Document Assisted Symbolic Execution (DASE)

Paper: *Document- Assisted Symbolic Execution for Improving Automated Software Testing*

Author: *Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan*
Electrical and Computer Engineering, University of Waterloo, Canada

- Sandeep Jala, Fathima Shajahan, Pegah Taheri

INTRODUCTION:

- Symbolic execution has been leveraged to automatically generate high code coverage test suites to detect bugs. The input is symbolic values
- Once the execution of path terminates, the constraints are used to generate concrete inputs to exercise the path.

PROS	CONS
Generates high code coverage test suites to detect bugs.	SE suffers from the fundamental problem of path explosion
Improves testing effectiveness	Does not analyze documents automatically and requires constraints to be given

Table 1. Pros and Cons of SE

KLEE BACKGROUND:

- KLEE is SE engine based on LLVM

  i) LLVM bytecode to be interpreted by KLEE

  ii) KLEE checks for dangerous operations that cause failure

  iii) 2 default search strategies:

  - Coverage-Optimized Search-

    - Uses heuristics to choose a state that is most likely to cover a new code in the immediate future

  - Random Path Selection

    - Randomly choose a branch to follow at a branch point

- DASE automatically extracts input constraints from document: Uses constraint as a filter to favor execution paths that text the core functionalities
- As a path pruning strategy, it can be used to improve SE
- 2 Categories of DASE-
    i) Format of input file
    ii) Valid values of command prompt
- DASE was used on 88 programs from 5 widely-used software suites
- DASE detected 12 previously unknown bugs that KLEE failed to detect
    - 6 of which have already been confirmed by the developers
- Compared to KLEE, DASE increases line coverage, branch coverage

Design and implementation

- Automatically extracting input constraints from code comments, man page and header files using NLP and regex
  - Dase uses 4 grammar rules to find relevant comments and notes as the main rule.

```
/* 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'
```

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

Figure 1.

- Adding file layout constraints for ELF files.
  - ELF files' specific layout: kept incomplete -> DASE has a chance to explore close-to-valid inputs and all of the boundary case inputs

Design and implementation

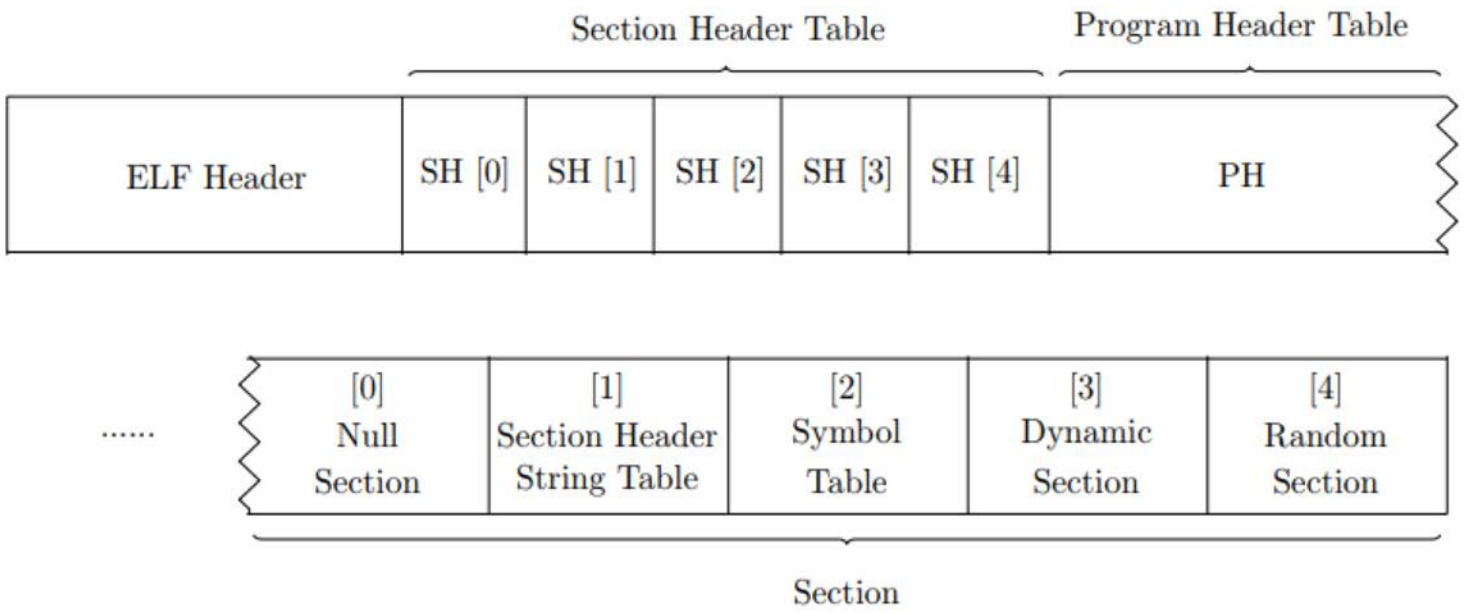


Figure 2. DASE's ELF layout. SH is Section Header, and PH is Program Header. Numbers in brackets are array indices.

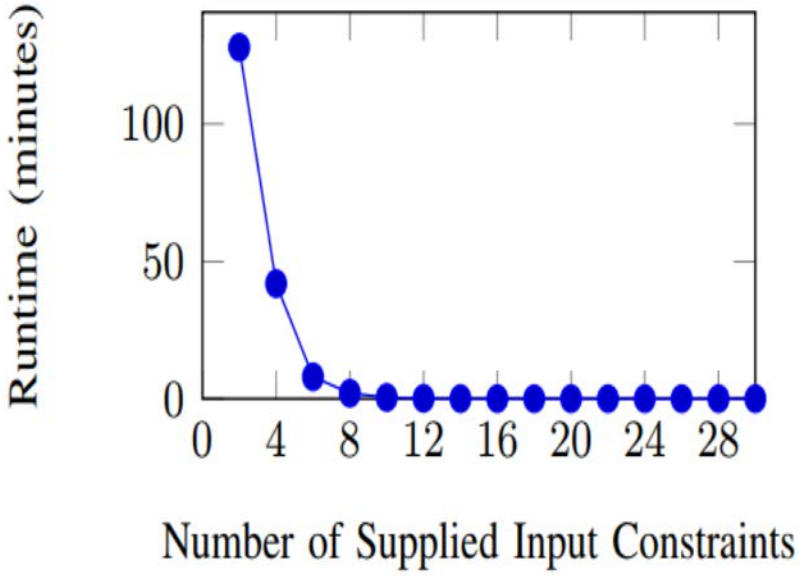
Design and implementation cont'd.

- Automatically extracting valid options from man pages: parsing the standard list of valid options
  - Simple regular expression matching.
  - Input: man page, Output: command line options
    - Two kind of regular expressions: "short" and "long"
    - Example: Short: -s, Long: --this is a long one
- Flattening Symbolic Execution using those options.
  - Options are used to trim and reorganize the tree
  - n branches are created for n valid options
  - Branches are prioritized
  - Goal: to balance the testing effort on each option

Why can input constraints help SE find more bugs and test more code?

- There can be a large number of branches within the entire program
- DASE focuses on constraints to test the path and focus on certain branches
- DASE found bugs in 0.1s but KLEE did not find them even after 10 hours

Figure 3.
Constraints vs. Runtime



How to flatten symbolic execution to find more bugs and test more code?

- DASE extracts valid options by analyzing programs' documentation and use them as input constraints

i) DASE found 11 valid values out of 256^m options for *rm* in cmd prompt

ii) All possibilities are treated evenly for improved test coverage

iii)Some bug might take more time to be found

- finding more bugs is more important than small time loss

Example:

```
1   int counter = 0;
2   for (int i = 0; i < 30; i++) {
3       if (input[i] == 'A') {
4           counter++;
5           foo();
6       }
7   }
8   if (counter == 30) {
9       process_boundary_cases(); // bug!
10      if (input[30] == 'B')
11          process_valid_input(); // bug!
12  }
```

Figure 4. Coding example

Table 2.

DASE: PROS AND CONS

PROS	CONS
Saves manual work for practitioners	Drawing constraints from documents is challenging
Allows for easier use of SE techniques	
Uses SE to identify the semantic importance of different execution paths to focus on core functionalities	
Find more bugs on KLEE	

Table 3.

DASE vs KLEE

DASE	KLEE
Uses input constraints from documents.	Does not use input constraints.
Covers more functionalities	
Finds higher number of bugs	
Explores deeper	

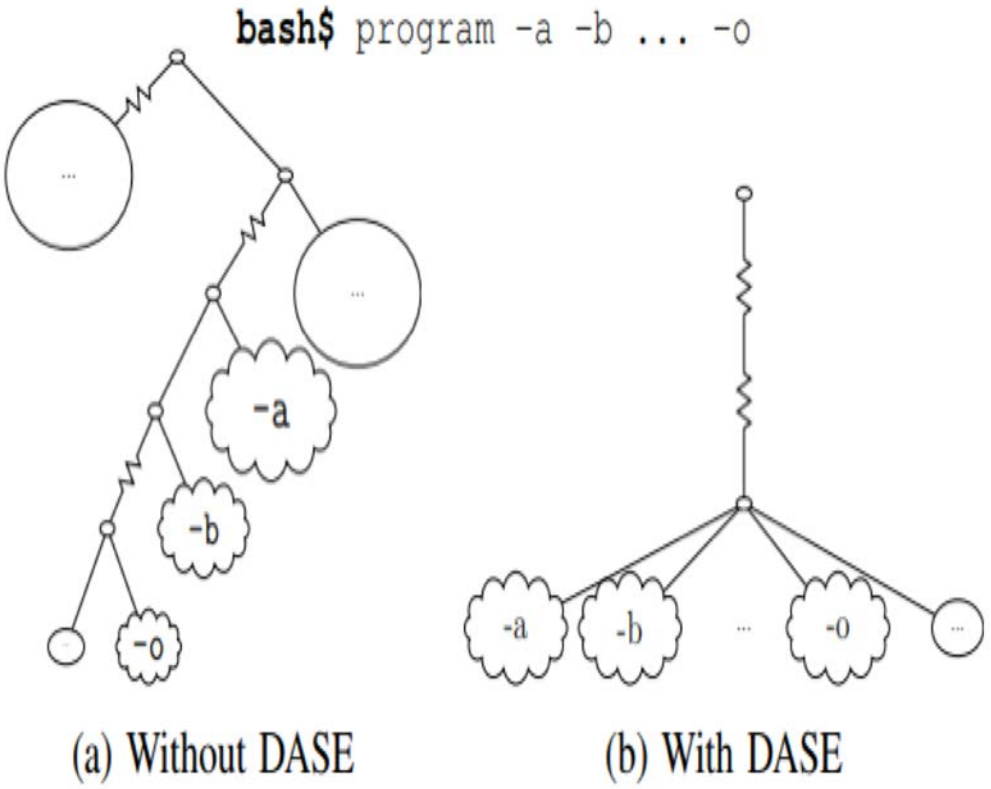


Figure 5. Abstract view of execution trees for command-line options. Clouds are execution subtrees related to valid command-line options. Ovals are other execution subtrees. Deep options such as -o are more likely to be tested with DASE.

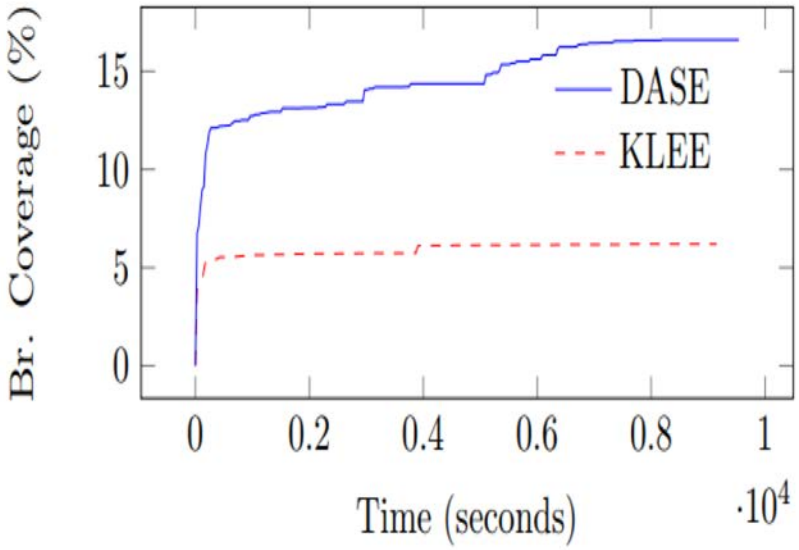


Figure 6. Branch Coverage on `readelf(b)` over time

NOTE: 3b is very different from BFS

Results:

- DASE found 5 previously unknown bugs in COREUTILS and BINUTILS that were already processed by other SE tools
- 2 bugs were found by KLEE but not DASE
- DASE generated test cases with more instructions executed = explored deeper in instruction tree
- In the *diff* program, KLEE only covers 27 of the 55 distinct options while DASE explores 46 of the options
- KLEE's and DASE's search strategy were changed to BFS and DASE still performed better
- DASE found 13 bugs that developer written tests did not find
- DASE extracted input constraints from manual pages and code comments with 97.8-100%
- All this was compared to an updated KLEE model, which was still outperformed by DASE

Results:

TABLE 4: 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."

No	Program	Location	Problem	KLEE	DASE
1	readelf(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		✓

Discussion Question 1:

What kind of documentation can be the best sources of information regarding input constraints? What are easier to read?

Discussion Question 2:

Can we use DASE to extract information and constraints from use-case and other diagrams?

Discussion Question 3:

What are your thoughts on the use of regex for manual pages and NLP for comments? Can this be reversed?

Discussion Question 4:

Will adding API documentation improve constraint extraction?

Discussion Question 5:

What about cases where the input is not specific enough? Such as methods that work with any length of string? How can DASE still be a good choice there?