DASE: Document-Assisted Symbolic Execution for Improving Automated Test Generation

by

Lei Zhang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Software testing is crucial for uncovering software defects and ensuring software reliability. Symbolic execution has been utilized for automatic test generation to improve testing effectiveness. However, existing test generation techniques based on symbolic execution fail to take full advantage of programs' rich amount of documentation specifying their input constraints, which can further enhance the effectiveness of test generation.

In this thesis we propose a general approach, *Document-Assisted Symbolic Execution* (*DASE*), to improve automated test generation and bug detection. DASE leverages natural language processing techniques and heuristics to analyze programs' readily available documentation and extract input constraints. The input constraints are then used as pruning criteria; inputs far from being valid are trimmed off. In this way, DASE guides symbolic execution to focus on those inputs that are semantically more important.

We evaluated DASE on 88 programs from 5 mature real-world software suites: GNU Coreutils, GNU findutils, GNU grep, GNU Binutils, and elftoolchain. Compared to symbolic execution without input constraints, DASE increases line coverage, branch coverage, and call coverage by 5.27–22.10%, 5.83–21.25% and 2.81–21.43% respectively. In addition, DASE detected 13 previously unknown bugs, 6 of which have already been confirmed by the developers.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Lin Tan, for her consistent and invaluable mentorship, without which, this thesis would not have been possible. In addition, I would like to thank the readers, Professor Vijay Ganesh and Professor Mahesh Tripunitara, for their valuable time and comments. I would also like to extend my appreciation to my project colleague, Edmund Wong, for his help with the natural language processing part of this thesis. The KLEE community, especially Professor Cristian Cadar, also deserves my gratefulness for their help with my understanding of KLEE.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software testing is an essential part of software development. It is estimated that inadequate testing costs billions of dollars to our economy annually [45]. To improve testing, many automated test generation techniques are proposed and used [10, 11, 12, 15, 26, 46]. Compared to manually creating test cases, which is time- and effort-consuming, automatic test generation is a more efficient way to achieve high code coverage and detect bugs.

Symbolic execution [20, 29] has been leveraged to generate high code coverage test suites effectively and detect previously unknown bugs [9, 17, 22, 31, 35, 50]. Symbolic execution represents inputs as symbolic values instead of concrete values. Upon exploring a branch whose condition involves symbolic values, two paths are created, and corresponding constraints are put on each path. In this way, symbolic execution can systematically explore all the execution paths. Once the execution of a path terminates, the collection of constraints along that execution path can be used to generate concrete inputs to exercise the path. Symbolic execution suffers from the fundamental problem of path explosion. In practice, one needs to use search heuristics and other techniques to guide symbolic execution [10, 14, 26, 34, 35].

Although symbolic execution has been successful in improving testing effectiveness, existing techniques do not take full advantage of programs' high level input constraints. Program inputs typically need to follow certain constraints. For example, `rm` only accepts 12 options including `-r`, `-f` and others, and `readelf` requires its input files to follow Executable and Linkable Format (ELF). Focusing on the valid or close-to-valid inputs can help test the core functionalities of the program, which should improve testing coverage and effectiveness. Fortunately, information about such input constraints commonly exists in various sources, such as argument parsing functions, programs' help manuals (e.g., the

output of `rm --help`), and library header files (e.g., the comments and code in `elf.h`).

Thus, we propose a general approach, *Document-Assisted Symbolic Execution* (*DASE*), to enhance the performance of symbolic execution for automatic test generation and bug detection. DASE automatically extracts input constraints from documents, and use these constraints as a "filter" to favor execution paths that execute the core functionalities of the program. This allows symbolic execution to devote more resources on testing code that implements program's core functionalities, as opposed to code for input sanity check and error handling. DASE, as a path pruning strategy, can be used on top of existing search strategies to further improve symbolic execution.

DASE considers two categories of input constraints: valid choices for a command-line option (e.g., `-r` for `rm`) and the format for an input file (e.g., ELF). These two types are sufficient for a wide spectrum of programs. In addition, one can convert interactive programs into command-line programs [27].

1.1 Research Contributions

By combining symbolic execution with input constraints from documentation, this thesis makes the following contributions:

- We propose a novel approach, *Document-Assisted Symbolic Execution* (*DASE*), to improve dynamic test generation. By incorporating input constraints from documentation, DASE enables symbolic execution to distinguish the *semantic* importance of different execution paths to focus on programs' core functionalities, thus being more effective.

- DASE combines natural language processing techniques, for instance, grammar relationships, and heuristics to automatically extract input constraints from documents, including valid values for command-line options and file format constraints for ELF. File format constraints enable us to build a partial ELF file model that can possibly be used for other tasks such as program comprehension and constraint verification.

- Our evaluation shows that DASE outperforms KLEE [15] on 88 programs from 5 mature widely-used software suites—GNU Coreutils, GNU findutils, GNU grep, GNU Binutils, and elftoolchain. Compared to KLEE, DASE increases line coverage, branch coverage, and call coverage by 5.27–22.10%, 5.83–21.25%, and 2.81–21.43% respectively. In addition, DASE detected 13 previously unknown bugs, 6 of which have been confirmed by the developers after we reported the bugs to them.

1.2 Thesis Organization

This thesis is divided into nine chapters. In Chapter 1, we introduce the problem and outline our research contributions. In Chapter 2, we describe the intuition and general idea behind DASE. Then we introduce the background of command-line options, ELF file format, and KLEE symbolic execution in Chapter 3. The design and implementation of DASE are presented in Chapter 4. Chapter 5 presents our experimental method and Chapter 6 lists the results. Previous attempts and further work are presented in Chapter 7. Chapter 8 contains related work. Finally, we conclude our findings in Chapter 9.

Chapter 2

Intuition and Overview

A real world program typically contains numerous or even infinite number of execution paths. Given limited time, it is crucial for testing to effectively prioritize the paths. Researchers have proposed approaches to guide the path exploration of symbolic execution [10, 14, 26, 34, 35], which have been shown to improve code coverage.

Path pruning, which applies a "filter" to prune "uninteresting" paths before employing a search strategy, can be used to further address the path explosion problem. Path pruning significantly reduces the size of the search space for a search strategy.

We propose using *input constraint*s as a "filter" to aid search strategies to focus on valid and close-to-valid inputs (e.g., boundary cases). The core functionality of a program is typically related to processing valid inputs. For example, a C compiler's core functionality is parsing and compiling valid C programs. Valid C programs are only a small portion of all strings (the input space of a C compiler).

Randomly generated inputs can cover many invalid inputs, but miss valid and close-to-valid ones. While symbolic execution addresses this issue by exploring paths systematically, it is unaware of which branch (the "then" branch or the "else" branch) leads to valid inputs upon a conditional statement. Input constraints that define valid inputs can guide symbolic execution to focus on paths corresponding to valid and close-to-valid inputs. These paths can pass the trivial part of input sanity check to go deeper and are more likely to cover buggy code, thus more "interesting".

Keeping invalid inputs in the search space hurts the effectiveness of symbolic execution based test generation, because exploring invalid inputs takes up time and memory, which can be used for testing valid and close-to-valid inputs instead. Specifically, the two reasons

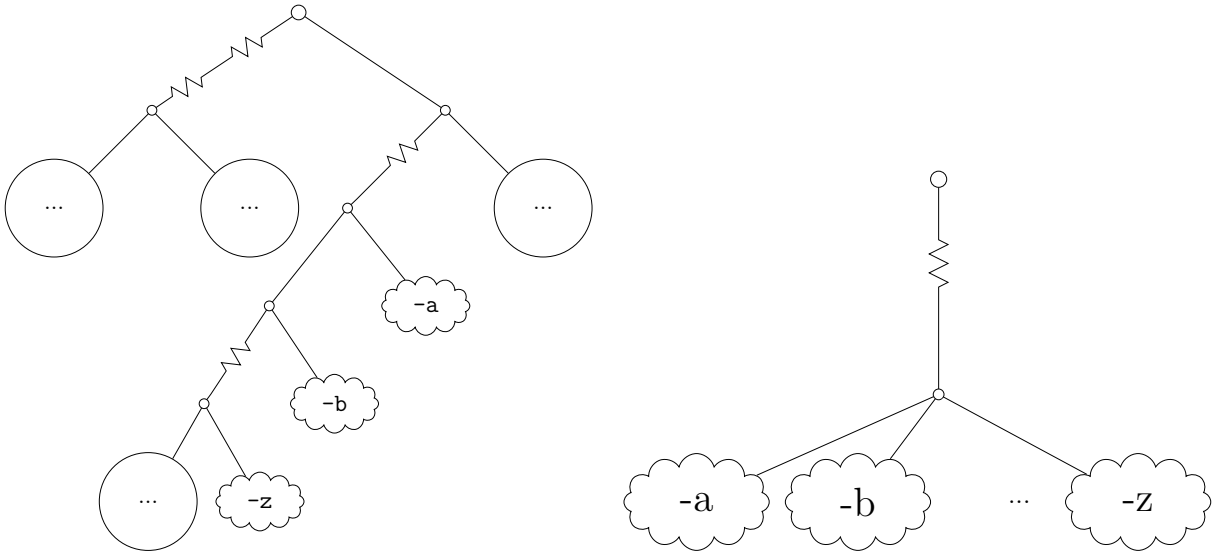
are: (1) search strategies have more chances to deviate from the paths related to valid inputs, and be stuck at "shallow" paths for a long time; and (2) more constraints need to be solved and search strategies need to compute metrics for invalid inputs in every round, which is a waste of computation.

A program typically accepts two types of input *argument*s: *option*s and *input file*s. Thus, there are two categories of constraints to leverage: valid values of an option, and the format of an input file. Below we explain in detail why path pruning based on these two categories of constraints can improve testing coverage and how we perform path pruning in our prototype, DASE. Since path pruning is used to reduce the search space of search strategies, we explain it in the context of a search strategy. DASE is built on top of a widely-used symbolic execution based test generation tool KLEE [15]. We use KLEE's default search strategy (explained in the following paragraph) as an example to aid explanation. Nonetheless, the idea of path pruning is applicable to other search strategies.

During execution, KLEE gradually builds a *binary execution tree*. The *leaf node*s of this tree are the current execution *state*s and the *non-leaf node*s are the branching points in the program. KLEE's default search strategy consists of two atom search strategies that are interleaved in a round robin fashion to prevent one atom strategy from getting stuck. The first atom strategy is random path selection, which traverses the execution tree from the root and randomly selects a branch to follow at each non-leaf node, until it reaches a leaf node. This strategy favors "shallow" leaf nodes to alleviate starvation. The second strategy, coverage-optimized search, tries to choose a state that is most likely to cover new code in the immediate future. This strategy uses a heuristic weighting scheme that considers factors such as the minimum distance to an uncovered instruction.

2.1 How can option-related constraints help?

Command-line options are used to tune parameters or invoke certain functionalities of programs. For example, the option `-r` tells `rm` to perform a recursive deletion. A comprehensive test suite must cover the option `-r` to explore the code segment for recursive deletion. After input argument parsing, a program typically uses a switch-case structure to check the input argument against all valid options until a match is found, and then the corresponding functionality is invoked. Figure 2.1 illustrates the execution trees before and after path pruning. Without pruning, testing all valid options of a program requires a search strategy to make good choices on the execution tree in Figure 2.1a so that all clouds (subtrees related to valid command-line options) are covered. Options are at differ-



(a) Before Pruning. (b) After Pruning.

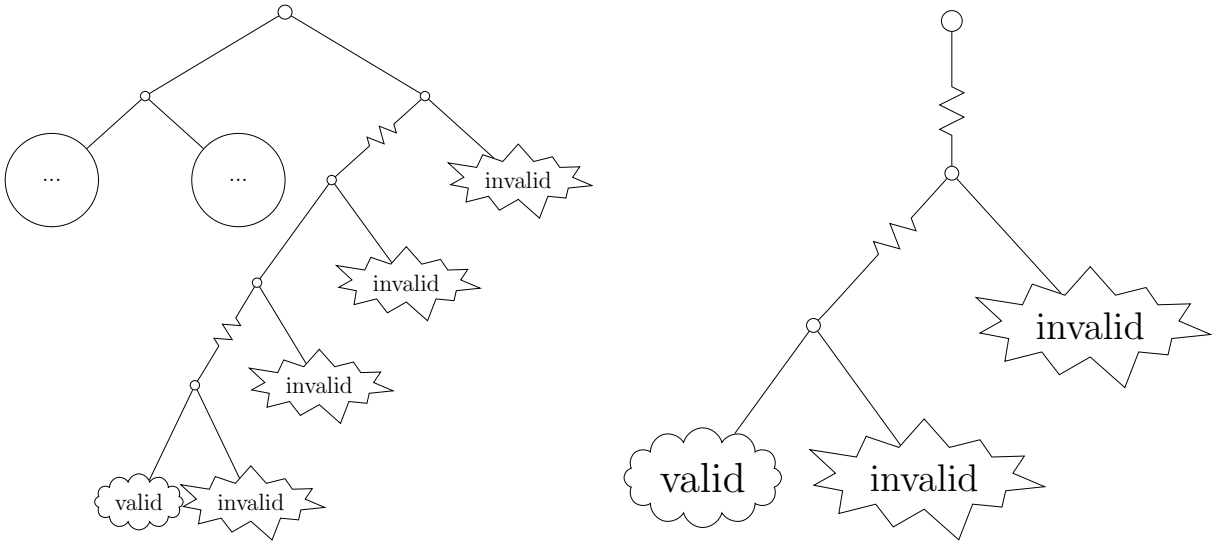
Figure 2.1: Abstract view of execution trees for command-line options, which illustrates that pruning can potentially improve testing coverage. Clouds are execution subtrees related to valid command-line options. Circles are other execution subtrees.

ent depths; it is easy for search strategies to miss deep options because of reason (1) and (2) discussed earlier.

DASE knows what options are valid by analyzing programs' documentation, which enables symbolic execution to explore all valid options directly as shown in Figure 2.1b. Given n valid options, DASE "forks" the symbolic execution n times[1], with each forked execution branch taking a valid option. This approach concretizes the symbolic option with a valid option one at a time, which essentially trims off paths related to all other options for each forked execution. Symbolic execution can go directly to the core functionality code related to the valid option. This allows symbolic execution to spend more time on code related to the valid option (especially if the valid option is deep before pruning), which should improve testing coverage. In addition, some constraints are solved or simplified (e.g., the ones related to the concrete valid option), which reduces the computation time of the constraint solver.

Although the after-pruning-approach may appear to be similar to breadth-first search

[1]We fork additional executions for an invalid option and a null option for completeness. But the additional executions should take less time than the circles in Figure 2.1a combined.



(a) Before Pruning. (b) After Pruning.

Figure 2.2: Abstract view of execution trees for input file formats.

(BFS), it is very different from BFS. Without pruning, BFS will explore paths in Figure 2.1a, which still suffers from the two problems (1) and (2) described above. In fact, our evaluation shows that DASE outperforms KLEE even if BFS is used as an underlying search strategy (Section 6.1.2). Chapter 8 discusses how DASE is different from other search strategies.

2.2 How can file-format-related constraints help?

File-format-related constraints can filter out less interesting paths especially if the file format is complex. Programs parse and process input files (including sanity check) gradually. The high level execution tree is shown in Figure 2.2a. There are many aborts resulting from invalid inputs along the process. These aborts typically point to the error handling logic. With all these execution paths for invalid inputs, search strategies are continuously distracted. On the contrary, input constraints can trim them off and directly drive symbolic execution deeper to start exploring interesting code there as shown in Figure 2.2b. DASE builds a file format model by combining file structure with value-based constraints that are extracted from documentation, and uses the model to prune paths. We focus on the Executable and Linkable Format (ELF) in this thesis. ELF is the underlying stan-

dard for all binaries in Unix-like systems. Any program that reads or writes binaries on a Unix-like platform needs to parse files in ELF format. In this thesis, we evaluate DASE on 4 programs that use the ELF file format. Since ELF is broadly used and of crucial importance, DASE can be used to improve test generation for many other programs.

Chapter 3

Background

This chapter gives a brief background about option styles (Section 3.1), ELF file format (Section 3.2), and KLEE (Section 3.3).

3.1 Option Styles

Command-line options are used to tweak the logic of programs from outside, which is important especially for programs that function as pipes or filters. There are several command-line option styles existing in Unix-like operating systems, including the popular POSIX- and GNU-style, which are described below.

- POSIX-style [6] is generally in the form of *short options*, namely, a single dash followed by a letter (e.g., `ps -a`). Letters from distinct options can be combined into one word under certain circumstances. For example, `-xy` may indicate the same meaning as `-x -y`. Some options may require additional arguments. These options and their arguments may or may not be separated by whitespaces (e.g., both `ps -u root` and `ps -uroot` are valid).
- GNU-style [8] extends the POSIX-style with *long options*, which are generally in the form of two dashes followed by several words concatenated by single dashes (e.g., `ps --no-headers`).

- Apart from the above two styles, there exist several other styles. `ps` also supports BSD-style. BSD-style is similar to POSIX-style but without dash prefix (e.g., `ps a`). Options can also be combined if no conflict is introduced. Other programs, such as `dd`, use a style without any dash, e.g., `dd if=FILE of=FILE`.

A program may support several option styles for convenience and compatibility. Additionally, any program can define its own option style. It is not possible to cover all of them, therefore we focus on the first two prevalent styles and discuss how we extract valid options in Section 4.1.

3.2 Executable and Linkable Format

Executable and Linkable Format (ELF) [1] is a fundamental standard for Unix-like systems. It specifies the underlying layout for executable files, object files, shared libraries, and core dumps. An ELF file is usually composed of multiple sections as well as several headers serving as "road map" to the sections as shown in Figure 3.1. We briefly describe the basic layout here.

At the beginning of any ELF file is an *ELF header*, which contains ELF identification information, target machine, and several other attributes. The ELF header is defined as a C struct of type `Elf32_Ehdr` and `Elf64_Ehdr` in `elf.h`. For example, `Elf64_Ehdr` is

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
  Elf64_Half    e_type;             /* Object file type */
  Elf64_Half    e_machine;          /* Architecture */
  Elf64_Word    e_version;          /* Object file version */
  Elf64_Addr    e_entry;            /* Entry point virtual address */
  Elf64_Off     e_phoff;            /* Program header table file offset */
  Elf64_Off     e_shoff;            /* Section header table file offset */
  Elf64_Word    e_flags;            /* Processor-specific flags */
  Elf64_Half    e_ehsize;           /* ELF header size in bytes */
  Elf64_Half    e_phentsize;        /* Program header table entry size */
  Elf64_Half    e_phnum;            /* Program header table entry count */
  Elf64_Half    e_shentsize;        /* Section header table entry size */
  Elf64_Half    e_shnum;            /* Section header table entry count */
  Elf64_Half    e_shstrndx;         /* Section header string table index */
} Elf64_Ehdr;
```

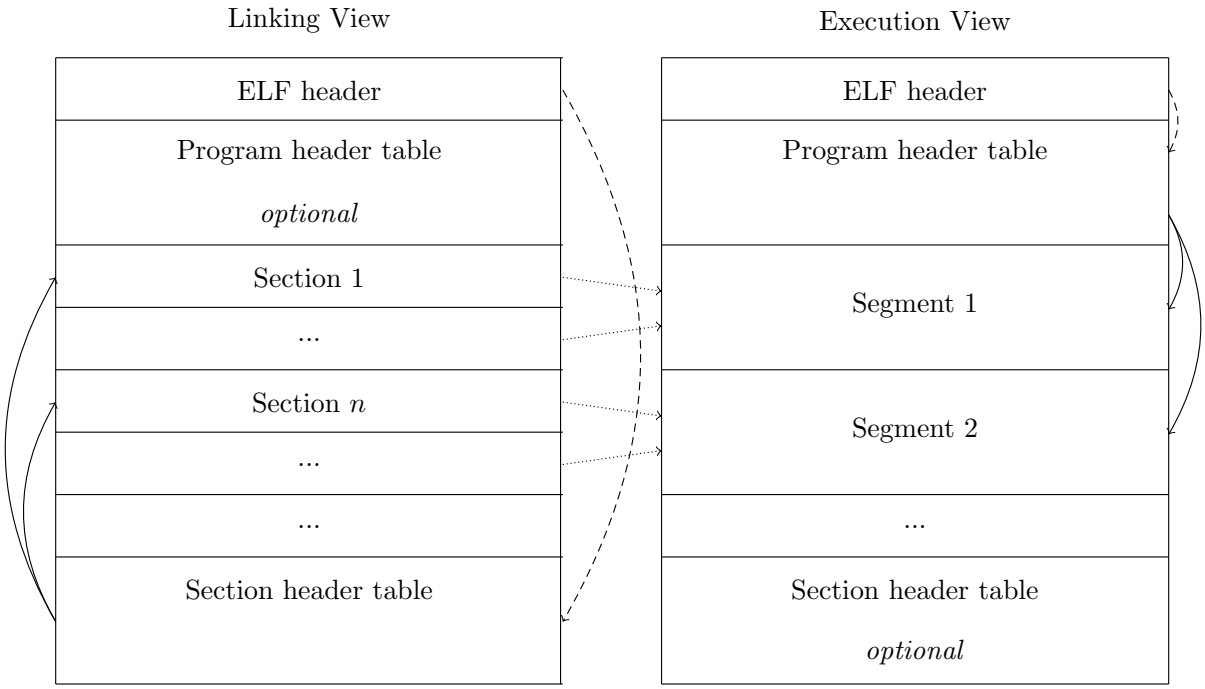


Figure 3.1: ELF structure from ELF specification [1]. Lines with arrowheads represent the relationships between different components.

For convenience, we will use `Elf*` to represent both `Elf32` and `Elf64` in the rest of this thesis. The ELF header also includes information about two other important header tables: the *section header table* (SHT) and the *program header table* (PHT), as shown by dashed lines in Figure 3.1.

The SHT, which is useful for program linking, records the locations and attributes for all sections; while the PHT, which is useful for program loading, tells the system how to create a process image from the file on disk. The SHT/PHT contains an array of *section header*s/*program header*s, as shown by solid lines in Figure 3.1.

A section header, which is defined as a C struct of type `Elf*_Shdr`, contains the information such as name, type, and size for the corresponding section. A program header contains information for a *segment*, which is composed by sections with similar attributes for loading (shown in Figure 3.1 using dotted lines). Object files' real information, such as instructions and data, is held in *section*s. Because section headers and program headers are important to our ELF model, we just copy their definitions (for 64-bit machines) here from `elf.h`.

```
typedef struct
{
  Elf64_Word  sh_name;      /* Section name (string tbl index) */
  Elf64_Word  sh_type;      /* Section type */
  Elf64_Xword sh_flags;     /* Section flags */
  Elf64_Addr  sh_addr;      /* Section virtual addr at execution */
  Elf64_Off   sh_offset;    /* Section file offset */
  Elf64_Xword sh_size;      /* Section size in bytes */
  Elf64_Word  sh_link;      /* Link to another section */
  Elf64_Word  sh_info;      /* Additional section information */
  Elf64_Xword sh_addralign; /* Section alignment */
  Elf64_Xword sh_entsize;   /* Entry size if section holds table */
} Elf64_Shdr;


typedef struct
{
  Elf64_Word  p_type;    /* Segment type */
  Elf64_Word  p_flags;   /* Segment flags */
  Elf64_Off   p_offset;  /* Segment file offset */
  Elf64_Addr  p_vaddr;   /* Segment virtual address */
  Elf64_Addr  p_paddr;   /* Segment physical address */
  Elf64_Xword p_filesz;  /* Segment size in file */
  Elf64_Xword p_memsz;   /* Segment size in memory */
  Elf64_Xword p_align;   /* Segment alignment */
} Elf64_Phdr;
```

Detailed explanation is available in the specification [1]. We focus on this basic layout and describe our ELF model in Section 4.3.

3.3 KLEE Symbolic Execution

KLEE is a symbolic execution engine based on the LLVM compiler framework [3]. Programs are firstly compiled into LLVM bytecode, which uses RISC-like virtual instruction set, and then directly interpreted by KLEE. KLEE manipulates programs' running states including registers, stacks, heaps and program counters. In addition, KLEE models the environments that programs interact with. Therefore, "at a high level, KLEE functions as a hybrid between and operating system for symbolic processes and an interpreter."

The core of KLEE is an interpreter loop. For each iteration, an execution state is chosen to run, and several instructions in the context of that state are symbolically executed. This loop continues until all states are explored, or timeout threshold is reached. KLEE provides many strategies for selecting which states to explore next; the default is explained in the previous chapter. For symbolically executing an instruction, the KLEE paper [15] gives an example:

```
%dst = add i32 %src0, %src1
```

"KLEE retrieves the addends from the `%src0` and `%src1` registers and writes a new expression `Add(%src0, %src1)` to the `%dst` register." It is clear that what should be raw data values (`%dst`) in normal execution becomes expressions (the `Add` expression) when symbolically executed. These expressions have symbolic variables or constants as their leaves, and LLVM language operations (e.g., arithmetic operations, comparisons, etc.) as the interior nodes. If all leaves are concrete, the whole expression will be evaluated natively into a concrete value for efficiency purposes.

The execution of conditional branches is different from the way for the previous assignment instruction. The program counter will be altered according to the value of the condition. KLEE queries the constraint solver to determine whether the branch condition is provably true or provably false; if it is the case, the program counter is changed to the appropriate location. If both branches are possible, the current state will be cloned into two, with the program counters updated correspondingly. Thus, both paths can be explored.

KLEE detects bugs by wrapping dangerous operations with special checks. For example, a division operation will incur an additional check against whether the divisor is zero. If so, a test case triggering the bug will be generated. Load and store instructions are also wrapped to check whether the access is in-bounds of a valid memory object.

At the beginning of a program, memory objects that users are interested in must be marked as symbolic; otherwise, all the instructions manipulate concrete values. KLEE provides an intrinsic function `klee_make_symbolic()` to symbolicize memory, whose usages are tracked and constraints are collected. Moreover, KLEE also provides mechanisms to symbolicize the memory objects corresponding to command-line options. `klee_init_env()` intercepts the startup of programs and inserts logic to make them support symbolic options:

- `--sym-args MIN MAX N`, which represents at least `MIN` and at most `MAX` symbolic arguments, each with a maximum length `N`,

- `--sym-files NUM N`, which makes `stdin` and up to `NUM` symbolic files, each with a maximum size `N`,
- `--sym-stdout`, which symolicizes `stdout`.

One can use `--max-time` to specify the maximum allowed execution time, and use `--max-memory` to restrict the memory used.

KLEE uses STP [7] for constraint solving. Since the cost of constraint solving dominates everything else, KLEE implements many optimizations before issuing a query to STP. Instead of mapping the program's address space as a flat byte array, different memory objects are mapped into distinct STP arrays. Furthermore, KLEE provides query optimizations including expression rewriting, constraint set simplification, implied value concretization, constraint independence, and counter-example cache. The KLEE paper [15] explains these optimizations in detail. When designing our ELF model, we also spend a lot of effort to guarantee that it does not incur much overheads for STP.

Chapter 4

Design and Implementation

We describe how DASE extracts and utilizes the constraints for options (Section 4.1 and Section 4.2) and ELF (Section 4.3 and Section 4.4) in this chapter.

4.1 Extracting Valid Options

We can extract a program's valid command-line options from various sources like argument parsing functions (e.g., `getopt_long` from GNU getopt library [4]), help manuals (e.g., the output of `ls --help`), man pages (e.g., `man ls`), and other documentation. Among them, programs' argument parsing functions are the most reliable source. So, we extract valid options from them when applicable. We use the help manuals as a fallback.

The API for `getopt_long` is

```
getopt_long(argc, argv, short_options, long_options, NULL)
```

where the parameter `short_options` contains the short options (starting with `-`), and the parameter `long_options` contains the long options (starting with `--`). For example, `mkdir`'s `short_options` is "`pm:vZ:`", which means `mkdir` has four POSIX-style options, and two of them (`-m` and `-Z`) require arguments (denoted by ":"). Variable `long_options` is an array of `struct option`, each of which defines one long option. For example, one element in `mkdir`'s `long_options` array is {`"verbose", no_argument, NULL, 'v'`}, which means

`--verbose` is a long option that requires no additional argument (`no_argument`) and is equivalent to the short option `-v` (`NULL` is irrelevant here).[1]

Therefore, if a program utilizes GNU `getopt_long` to parse its options, it is straightforward to obtain all valid options. We first identify the invocation of `getopt_long` in a program to acquire the values for its parameters `short_options` and `long_options`. We then parse the values of these two variables based on their formats defined in the API to extract the options supported.

Some programs do not use a standard argument parsing function to parse their inputs. While it is still possible to analyze these special parsing functions to extract all valid options, the parser will not generalize. Therefore, we build a help manual parser to extract valid options from the programs' help manuals, which follow a standard format. For example, help manual for `echo` is

```
Usage: ./echo [OPTION]... [STRING]...
Echo the STRING(s) to standard output.
-n             do not output the trailing newline
-e             enable interpretation of backslash escapes
-E             disable interpretation of backslash
               escapes (default)
   --help      display this help and exit
   --version   output version information and exit
```

Since both argument parsing functions and help manuals are simple and well structured, we do not employ complex techniques to complete the option extraction task; our parsers perform simple regular expression matching, which is effective and accurate.

4.2 Options as Guide for Symbolic Execution

DASE takes the options extracted in Section 4.1 to trim and reorganize the dynamic symbolic execution tree as shown in Figure 2.1. A program is usually tested with several arguments. Rather than marking all arguments symbolic, DASE singles one out. This selected symbolic argument is concretized with the program's valid options one by one, and each valid option will be carried out in a new execution branch. Specifically, if a program supports n options, DASE will create n execution branches. At each branch, instead of

[1]For more information about the meaning of this long option, please refer to http://www.gnu.org/software/libc/manual/html_node/Getopt-Long-Options.html

having m symbolic arguments, DASE runs the program with $m-1$ symbolic arguments and a valid option. For example, to test `echo`, instead of using `echo --sym-args 0 m x` (KLEE's terminology, `x` means the length of each option), we use `echo arg --sym-args 0 m-1 x`, where `arg` will be substituted by `-n`, `-e`, `-E`, `--help`, and `--version` one by one at the five generated execution branches.

To facilitate this process, we add another intrinsic function, `klee_enumerate()`, into KLEE. `klee_enumerate()` accepts a symbolic variable and a list of n concrete choices. `klee_enumerate()` then forks the current state into n child states and the symbolic variable in each child state is set to be one of the choices in the list.

In this way, the selected argument becomes one concrete valid option at each branch, which essentially trims off paths related to all other options for each forked execution. In addition, the concretization of the selected argument covers all the valid options at the same depth of the execution tree, which means all valid options are treated equally from the beginning. We can also think this technique as a "partition" of execution tree; the aim is to balance the effort on each option (hence the corresponding functionality), which should be of the same semantic importance.

To ensure the completeness of this "partition", we also consider invalid option and null option. In other words, apart from valid options extracted from the program's documentation, the selected argument is also filled with an invalid option and a null option.

The generated branches are then prioritized by search strategies.

4.3 Building the ELF Model

As introduced in Section 3.2, data in ELF is stored in sections. There are several types of headers, i.e., the ELF header, section headers, and program headers, acting as "road map" and meta-information containers to the sections. These headers record sections' attributes such as locations, sizes, and types, which tell us where each section is in the file and how to handle them properly.

Therefore, building an ELF model requires (1) setting up the "road map" layout and (2) specifying constraints for sections' attributes. We build the layout by reading the ELF specification, and then automatically extract constraints for section attributes from `elf.h` (its comments are written in natural language) to complete the ELF model. Our results show that both the layout and the constraints contribute to the coverage improvement of DASE.

For the layout, it is mainly the positioning of the headers and the sections. Except that the ELF header must appear at the beginning of the ELF file, the SHT, the PHT, and sections are quite flexible as their positioning. However, to reduce the workload of the constraint solver and also focus on important parts of ELF, we adopt a relatively rigid layout in our model as shown in Figure 4.1. The ELF header is followed directly by the SHT and then the PHT. This is enforced by fixing the `e_shoff` (SHT's offset from the beginning of the file) and `e_phoff` (PHT's offset from the beginning of the file) attributes in the ELF header. The SHT is set to have 5 section headers, which correspond to the 5 sections after the PHT. The first section (at index 0) is always a null section, and we force the following three sections to be a section header string table (holding null-terminated strings for section names), a symbol table (which is an array of symbols used for relocating), and a dynamic section (which is an array of symbols used for dynamic linking). We keep the fifth section as a random section. The PHT is set to contain only one random program header. For the second and fifth sections, we set their size to 8 bytes; for the third and fourth sections, we set their size to be enough for two symbols. Example constraints for the layout are:

```
assume(ehdr->e_shoff == sizeof(Elf*_Ehdr));
assume(ehdr->e_phoff ==
       sizeof(Elf*_Ehdr) + 5 * sizeof(Elf*_Shdr));
......
assume(shdr2->sh_type == SHT_SYMTAB);
assume(shdr2->sh_size == 2 * sizeof(Elf*_Sym));
......
```

where `assume()` is a function for putting constraints onto the current path. The boolean expression in the function must be true thereafter. `ehdr` and `shdr2` are pointers to the ELF header and the second section header respectively.

Specifically, the code is implemented in KLEE's POSIX emulation layer. KLEE models a plain symbolic file as a sequence of bytes in memory and symbolicizes them using `klee_make_symbolic()` (in `runtime/POSIX/fd_init.c`). We append the above constraints right after symbolicizing the memory. `assume()` in the above is implemented using KLEE intrinsic function `klee_assume()`, which accepts boolean expressions and enforces them to be true.

Our model is designed to reflect the basic structure of ELF, without introducing too much overhead for the constraint solver at the same time. We manually enforce the types of several sections because those section types are important for an ELF file and should be covered in testing.

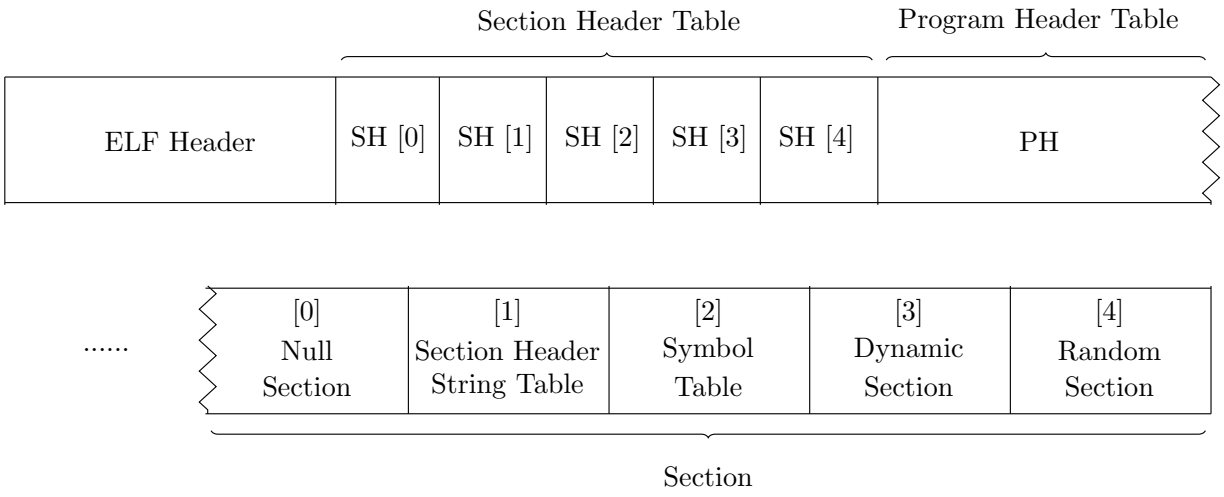


Figure 4.1: ELF model for DASE. "SH"/"PH" is short for "Section Header"/"Program Header". The numbers in brackets are array indices.

The other attributes in various headers are restricted using the constraints automatically extracted from `elf.h`. We explain the technique in the following section. Note that with the constraints from the following section, our ELF model is still incomplete. We retain this incompleteness to give DASE the ability to explore close-to-valid inputs, which are also "interesting".

4.4 Extracting Complex Input Constraints

In addition to the layout constraints in Section 4.3, we apply NLP techniques to automatically analyze the comments and code in the ELF header file (`elf.h`) to extract constraints. The ELF specification [1] and the man page [2] can also be used for input constraint extraction. However, they can easily become outdated and incomplete. For example, the ELF specification misses valid values for data fields such as "e_type" and "e_machine", and does not describe 64-bit related data structures. The header file `elf.h` is compiled together with the target code to test, therefore, it is consistent with the latest ELF file format under test.

The header file contains a large number of comments that describe the constraints for the struct data fields (i.e., each comment is followed by a list of macros representing the valid values). One example is:

```
/* Fields in the e_ident array.  The EI_* macros
   are indices into the array.  The macros under each
   EI_* macro are the values the byte may have. */
#define EI_MAG0     0
#define ELFMAG0     0x7f

#define EI_MAG1     1
#define ELFMAG1     'E'
...
```

DASE automatically generates the following two constraints regarding array index-value pairs from the comments and code. The rest of this section explains the NLP techniques used by DASE to generate the constraints.

```
assume(Elf32_Ehdr->e_ident [EI_MAG0] == ELFMAG0);
assume(Elf32_Ehdr->e_ident [EI_MAG1] == ELFMAG1);
```

Our technique extracts two types of value constraints: array index-value pairs and struct field values (e.g., `assume( Elf32_Shdr→e_type == 0 | ...);`). Since comments are written in a natural language, developers can use different forms to express the same meaning. For example, they may use "Fields in the e_ident array", "Fields of the e_ident array", "The e_ident array's fields", or "The array e_ident's fields" to start the listing of fields. These sentences use different sentence structures and different words to express the same meaning, which are difficult to analyze automatically. Simple regular expression matching will fail to accommodate all these and other variants.

We use typed dependency to analyze the dependencies and grammatical relations among words and phrases in a sentence to handle these variants. Compared to classification approaches used in prior work for comment analysis [44], no training data is required, thus less manual work. DASE uses the Stanford parser[2] to generate typed dependencies.

The grammar rules (GR) used to identify relevant comments and extract constraints from them are shown below. The main rules (GR2, GR3, and GR4) are used to identify relevant comments (if a sentence contains the typed dependency defined by a GR, it is considered relevant and remains for further analysis), then the supporting rules help identify the parameters in a rule, e.g., array and field names.

GR1 Noun or Adjectival Modifier (support rule)

[2] http://nlp.stanford.edu/software/stanford-dependencies.shtml

Noun or Adjectival modifier is an noun or adjectival phrase that modifies a noun phrase [21]. For example, in the comment "Fields in the e_ident array", the noun phrase "e_ident" modifies the noun "array". DASE applies this grammar relationship to retrieve data structure names and index names.

GR2 Prepositional Modifier (main rule)

Prepositional Modifier is a prepositional phrase that modifies the meaning of a verb, adjective, noun or preposition [21]. For example, in the comment "Legal values for sh_type field of Elf32_Shdr", the prepositional phrase "for ... Elf32_Shdr" modifies the noun "values". DASE applies this grammar on modifiers (i.e., "for", "of" and "in") to locate specific nouns (i.e., "value" and "field").

After locating the prepositional modifier, the dependency tree links "values" to the content word "field". If the content word is being modified by an adjectival modifier, DASE applies GR1 to resolve the properties. In this example, GR1 will return "sh_type" as the property of "field", and GR2 will flag the macros as the legal values for that data field.

GR3 Nominal subject (main rule)

Nominal subject is a noun phrase that is the syntactic subject of a clause [21]. For example, in the comment "The EI_* macros are indices into the array". The noun, "macros", is the subject of the clause, "indices into the array". DASE applies this grammar to locate specific clauses (i.e., "indices ..." and "values ...").

After locating the nominal subject, DASE applies GR1 to resolve the properties. In this example, GR1 will return the regular expression "EI_*" as the property of "macros", and GR3 will flag the macros named under this regular expression as the indices of an array.

GR4 Possession modifier (main rule)

Possession modifier holds the relation between the head of a noun phrase and its possessive determiner [21]. For example, in the comment "sh_type field's legal values". The head noun is "field" and the possessive determiner is "values". DASE applies this grammar to locate specific possessive determiners (i.e., "value").

After locating the possession modifier, DASE applies GR1 to resolve the properties of the head noun. In this example, GR1 will return the field name "sh_type" as the property of "field".

If a comment only specifies a partial field name, DASE will resolve the name into a fully qualified name. For example, the comment "Legal values for e_type" specifies a

field name "e_type" without the struct name. DASE maps this field name to structs that contain this field name and generates the fully qualified names, "Elf32_Ehdr→e_type" and "Elf64_Ehdr→e_type".

We use the example shown at the beginning of this section to illustrate how to extract one type of constraints (index-value pairs) in the following steps.

1. For the first sentence, GR2 identifies a prepositional link, "in", between "fields" and "array", and invokes GR1 to resolve "array". GR1 queries the noun modifier for "array" and returns "e_ident". Therefore, it captures the array name as "e_ident".
2. For the second sentence, GR2 identifies a prepositional link, "into", between "indices" and "array", but it does not invoke GR1 because there is no noun modifier. GR3 identifies "indices" as the subject of "macros", and invokes GR1 to resolve "macros". GR1 queries the noun modifier for "macros" and returns "EI_*". Therefore, macros with the name, "EI_*", are treated as the indices of an array.
3. For the third sentence, GR3 is invoked before GR2 because of the structure of the dependency tree. GR3 identifies "values" as the subject of "macros", but it does not invoke GR1 because there is no noun modifier. GR2 identifies a prepositional link, "under", between "macros" and "macro", and invokes GR1 to resolve "macro". GR1 queries the noun modifier for "macro" and returns "EI_*". Therefore, the macro below the macro name, "EI_*", is treated as the value of an array.
4. DASE resolves the array name "e_ident" into a fully qualified name, "Elf32_Ehdr→ e_ident" and generate the two constraints.

Chapter 5

Experimental Methodology

We use the three coverage criteria reported by GNU `gcov`, i.e., line coverage, branch coverage, and call coverage, as our main metrics. Call coverage is the percentage of function calls that are executed throughout the source file. Both `gcov` and these coverage criteria are long-established.

5.1 Evaluated Programs

We evaluate DASE on the following 88 programs from 5 popular and mature software suites, all of which are fundamental tools for Unix-like systems:

Coreutils 6.10. Coreutils is a package of GNU programs that consists of basic file, shell, and text manipulation utilities that are indispensable for Unix-like systems. KLEE authors chose Coreutils as their main test software [15]. We also include it in our evaluation. For a fair comparison with KLEE, we try our best to set the same environment for Coreutils as KLEE's authors. We choose the same version, 6.10, and follow their parameters for both KLEE and DASE. Among the 82 stand-alone programs[1] we tested in Coreutils, the largest program `ls` has 1,475 effective lines of code (ELOC)[2].

`diff` 3.3. `diff` compares files line by line and outputs the differences. It is typically used together with `patch` to pass modifications around. The program has 526 ELOC.

[1] `dd` is excluded because it uses a different option style; `chmod`, `kill`, `mv`, `rm`, and `rmdir` are excluded because they continually cause dangerous test cases to be generated that destroy our experiment data. In the future, we can apply DASE on these programs in a sandbox to address this issue.

[2] All the ELOC counts in this thesis are reported by `gcov` 4.8.1.

`grep` **2.18**. `grep` searches files for given patterns. The program has 932 ELOC.

`objdump` & `readelf`**(b) 2.24**. These two programs are from BINUTILS, which is a set of GNU programs for processing binaries, libraries, object files, and so on. `objdump` and `readelf` are used for displaying the contents of ELF files. They have 1,687 and 6,998 ELOC respectively. Since both BINUTILS and ELFTOOLCHAIN contain a `readelf` program, we use `readelf`(b) to denote the `readelf` program in BINUTILS and `readelf`(e) to denote the one in ELFTOOLCHAIN.

`elfdump` & `readelf`**(e) r2983**. In order to test our ELF model more thoroughly, we select ELFTOOLCHAIN's counterparts for the above two programs. ELFTOOLCHAIN is another set of program development tools for ELF files. It provides similar tools as BINUTILS, but favors well-separated and well-documented libraries. They have 1,539 and 3,571 ELOC respectively.

DASE applies option related constraints to prune the execution paths for COREUTILS, `diff`, and `grep`. All automatically extracted valid options are used as input constraints for path pruning. Since these programs do not process ELF files, we apply DASE on 4 ELF processing programs (`objdump`, `readelf`(b), `elfdump`, and `readelf`(e)) for path pruning. In the future, we would like to have DASE apply both the option related and ELF file format related constraints for path pruning for these programs.

5.2 Experimental Setup

We conduct our experiments on an Intel Core i5-2400 3.10GHz CPU machine running Ubuntu 13.10. KLEE is built from git revision `a45df61` with LLVM 2.9.

We run KLEE and DASE on each program until there are no new instructions covered in a certain amount of time. We run a preliminary experiment to determine this threshold as 15 minutes for COREUTILS programs. For the rest programs with larger size, the threshold is set to 30 minutes.

The other parameters are set by following the instructions from KLEE's authors[3]. The key parameters are:

```
klee PROG -sym-args 0 1 10 -sym-args 0 2 2
          -sym-files 1 8 -sym-stdout
```

[3] http://klee.llvm.org/CoreutilsExperiments.html.

where `PROG` is a program in Coreutils. While for DASE, we keep all the parameters the same as for KLEE, except for replacing a symbolic argument with a list of valid options.

For `diff` and `grep`, we set the symbolic file size to 100 bytes because they are meant to process textual files.

For the ELF processing programs, we use the following parameters respectively for KLEE and DASE:

```
klee -sym-args 0 2 2 -sym-files 1 640
klee -sym-args 0 2 2 -sym-elfs 1 640
```

where `-sym-elfs` holds our ELF model described in Section 4.3.

Chapter 6

Experimental Results

This section demonstrates DASE's ability in improving code coverage and finding previously unknown bugs.

6.1 Code Coverage

Table 6.1 shows the overall code coverage achieved by KLEE and DASE on all the 88 evaluated programs. The 82 Coreutils programs are listed together due to space constraints. Cumulatively, experiments on these programs for both KLEE and DASE take us approximately 186.5 machine hours (almost 8 days).

Table 6.1 shows that DASE outperforms KLEE on the 88 programs: it increases the line coverage, branch coverage, and call coverage by 5.27–22.10%, 5.83–21.25%, and 2.81–21.43% respectively. For example, the coverage boost on `grep` is over 20% for all three coverage metrics. These programs, `readelf`(b), `objdump`, `readelf`(e), and `elfdump`, are difficult to test because their inputs involve complicated ELF format. In addition, the sizes of the evaluated programs are at the same scale as the ones evaluated by previous work [35, 38]. Therefore, the coverage improvement demonstrates the effectiveness of DASE: high level information from documentation regarding input constraints can guide symbolic execution to improve automated test generation.

For Coreutils, which was also evaluated in the KLEE paper [15], the percentages we obtain are different from that paper. This is inevitable because the KLEE tool has evolved significantly since then, including major source code changes of KLEE (e.g., removals of special tweaks), upgrade of LLVM, and an architecture change from 32-bit to 64-bit. We

Program	CC	Count	KLEE(%)	DASE(%)	Δ(%)
Coreutils	L	18326	66.12	75.13	+ 9.00
	B	12674	69.81	76.72	+ 6.90
	C	7003	56.52	66.16	+ 9.64
`diff`	L	526	59.13	73.19	+ 14.06
	B	489	67.28	75.87	+ 8.59
	C	150	48.00	64.00	+ 16.00
`grep`	L	932	37.34	59.44	+ 22.10
	B	786	40.33	61.58	+ 21.25
	C	266	33.46	54.89	+ 21.43
`objdump`	L	1687	19.38	25.55	+ 6.17
	B	1270	16.93	22.76	+ 5.83
	C	463	16.63	19.44	+ 2.81
`readelf`(b)	L	6998	6.89	16.55	+ 9.66
	B	5410	6.19	21.55	+ 15.36
	C	1959	6.89	15.62	+ 8.73
`elfdump`	L	1539	16.11	21.38	+ 5.27
	B	1157	20.40	30.68	+ 10.28
	C	533	16.51	22.33	+ 5.82
`readelf`(e)	L	3571	12.99	29.91	+ 16.92
	B	2550	18.51	35.88	+ 17.37
	C	1126	10.75	25.04	+ 14.29

Table 6.1: Coverage results with KLEE's default search strategy. "CC" means "Coverage Criteria". "L", "B", and "C" stand for "Line coverage", "Branch coverage", and "Call coverage", respectively. The third column, "Count", contains total number of ELOC, branches, calls for each software. Δ(%)" is the improvement of DASE over KLEE.

choose the latest version of KLEE at the time of experiment because (1) it represents the current status of KLEE, and (2) the original version used in the KLEE paper is not publicly available. For a fair comparison, the configurations for KLEE and DASE are identical.

DASE outperforms KLEE for all (82) but 15 programs in Coreutils. Among these 15 programs, the coverage difference is very small (5 or fewer ELOC) for 9. One possible reason is that the options are used to tune parameters for some of the programs (e.g., `csplit`), and the core functionality code is always executed no matter the parameters are tuned or not. In the future, we can improve DASE to automatically understand the semantics of options to avoid using some options for pruning. In addition, 10 of these 15

Program	K-AVGi	D-AVGi	K-MAXi	D-MAXi
COREUTILS	7132 49688	7807 54035	11682 320138	12816 320880
`diff`	18483	23184	35432	47584
`grep`	25942	25698	43424	68504
`objdump`	45915	75519	104479	245370
`readelf`(b)	11570	17095	24884	37377
`elfdump`	13827	25295	24433	50468
`readelf`(e)	18009	24187	29140	50255

Table 6.2: Number of instructions for generated test cases. "K-" stands for KLEE and "D-" stands for DASE. "AVGi" and "MAXi" is the average and maximum number of instructions for the generated test cases respectively. Since COREUTILS includes multiple programs, a range (the minimum and the maximum) is shown.

programs have five or fewer valid options. The benefits of DASE may not show if the input format is very simple (e.g., a few valid options). In the future, we want to improve DASE so that it can automatically detect such relationship between valid options to automatically determine what valid options to use for pruning.

6.1.1 Understanding DASE's Advantage

Since DASE filters out "uninteresting" execution paths, it should enable symbolic execution to go deeper into the execution tree to explore paths. Table 6.2 shows that this is the case for the evaluated programs. We count the number of instructions that are executed for each test case generated by KLEE and DASE, which is approximately the depths of the corresponding execution paths. The average and maximum numbers are shown in Table 6.2.

From Table 6.2, we can see clearly that DASE generates test cases with much more instructions executed, indicating that DASE goes much deeper into the execution tree than KLEE. The improvement is particularly big for the ELF processing programs; both the averages and maximums almost double their counterparts of KLEE. This is expected because while KLEE is still exploring at the early stage of the ELF sanity check, DASE has already quickly penetrated through that part with the help of our ELF model.

To investigate DASE's performance gain in detail, we manually check the coverage difference on `readelf.c` (BINUTILS). For the three functions related to dynamic section,

`*_dynamic_section()`, in which `*` means `get_32bit`, `get_64bit`, or `process`, KLEE fails to cover any of them, while DASE naturally tests them all because our ELF model has a dynamic section. Many other functions, such as `print_symbol()`, are also not covered by KLEE but covered by DASE.

We have similar observations for options. We manually examine the coverage difference for `diff`. In the switch-case structure for argument parsing, only 27 out of the 55 distinct options[1] are explored by KLEE, and the explored options are the top part of the switch-case structure. This result agrees with our analysis in Figure 2.1a. On the other hand, DASE covers 46 options. For example, KLEE fails to cover the option `-X` (to exclude files matching a certain pattern) and the function invoked by this option, `add_exclude_files()`, while DASE tests them.

6.1.2 Combining with Other Search Strategies

Path pruning using input constraints is a general approach which can be combined with different search strategies. To show that the coverage improvement of DASE over KLEE is not tied to KLEE's default search strategy, we change the underlying search strategies for both KLEE and DASE to BFS and rerun our experiments in Table 6.1. Because it is too time-consuming (almost 140 hours) to run all the programs from COREUTILS, we randomly sample 10 from it.

Table 6.3 shows that when the search strategy is BFS, DASE still outperforms KLEE. The improvement for `diff` is big (40.88% line coverage improvement). Comparing Table 6.3 with Table 6.1, we can see that BFS achieves higher coverage than KLEE's default search strategy for COREUTILS, while BFS is less effective for BINUTILS. The result shows that although pruning can help, it is still important to select an effective search strategy for the program under test. Nonetheless, DASE is consistently better than KLEE for the two search strategies and the programs evaluated.

6.1.3 Comparison with Developer Generated Tests

Since automated test generation aims to complement developer generated tests, we evaluate whether DASE improves code coverage on top of developer generated tests. Table 6.4 shows the results. `readelf`(e) is missing because there are no developer generated tests for it in ELFTOOLCHAIN. From this table we can see that by adding DASE generated tests,

[1]For options that invoke the same code segment, we count them as one option.

Program	CC	Count	KLEE(%)	DASE(%)	Δ(%)
Coreutils	L	1840	69.13	79.35	+ 10.22
	B	1285	74.09	79.30	+ 5.21
	C	728	53.85	72.94	+ 19.09
`diff`	L	526	40.87	81.75	+ 40.88
	B	489	60.74	82.82	+ 22.08
	C	150	33.33	76.67	+ 43.34
`grep`	L	932	33.37	63.52	+ 30.15
	B	786	41.35	69.21	+ 27.86
	C	266	25.19	58.65	+ 33.46
`objdump`	L	1687	2.90	3.44	+ 0.54
	B	1270	5.28	5.43	+ 0.15
	C	463	5.18	5.62	+ 0.44
`readelf`(b)	L	6998	0.74	0.81	+ 0.07
	B	5410	1.63	1.63	+ 0.00
	C	1959	1.12	1.17	+ 0.05
`elfdump`	L	1539	17.93	20.08	+ 2.15
	B	1157	21.43	30.68	+ 9.25
	C	533	19.32	21.39	+ 2.07
`readelf`(e)	L	3571	13.58	20.30	+ 6.72
	B	2550	18.43	33.22	+ 14.79
	C	1126	10.30	19.27	+ 8.97

Table 6.3: Coverage results with BFS as the underlying search strategy.

the code coverage is improved. Together with Table 6.1, we can see that for Coreutils, `diff`, `objdump`, and `readelf`(b), DASE alone can generate tests to achieve comparable code coverage as developer generated ones. Although the coverage improvement on `objdump`, `readelf`(b), and `elfdump` is relatively small, the DASE generated tests detected previously unknown bugs for all of them. DASE detected a total of 13 bugs on the evaluated programs that developer generated tests fail to detect (Section 6.2). The results demonstrate that DASE can be used by developers to further improve testing coverage and find more bugs even if manual tests exist.

ZESTI [35] uses developer generated tests as "seeds" and explore similar paths as in the developer generated tests to cover more code and find more bugs. Different from DASE, the effectiveness of ZESTI depends on the quality of developers' tests. For example, ZESTI cannot help programs such as `readelf`(e) which do not have developer generated tests. In

Program	CC	Dev(%)	DASE + Dev(%)	Δ(%)
Coreutils	L	65.87	84.12	+ 18.25
	B	73.16	86.74	+ 13.58
	C	54.19	73.83	+ 19.64
`diff`	L	57.03	80.04	+ 23.01
	B	72.19	83.64	+ 11.45
	C	50.67	72.00	+ 21.33
`grep`	L	81.22	86.37	+ 5.15
	B	86.77	89.82	+ 3.05
	C	73.31	81.20	+ 7.89
`objdump`	L	56.85	61.41	+ 4.56
	B	65.43	66.69	+ 1.26
	C	48.81	53.13	+ 4.32
`readelf`(b)	L	28.45	28.95	+ 0.50
	B	43.35	43.49	+ 0.14
	C	28.38	29.10	+ 0.72
`elfdump`	L	55.37	56.34	+ 0.97
	B	65.91	65.91	+ 0.00
	C	46.62	48.87	+ 2.25

Table 6.4: Coverage results for combining DASE with developers' test cases.

addition, we find that only 41.8% (318 out of 761[2]) of the valid options are covered by the developers' hand-written tests and only for 31.5% of the Coreutils programs, all of their valid options are covered. Therefore, it would be difficult for techniques such as ZESTI to cover all command-line options like DASE does. In fact, due to the higher coverage, DASE detected two previously unknown bugs in Coreutils that were not detected by ZESTI (the same version of Coreutils was evaluated by DASE and ZESTI). More discussion can be found in Chapter 8.

[2]This number is accumulated from all the programs existing in Coreutils, including those not tested by us.

No	Program	Location	Problem	KLEE	DASE
1	readlef(b)	readelf.c:12202	IU		✓
2	objdump	elf-attrs.c:463	IU		✓
3	objdump	elf.c:1351	POB		✓
4	readelf(e)	readelf.c:4015	DBZ		✓
5	readelf(e)	readlef.c:2862	DBZ		✓
6	readelf(e)	readelf.c:3680	DBZ		✓
7	readelf(e)	readelf.c:3930	IU		✓
8	readelf(e)	readelf.c:3961	IL		✓
9	readelf(e)	readelf.c:4102	IL		✓
10	readelf(e)	readelf.c:2662	NPD		✓
11	readelf(e)	readelf.c:2426	POB	✓	
12	elfdump	elfdump.c:1509	POB	✓	✓
13	elfdump	elf_scn.c:87	POB	✓	
14	head	head.c:207	ME		✓
15	split	split.c:333	ME		✓

Table 6.5: New bugs detected by KLEE and DASE. "✓" denotes a bug is found by a tool. "IU" means "Integer Underflow". "DBZ" is "Divide By Zero". "IL" is "Infinite Loop". "NPD" means "NULL Pointer Dereference". "POB" stands for "Pointer Out of Bounds". "ME" is "Memory Exhausted".

6.2 Detected Bugs

DASE finds more bugs than KLEE. KLEE detects 3 bugs[3] from the 88 programs, while DASE is able to uncover 13 bugs. Table 6.5 lists all the bugs. We explain a few example bugs to demonstrate DASE's bug finding capability.

`readelf` from BINUTILS will fail with segmentation fault when the input file contains malformed attribute sections (of type `SHT_GNU_ATTRIBUTES`).The bug exists in the function `process_attributes()`, which is shown in Listing 6.1. In this listing, `p` is a pointer walking through the whole section. At line 12177, 4 bytes are read and interpreted as the length (`section_len`) of the subsequent data structure. Directly after that, the program expects to read a string and assign its length to `namelen`. However, `section_len` can be a number smaller than `namelen + 4`, which causes an integer underflow at line 12202. `section_len`,

[3]Here we do not count the bugs already reported in KLEE's original paper. Those bugs can also be found by DASE.

which becomes an extremely big number after underflow, is later used as the stop condition of a continuing reading of the following memory, which eventually causes a segmentation fault. The fifth random section in our ELF model enables DASE to successfully generate an attribute section to trigger this bug. `readelf`(e) also contains a similar bug.

```
12177   section_len = byte_get(p, 4);
12188   p += 4;
  ...
12200   namelen = strlen((char *)p) + 1;
12201   p += namelen;
12202   section_len -= namelen + 4;
12203
12204   while (section_len > 0)
```

Listing 6.1: Buggy code in `readelf.c` from BINUTILS.

`readelf` from ELFTOOLCHAIN suffers from another segmentation fault because of NULL pointer dereference. The buggy function is `timestamp()`, which is shown in Listing 6.2. `timestamp()` calls C library function `gmtime()` to convert time from type `time_t` to `struct tm`. However, on a 64-bit machine, `time_t` is 64-bit long; its maximum value cannot be properly stored in a `struct tm` because the `tm_year` field in `struct tm` is an `int`. `gmtime()` will return NULL in this case. The code in `timestamp()` fails to check `gmtime()`'s return value against NULL and dereferences it right afterwards, which causes a segmentation fault.

```
2661   t = gmtime(&ti);
2662   snprintf(ts, sizeof(ts), ...
2663        t->tm_year + 1900, ...
```

Listing 6.2: Buggy code in `readelf.c` from ELFTOOLCHAIN.

The `head` program will experience memory exhaustion when invoked with options `-c -1P`,which tells `head` to print all but the last `1P` bytes of the input file. And `P` is a very large unit which stands for 1024^5. Therefore the bug is that `head` tries to allocate a large amount of memory, which exceeds the total amount of available memory. Specifically, `elide_tail_bytes_pipe()` in `head.c` calls `xcalloc()`, which fails. According to the comment, `head` is not expected to "fail (out of memory) when asked to elide a ridiculous amount". For bigger units (e.g., `Z` and `Y`), `head` exits with the correct error message—"number of bytes is so large that it is not representable". Neither developers' hand-written tests nor KLEE generated tests covered this bug. And note that the second argument is not an option; it is a parameter for the `-c` option. Therefore simply enumerating all combinations of options is impossible to detect this bug either.

Two bugs can be found by KLEE but not by DASE. The first one has a very large value in `e_shoff` of the ELF header, which is incompatible with our ELF model. As shown in Section 4.3, we manually fixed `e_shoff` to layout the SHT. So DASE loses the possibility of finding this bug. DASE cannot detect the second bug for a similar reason. Missing these two bugs shows the tradeoff involved in designing the ELF model. We want to focus on those more valid inputs, which are more important and fruitful. Therefore we adopt a relatively rigid layout. As the result shows, this tradeoff is quite beneficial: DASE find 10 more bugs than KLEE. One can always relax the constraints to explore less valid inputs and potentially cover these two bugs. Running KLEE and DASE together to gain benefits from both is also a good solution.

6.3 Constraint Extraction Results

For command-line options, we automatically extracted 821 valid option values: 720 from the 82 Coreutils programs, 46 from `grep`, and 55 from `diff`. Among them, 22 options are invalid: 21 for the `[` utility from Coreutils and 1 for `grep`. The accuracy is 97.3%. The `[` utility only accepts two options, namely `--help` and `--version`; however, it supports expressions such as `-n STRING` to test whether `STRING` is empty and `-d FILE` to test whether `FILE` exists and is a directory. Our command-line option parsing tool will extract the operators `-n` and `-d` as options, although they are conceptually not. However, this brings no harm for testing the `[` utility because these different operators should be covered. The invalid option for `grep` is `-NUM`. `NUM` here is meant to be substituted with some number such as 1. Our extracting tool thinks `-NUM` is a valid option. However, we can easily filter out this invalid option by using the fact that a short option (started with a single dash) only has a single character.

For ELF processing programs, we manually enforced 26 constraints to form the layout of our ELF model shown in Figure 4.1. By analyzing the ELF header file, DASE automatically extracted 56 values for 14 constraints regarding array index-value pairs and 208 values for 21 constraints regarding valid field values. For example, the constraint "`assume( Elf32_Shdr→e_type == 0 | Elf32_Shdr→e_type == 1);`" is one constraint with two values (0 and 1). Among the 208 values for 21 constraints regarding valid field values, 10 values are invalid, which affect six constraints. The accuracy is thus 96.2% for the 21 constraints. The imprecision results from a special kind of macros used in `elf.h`. After listing all valid values of a field, there may be an additional macro, typically named as `*_NUM`, mapping to the next available number. For example, in the following code, `EV_NUM` is not a valid value for the `e_version` field. Our extraction tool will not distinguish this special case and thus treat `EV_NUM` as a legal value.

```
/* Legal values for e_version (version).  */

#define EV_NONE     0       /* Invalid ELF version */
#define EV_CURRENT  1       /* Current version */
#define EV_NUM      2
```

Among all the constraints, 9 constraints (consisting of 60 values) are not used because they are not applicable to our model. These unused constraints are all for special section types. We can incorporate them when we improve our ELF model in the future.

DASE extracted almost all constraints in the header file, which helped DASE improve the testing coverage and testing effectiveness. To extract more constraints to further improve the testing effectiveness, we can analyze the ELF specification and the man page. Although they contain outdated and incomplete information, we may manually verify the extracted constraints (currently the constraints extracted from the header file are used directly by DASE for test generation to minimize manual effort) or leverage time information to automatically identify potentially outdated constraints. In addition, the proposed NLP techniques can be generalized to analyze other formats, e.g., TCP/IP packets and XML format, etc.

Chapter 7

Discussion And Future Work

Using information from documents, DASE aims to help symbolic execution focus on semantically important paths. We have experimented with alternative design choices, which include distributing execution time equally among command-line options and making the ELF model symbolic. These alternative design choices are described below.

7.1 Previous Attempts for Command-line Options

In our first try, time spent on each option is predetermined. If a program has n options, we then divide the total time into n time slices and allocate one slice to each option. This approach guarantees the equality of options. However, it may perform poorly because the importance and code coverage of different options are not identical. For example, in `rm`, `--help` and `--version` are trivial compared to `-r`. It is not easy to devise a rational time allocation scheme.

We also try to represent all possible options of a program using a single constraint. For example, to test `echo` with m symbolic arguments (using KLEE's terminology, this is `echo --sym-args 0 m x`, in which `x` means the length of each symbolic argument), we single one symbolic argument out and put the following constraint on it:

```
assume(arg == "-n"     || arg == "-e"     || arg == "-E"     ||
       arg == "--help" || arg == "--version");
```

where `arg` is the selected symbolic argument. The rest $m-1$ arguments are kept untouched (again using KLEE's terminology, this is `echo arg --sym-args 0 m-1 x`). In this way,

there will be no concretization of the selected argument. Also, there is no explicit execution branch forking as described in Section 4.2. Experiments demonstrate that this method almost contributes no coverage improvement to symbolic execution. One reason is that the constraint, only for a single variable, is too simple. Another explanation is that the constraint gives little guidance to symbolic execution. Practically, symbolic execution can build this constraint by walking through a program's input argument parsing module.

Therefore we propose to use the method in Section 4.2 to improve symbolic execution for command-line programs. It combines equality of each option at the beginning and flexibility of effort allocation for further exploration.

7.2 Previous Attempts for ELF

We have tried a more flexible version of our ELF model. In the current model, the size and offset of each section are set to concrete values. In the flexible model, they are symbolic. That is, the size and offset of each section will be determined by symbolic execution when exploring ELF processing programs. Although more general, it loses the benefit of concretizing ELF's "skeleton." If the offset of a section is symbolic, all indexes and pointers to its contents become symbolic. When referring fields in the section, KLEE needs first to resolve the symbolic pointer. As explained in the KLEE paper [15], this is an expensive operation. Therefore, the performance gain is much smaller than using rigid ELF skeleton.

7.3 Future Work

Our current approach of using information from documentation to improve symbolic execution is efficient. However, our approach can be improved in the following ways, which remain as our future work.

- Analyzing the importance of options. As explained in Section 6.1, DASE's performance on some command-line programs is not satisfying now. This is because that some programs have few options, or their options are just used to tune parameters while the core functionality code is always executed no matter the parameters are tuned or not. Under such circumstances, it is not suitable to use options for pruning.
- Analyzing the relationship between options. Command-line options can overlap with each other. For example, `readelf` has an option `-a` which is equivalent to the combination of `-h`, `-l` `-S`, `-s`, `-r`, `-d`, `-V`, `-A`, and `-I`. It will be more beneficial to avoid pruning using both the "super" option and its equivalent options.

- Adding more sections in the ELF model. Currently, we only force five sections in our ELF model, while only three of them have types predetermined. These three types of sections are important. Nonetheless, it is meaningful to try more different types.
- Building models for other file formats. ELF is prevalent in Unix-like systems. However, there exist plenty of other important file formats. Various network package protocols are crucial for the current computer world. XML as the *de facto* standard of message exchanging is also widely used and its parsers should function correctly to their best. A model for a file type can benefit vast numbers of programs.

Chapter 8

Related Work

Recent years have witnessed re-emerging interest in utilizing symbolic execution [20, 29] for automated test generation. Researchers have proposed frameworks [15, 16, 18, 25, 37, 41, 42, 47] for different programming languages by using symbolic execution alone or combining it with concrete execution. The later executes a program on a concrete input, and then flips the branch conditions to systematically explore more paths. Although symbolic execution already shows its excellence in improving test efficiency, its scalability is hindered by path explosion.

To alleviate the path explosion problem, a number of strategies have been proposed [10, 14, 15, 19, 30, 40, 42]. Simple strategies such as depth-first search (DFS) favors deep paths in the execution tree whereas breadth-first search (BFS) favors shallow paths. CUTE [42] and CREST [14] both use a bounded DFS strategy. It limits the number of branches that can be explored at any path but it does not scale to programs that have a deep search tree. KLEE [15] uses two atom search strategies in a round robin fashion as explained in Chapter 2. CREST [14] also proposed control-flow graph (CFG) directed search, uniform random search, and random branch search. Santelices and Harrold proposed Symbolic Program Decomposition [40] that exploits the control and data dependencies to avoid analyzing unnecessary combinations of subpaths. Babić *et. al.* proposed a three-stage process [10] which exploits static analysis of the binaries under test to guide exploration. Krishnamoorthy *et. al.* [30] proposed a reachability-guided strategy that guides the search towards important parts of the code specified by the user; a conflict-driven backtracking strategy that utilizes conflict analysis to discard redundant paths; and error-directed strategies that target on assertions, abort statements, and paths that are less likely to be executed.

DASE tackles the path explosion problem by automatically extracting input constraints from documentation and use the input constraints to prune execution paths. This is different from the previous techniques, which rely on information from the code logic (typically CFG) to guide the path exploration process. In addition, DASE focuses on valid or close-to-valid inputs, while the above techniques have no knowledge about whether an execution path corresponds to valid or invalid input.

BuzzFuzz [23] aims to test a program's core functionality code like DASE. It focuses on potential attack points, i.e., locations in code possibly resulting in errors, in the program under test. Dynamic taint tracing is used to identify the initial values affecting the attack points from a set of seed input files. These initial values are then fuzzed to generate new input files, which typically preserve the underlying syntactic structure of the seed input files. Thus, BuzzFuzz can easily pass the initial input parsing part and exercise core functionality code. It is clear that BuzzFuzz relies on existing seed input files; paths explored by BuzzFuzz are limited to around those appear in seed input files. Moreover, users need to specify attack points. The default set of attack points only focuses on library function arguments. In contrast, DASE is not a fuzzing technique. It does not depend on seed input files, and its bug finding is not limited to certain program points.

ZESTI [35] explores "interesting" code by starting with developer generated test to explore additional paths around predefined sensitivity operations—pointer dereferences and divisions—using symbolic execution. This means ZESTI's performance is affected by existing tests, while DASE does not suffer from this problem, as discussed in Section 6.1, In addition to ZESTI, other work utilizing existing tests for *test suite augmentation* includes directed test suite augmentation [49]. KATCH [38] uses existing tests to explore the added or modified code in software patches. It first locates a developer test case that has an execution path *close* to the patch code, and then modifies the execution path to test the path code. Kin-Keung *et. al.*'s work [32] also aims at finding execution paths that reach target code. They all rely on existing tests and use information from the code logic.

Input constraints and specifications in general have been used for automated test generation [12, 13, 36]. These techniques do not use the input constraints to guide symbolic execution for test generation. In addition, the input constraints need to be provided, and they do not automatically extract input constraints. CESE [33] and grammar-based whitebox fuzzing [24] are related to our work with respect to testing programs expecting highly-structured inputs. But their methods require context-free grammar and they focus on completely valid inputs.

Partitioning has been used to improve symbolic execution. `FlowTest` [34] partitions the inputs into "non-interfering" blocks by analyzing the dependency among inputs. Simple

Static Partitioning [43] approaches the problem by first performing a shallow symbolic execution up to a certain depth and collecting all the path constraints. Then these path constraints are broken down into individual ones and recombined into a reduced number of constraints for parallel execution. Qi *et. al.* proposed execution paths can be partitioned using their output similarity [39]. However, these techniques still only leverage information from the code logic, while DASE "divides" the execution based on semantic functionalities of the program. Although this resembles input space partitioning [48], DASE's division is not disjoint since we have several symbolic arguments and currently we only "divide" the execution according to the first one. The other unconstrained options can expand to whatever concrete options and introduce duplicates. Ensuring disjointness remains as our future work.

A recent study [28] had shown that there is little correlation between code coverage and the effectiveness of a test suite (measured by the number of killed mutants). However, the mutants are generated using PIT [5] with extremely simple mutants, which might not represent real bugs. In our study, we built DASE on top of KLEE. DASE significantly improves the code coverage and detect new bugs compared to KLEE through partitioning the input space by command-line options.

Chapter 9

Conclusions

This thesis presents *Document-Assisted Symbolic Execution* (*DASE*) as a general and effective approach to improve symbolic execution for automatic test generation and bug detection. DASE utilizes natural language processing techniques and heuristics to automatically extract high level input constraints from programs' documentation. DASE then uses these constraints as pruning criteria to focuse on valid or almost valid inputs. DASE prunes paths based on their *semantic* importance to help search strategies prioritize execution paths more effectively.

We build our DASE prototype based on KLEE and focus on two types of input constraints: command-line options and the ELF file format. These two types are sufficient for a wide spectrum of programs. We evaluate DASE on 88 programs from 5 mature real-world software suites: GNU Coreutils, GNU findutils, GNU grep, GNU Binutils, and elftoolchain. Compared to KLEE, DASE increases line coverage, branch coverage, call coverage by 5.27–22.10%, 5.83–21.25%, 2.81–21.43%, respectively. Additionally, DASE detected 13 previously unknown bugs, 6 of which have been confirmed by the developers. DASE's ability in improving code coverage and detecting more bugs clearly shows the benefits of incorporating high level information from documentation to symbolic execution.

There are many possible extensions for this work. We can further analyze the usefulness of options for pruning. We can also extract constraints and models for other types of formats, e.g., network packages and XML files, to improve test generation for programs that take these types of input.

APPENDICES

Appendix A

Bug Reports

Table A.1 lists all the bug reports in the bug tracking systems. Bugs are numbered the same as in Table 6.5. Bug #1 and #2 are pointing to the same report because the developers fixed the two issues in the same thread. However, they occurred in two distinct source files and affected two different programs, thus, they are different bugs.

No	Bug Report
1	https://sourceware.org/bugzilla/show_bug.cgi?id=16664
2	https://sourceware.org/bugzilla/show_bug.cgi?id=16664
3	https://sourceware.org/bugzilla/show_bug.cgi?id=16682
4	https://sourceforge.net/p/elftoolchain/tickets/439
5	https://sourceforge.net/p/elftoolchain/tickets/444
6	https://sourceforge.net/p/elftoolchain/tickets/445
7	https://sourceforge.net/p/elftoolchain/tickets/438
8	https://sourceforge.net/p/elftoolchain/tickets/440
9	https://sourceforge.net/p/elftoolchain/tickets/442
10	https://sourceforge.net/p/elftoolchain/tickets/441
11	https://sourceforge.net/p/elftoolchain/tickets/443
12	https://sourceforge.net/p/elftoolchain/tickets/446
13	https://sourceforge.net/p/elftoolchain/tickets/447
14	http://osdir.com/ml/bug-coreutils-gnu/2013-01/msg00137.html
15	https://lists.gnu.org/archive/html/bug-coreutils/2013-01/msg00148.html

Table A.1: Bug reports for all detected bugs.

References

[1] Executable and linkable format. http://www.skyfree.org/linux/references/ELF_Format.pdf.

[2] Linux programmer's manual - elf. http://man7.org/linux/man-pages/man5/elf.5.html.

[3] The llvm compiler infrastructure. http://llvm.org.

[4] Parsing program options using getopt. http://www.gnu.org/software/libc/manual/html_node/Getopt.html.

[5] Pit mutation operators. http://pitest.org/quickstart/mutators/.

[6] Posix utility conventions. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html.

[7] Simple theorem prover. http://stp.github.io/stp/.

[8] Standards for command line interfaces. http://www.gnu.org/prep/standards/html_node/Command_002dLine-Interfaces.html.

[9] Andrea Avancini and Mariano Ceccato. Comparison and integration of genetic algorithms and dynamic symbolic execution for security testing of cross-site scripting vulnerabilities. *Inf. Softw. Technol.*, 55(12):2209–2222, December 2013.

[10] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 12–22, 2011.

[11] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering, 2007. FOSE '07*, pages 85 –103, may 2007.

[12] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. *SIGSOFT Softw. Eng. Notes*, 27(4), July 2002.

[13] Achim D Brucker, Matthias P Krieger, Delphine Longuet, and Burkhart Wolff. A specification-based test case generation method for uml/ocl. In *Models in Software Engineering*, pages 334–348. Springer, 2011.

[14] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 443 –446, sept. 2008.

[15] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 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. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, 2006.

[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 *Proceedings of the 33rd International Conference on Software Engineering*, pages 1066–1071. ACM, 2011.

[18] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, pages 265–278, 2011.

[19] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. Mace: model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of the 20th USENIX conference on Security*, pages 10–10, 2011.

[20] L.A. Clarke. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, SE-2(3):215 – 222, sept. 1976.

[21] Marie-Catherine de Marneffe and Christopher D. Manning. *Stanford typed dependencies manual*. 2013.

[22] M.K. Ganai, N. Arora, Chao Wang, A. Gupta, and G. Balakrishnan. Best: A symbolic testing tool for predicting multi-threaded program failures. In *Automated Software*

Engineering (ASE), 2011 26th IEEE/ACM International Conference on, pages 596–599, Nov 2011.

[23] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 474–484. IEEE, 2009.

[24] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 206–215, 2008.

[25] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[26] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[27] Google. Googlecl brings google services to the command line. https://code.google.com/p/googlecl/.

[28] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the International Conference on Software Engineering*, 2014.

[29] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[30] Saparya Krishnamoorthy, MichaelS. Hsiao, and Loganathan Lingappan. Strategies for scalable symbolic execution-driven test generation for programs. *Science China Information Sciences*, 54(9):1797–1812, 2011.

[31] Kaituo Li, Christoph Reichenbach, Yannis Smaragdakis, Yanlei Diao, and Christoph Csallner. Sedge: Symbolic example data generation for dataflow programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 235–245. IEEE, 2013.

[32] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *Static Analysis*, pages 95–111. Springer, 2011.

[33] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 134–143, New York, NY, USA, 2007. ACM.

[34] Rupak Majumdar and Ru-Gang Xu. Reducing test inputs using information partitions. In *Proceedings of the 21st International Conference on Computer Aided Verification*, pages 555–569, 2009.

[35] Paul Dan Marinescu and Cristian Cadar. make test-zesti: a symbolic execution solution for improving regression testing. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 716–726, 2012.

[36] Jeff Offutt and Aynur Abdurazik. Generating tests from uml specifications. In *Proceedings of the 2Nd International Conference on The Unified Modeling Language: Beyond the Standard*, UML'99, 1999.

[37] Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[38] Cristian Cadar Paul Dan Marinescu. Katch: High-coverage testing of software patches. In *European Software Engineering Conference / ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 235–245, 8 2013.

[39] Dawei Qi, Hoang D.T. Nguyen, and Abhik Roychoudhury. Path exploration based on symbolic output. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 278–288, 2011.

[40] Raul Santelices and Mary Jean Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 195–206, 2010.

[41] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 513–528. IEEE, 2010.

[42] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly*

with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pages 263–272, 2005.

[43] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 183–194. ACM, 2010.

[44] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* iComment: Bugs or bad comments? */. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.

[45] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, (7007.011), 2002.

[46] Suresh Thummalapenta, K. Vasanta Lakshmi, Saurabh Sinha, Nishant Sinha, and Satish Chandra. Guided test generation for web applications. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 162–171, Piscataway, NJ, USA, 2013. IEEE Press.

[47] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. In *Tests and Proofs*, pages 134–153. Springer, 2008.

[48] E.J. Weyuker and T.J. Ostrand. Theories of program testing and the application of revealing subdomains. *Software Engineering, IEEE Transactions on*, SE-6(3):236 – 246, may 1980.

[49] Zhihong Xu. Directed test suite augmentation. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1110–1113, New York, NY, USA, 2011. ACM.

[50] P. Zhang, S. Elbaum, and M.B. Dwyer. Automatic generation of load tests. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 43–52, Nov 2011.