, pages 134–153. Springer, 2008.

82 Lisa Torrey and Jude Shavlik. Transfer learning. *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, 1:242, 2009.

83 Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *FSE*, pages 58:1–58:11, 2012.

84 Jonas Wagner, Volodymyr Kuznetsov, and George Candea. -overify: Optimizing programs for fast verification. In *HotOS XIV*. EPFL-CONF-186012, 2013.

85 Tao Wang, Abhik Roychoudhury, Roland HC Yap, and Shishir C Choudhary. Symbolic execution of behavioral requirements. In *PADL*, pages 178–192. Springer, 2004.

86 Xiaoyin Wang, Lingming Zhang, and Philip Tanofsky. Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. In *ISSTA*, pages 199–210, 2015.

87 Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *ICSE*, pages 620–631, 2015.

88 Guowei Yang, Corina S Păsăreanu, and Sarfraz Khurshid. Memoized symbolic execution. In *ISSTA*, pages 144–154, 2012.