DASE: Document-Assisted Symbolic Execution for Improving Automated Software Testing

Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan
Electrical and Computer Engineering, University of Waterloo, Canada
{e32wong, lei.zhang, song.wang, t67liu, lintan}@uwaterloo.ca

***Abstract*—We propose and implement a new approach, *Document-Assisted Symbolic Execution* (*DASE*), to improve automated test generation and bug detection. DASE leverages natural language processing techniques and heuristics to analyze program documentation to extract input constraints automatically. DASE then uses the input constraints to guide symbolic execution to focus on inputs that are semantically more important.**

We evaluated DASE on 88 programs from 5 mature real-world software suites: COREUTILS, FINDUTILS, GREP, BINUTILS, and ELFTOOLCHAIN. DASE detected 12 previously unknown bugs that symbolic execution without input constraints failed to detect, 6 of which have already been confirmed by the developers. In addition, DASE increases line coverage, branch coverage, and call coverage by 14.2–120.3%, 2.3–167.7%, and 16.9–135.2% respectively, which are 6.0–21.1 percentage points (pp), 1.6–18.9 pp, and 2.8–20.1 pp increases. The accuracies of input constraint extraction are 97.8–100%.

I. INTRODUCTION

Software testing is an essential part of software development. Many automated test generation techniques are proposed and used to improve testing effectiveness and efficiency [1]–[6].

Symbolic execution [7], [8] has been leveraged to automatically generate high code coverage test suites to detect bugs [9]–[15]. 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 the corresponding constraints are added to each path. Once the execution of a path terminates, the collection of constraints along that execution path is 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 [1], [5], [14], [16], [17].

Although symbolic execution has been successful in improving testing effectiveness, existing techniques do not take full advantage of programs' input constraints expressed in documents. Valid program inputs typically need to follow certain constraints. For example, `rm` (version 6.10) only accepts 11 options including `-r` and `-f`, and `readelf` requires its input files to follow Executable and Linkable Format (ELF). Focusing on the valid and close-to-valid inputs can help test the core functionalities of the program, which should improve testing coverage and effectiveness as shown by previous techniques [18], [19]. It 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. Fortunately, information about input constraints commonly exists in software documents, such as programs' manual pages (e.g., the output of `man rm`) and the comments of header files (e.g., `elf.h`).

Thus, we propose a general approach, *Document-Assisted Symbolic Execution* (*DASE*), to enhance the effectiveness of symbolic execution for automatic test generation and bug detection. DASE *automatically* extracts input constraints from documents, and uses these constraints as a "filter" to favor execution paths that execute the core functionalities of the program. DASE, as a path pruning strategy, can be used on top of existing search strategies to further improve symbolic execution (§VI shows that DASE can find more bugs and improve testing coverage on top of different search strategies).

Testing with invalid inputs can also be important, e.g., to check error-handling code or find defects due to malformed inputs [20], [21]. However, this paper focuses on testing with valid inputs as it allows symbolic execution to exercise deeper into program logic rather than focus on input parsing. If developers believe it is more important to test some programs with invalid inputs, they can use the DASE approach to focus on invalid inputs by negating the input constraints, which we would like to evaluate in the future. Regardless of what inputs (valid or invalid) to focus on, one cannot do so without knowing what inputs are valid and what are not. DASE *enables this choice automatically by extracting input constraints from program documents automatically*.

This automation is novel because existing symbolic execution techniques [18], [19] do not analyze documents automatically and require input constraints to be given. Previous work has shown that constraint extraction from documentation [22], [23] is important yet challenging. Since this automation can reduce manual effort, DASE could make it easier for practitioners to adopt these symbolic execution techniques [18], [19] and other techniques that require input constraints [3], [24], [25] such as constraint verification.

DASE considers two categories of input constraints: the format of an input file (e.g., ELF and tar), and valid values of a command-line option (e.g., `-r` for `rm`). These two types are sufficient for a wide spectrum of programs. This paper makes the following contributions:

- We propose a novel approach, *DASE*, to improve automated test generation. By leveraging input constraints automatically extracted from documents, DASE enables symbolic

Accepted for publication by IEEE. © 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

execution to automatically distinguish the *semantic* importance of different execution paths to focus on programs' core functionalities to find more bugs and test more code.

- We propose a new technique that combines natural language processing (NLP) techniques, i.e., grammar relationships and heuristics, to automatically extract input constraints from documents. The technique is general and should be able to extract input constraints for purposes other than symbolic execution such as program comprehension and constraint verification. We study two types of documents, i.e., manual pages and code comments, and extract input constraints from both.
- Our evaluation shows that DASE finds more bugs and has higher code coverage than KLEE [4] (a symbolic execution tool without input constraints from documents). We evaluated DASE on 88 programs from 5 widely-used software suites—GNU COREUTILS, GNU FINDUTILS, GNU GREP, GNU BINUTILS, and ELFTOOLCHAIN, most of which have been thoroughly tested by many symbolic execution tools [4], [9], [14], [26]. DASE detected 12 previously unknown bugs[1] that KLEE failed to detect, 6 of which have already been confirmed by the developers, while the rest await confirmation. Compared to KLEE, DASE increases line coverage, branch coverage, and call coverage by 14.2–120.3%, 2.3–167.7%, and 16.9–135.2% respectively, which are 6.0–21.1 percentage points (pp), 1.6–18.9 pp, and 2.8–20.1 pp increases. The input constraint extraction of three files formats—ELF, tar, and the Common Object File Format (COFF)—has accuracies of 97.8–100%.

II. OVERVIEW

A real-world program typically contains numerous or even infinite number of execution paths. Given limited time, it is crucial for testing to prioritize the paths effectively. Researchers have proposed approaches to guide the path exploration of symbolic execution [1], [5], [14], [16], [17] to find more bugs and improve code coverage.

Path pruning, which applies a "filter" to prune "uninteresting" paths before employing a search strategy, can 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 both valid and close-to-valid inputs (e.g., boundary cases) to explore deeper in a program's core functionality. 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 inputs. The constraints can be slightly relaxed (e.g., relaxing a constraint "x must be between 0 to 10 (inclusive)" to "x must be between -1 and 10 (inclusive)") to exercise paths corresponding to close-to-valid inputs to test boundary cases.

[1] We do not count bugs that are already reported in the KLEE paper. Those bugs, which can also be found by DASE, are not counted as newly detected ones by DASE either.

```
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 }
```

Fig. 1: Motivating Example

These paths (for valid and close-to-valid inputs) can pass the trivial part of input sanity check to go deeper and are more likely to uncover bugs [18], [19] for two main reasons. First, keeping invalid inputs in the search space hurts the effectiveness of symbolic execution based test generation. The reason is that exploring invalid inputs takes up time and memory, which can be used for testing valid and close-to-valid inputs instead. Second, 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.

Next we (1) illustrate why input constraints can help symbolic execution find more bugs and improve testing coverage and (2) summarize how DASE extracts these types of constraints automatically from two sources of documents.

Why can input constraints help symbolic execution find more bugs and test more code? We will use the code snippet in Figure 1 to answer this question, while real code from BINUTILS is shown later in Figure 5 to explain how the automatically extracted input constraints help DASE detect previously unknown bugs and improve coverage. The code snippet in Figure 1 has 32 branches (30 from line 2 and 3, one from lines 8 and one from line 10), indicating 2^{32} possible paths to explore. Without knowing which paths execute the core logic, it is hard to expose the bug deep in line 11 because only 1 out of the 2^{32} paths leads to that line. DASE automatically extracts constraints from documents and find that the first 30 characters of a valid input must be 'A', and the next character must be 'B'. These constraints will guide the execution to line 11. If the document is incomplete, e.g., only mentioning that the first 30 characters of a valid input must be 'A', we can still hit the bug in line 9 that is triggered by close-to-valid inputs. In addition, it increases the chance to detect the bug in line 11 ($\frac{1}{2^{32}}$ to $\frac{1}{2}$). In either case, DASE can cover more code (lines 9–10 and possibly 11), which is hard for standard symbolic execution to cover, in addition to detecting more bugs.

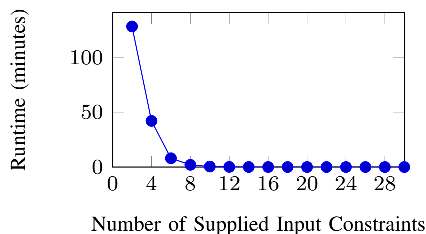

Fig. 2: The runtime to find the bug in line 11 decreases exponentially as we supply more input constraints. The runtime when no constraint is supplied is not depicted because the bug was not detected after 10 hours when we stopped the execution.

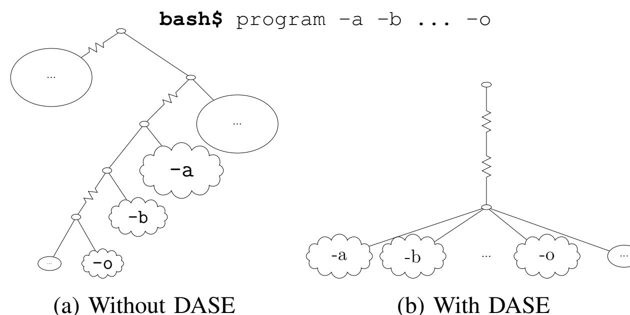

Fig. 3: Abstract view of execution trees for command-line options. Clouds are execution subtrees related to valid command-line options. Ovals are other execution subtrees. Deep options such as `-o` are more likely to be tested with DASE.

We run KLEE on this example for 10 hours, and KLEE detects neither of the bugs. In contrast, DASE detects both bugs in 0.1 seconds. In practice, one may not have all constraints to define the entire input. In order to understand the effect of the number of constraints, we plot how the time to discover the bug in line 11 changes as the number of given constraints changes in Figure 2. The runtime to find the bug decreases exponentially as we supply more input constraints, suggesting that input constraints can dramatically improve the efficiency of finding bugs, i.e., finding more bugs given the same amount of time.

How to flatten symbolic execution to find more bugs and test more code? Command-line options are a special type of input. Therefore, we propose a new way to leverage their constraints to improve testing effectiveness. Command-line options are used to invoke certain functionalities of programs or tune parameters. For example, the option `-r` tells `rm` to perform a recursive deletion. A program typically uses nested if-else statements or a switch-case statement to check the input argument against all valid options until it finds a match, and then invokes the corresponding functionality.

DASE extracts valid options by analyzing programs' documentation and use them as input constraints. For example, DASE finds that among the 256^m possibilities[2] for `rm` (m is the maximum number of characters allowed in an option), only 11 values are valid options. With n valid options ($n = 11$ in the example above), DASE "forks" the execution state n times, with each child execution state taking a valid option[3]. In this way, DASE creates n execution branches for a program with n valid options (one for each valid option).

The concretization moves all valid options at the same depth of the execution tree, indicating that all valid options are treated equally (Figure 3b). Figure 3a illustrates the dynamic execution tree of symbolic execution without DASE. Clouds are subtrees related to valid command-line options. If a program has 15 valid options `a`–`o`, and `o` is the deepest valid option as shown in Figure 3a, the time spent on testing code related to option `-o` could be $\frac{1}{2^{15}}$ of the total testing time without DASE[4]. The reason is there are 15 branches from the nested if–else or switch-case statements, one for each valid option. Such a small fraction often means option `-o` would not be tested at all in practice. With DASE, the time of testing option `-o` would be much longer—about $\frac{1}{15}$ as in Figure 3b. This way, a bug in the code that processes option `-o` will be more likely to be exposed with DASE. On the other hand, the probability of hitting a bug in a shallower option (e.g., `b`) would be reduced from $\frac{1}{2^2}$ to $\frac{1}{15}$, but the difference is much smaller, and it is still highly likely that the option `b` will be tested given that the probability is $\frac{1}{15}$. In addition to finding more bugs, since each option has about $\frac{1}{15}$ chance to be explored, more options are likely to be tested, improving testing coverage (§ VI-B shows that DASE covers more options than KLEE).

Although this approach may appear to be similar to breadth-first search (BFS), it is very different from BFS. Without DASE, BFS would explore paths in Figure 3a, which would still waste time on shallow paths and are less likely to explore deeper paths. In fact, our evaluation shows that DASE outperforms KLEE even if BFS is used as the underlying search strategy (§VI-B).

What documents to analyze and how to extract input constraints from them automatically? Many types of software documents are available: manual pages, code comments, API documentation, requirement documents, etc. This paper studies and analyzes two popular types for constraint extraction, i.e., manual pages and code comments, since they describe whole program constraints (as opposed to API documentation that describes method level constrains), and they contain more code-level constraints (compared to requirement documents). We conduct an informal qualitative study of 82 manual pages from COREUTILS and code comments of 3 header files (ELF, tar and COFF). **Manual pages** (*man pages* for short) typically have higher English quality (e.g., grammatically correct full English sentences) since they are meant to be read by more than just the developers. On the other hand, it is easier to link constraints from **code comments** to code artifacts since

[2]There are 256 possibilities for a single 8-bit character option.

[3]We have additional child execution states for an invalid option and a null option for completeness. But the execution time for these two options should be less than the circles in Figure 3a combined.

[4]The actual time depends on the search strategy, but the time spent on testing `-o` would be much smaller than that of testing `-a`.

comments are embedded in the code (e.g., a comment typically describes the code segment right below it).

Valid options are typically described in a well-structured manner in man pages. Therefore, we use simple regular expression matching to extract them (§IV-C). Input file formats are described in both man pages and code comments. We use regular expesssion matching to analyze the man pages. Since code comments are less structured, we use NLP techniques, i.e., grammar relationships, for extraction (§IV-A). Grammar relationships can help identify relevant sentence structures for constraint extraction. It can tolerate different word orders and paraphrases, thus more general than hard-coded heuristics.

III. KLEE Background

KLEE is a symbolic execution engine based on LLVM. Programs are compiled into LLVM bytecode, and then interpreted by KLEE. KLEE models the programs' running states. It checks for dangerous operations (e.g., pointer dereferences and assertions) that can cause the program to fail. In addition, KLEE maintains path constraints that drive the execution to the current state. KLEE provides a function `klee_make_symbolic()` to make the memory symbolic, whose usages are tracked and constraints are collected. KLEE can also intercept the startup of programs and insert logic to make them support options for symbolic execution by using function `klee_init_env()`. Supported options include (1) `--sym-args MIN MAX N`, which expands to at least `MIN` and at most `MAX` symbolic arguments, each with a maximum length of `N`; and (2) `--sym-files NUM N`, which makes `stdin` and up to `NUM` files symbolic, each with a maximum size of `N`.

KLEE's default search strategy consists of two atom search strategies that are interleaved in round-robin fashion to prevent one atom strategy from getting stuck. The first atom strategy, coverage-optimized search, uses heuristics to choose a state that is most likely to cover new code in the immediate future. The second atom strategy, random path selection, randomly selects a branch to follow at a branch point, which helps alleviate starvation.

IV. Design and Implementation

This section describes how DASE extracts and utilizes input constraints for file formats (§IV-A and §IV-B) and options (§IV-C and §IV-D).

A. Extracting File Format Constraints

DASE automatically extracts input constraints regarding file formats from both code comments and man pages. As discussed in §II, code comments and man pages have different characteristics, so different techniques are used to extract constraints from them: NLP techniques for code comments, and regular expressions for man pages. The same techniques are used for all three file formats—ELF, tar, and COFF.

We apply NLP techniques to analyze the comments and code in header files to extract constraints automatically. 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 two constraints regarding array index-value pairs from the comments and code:

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

where `assume()` is a KLEE function for putting constraints onto the current path. The rest of this section explains the NLP techniques to generate the constraints.

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 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 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 propose to use Stanford typed dependency [27] to analyze the dependencies and grammatical relations among words and phrases in a sentence to handle these variants. Our technique is different from prior work [28], [29].

DASE uses four grammar rules (GR) to identify relevant comments and extract constraints from them. All four rules are used as main rules to identify relevant comments—if a sentence contains the typed dependency defined by a GR, it is considered relevant and remains for further analysis. GR1 and GR2 can also act as a supporting rule for any main rule. For example, GR1 can help identify the parameters in a rule, e.g., array and field names. The four GRs are listed below:

- **GR1: Noun or Adjectival Modifier (main/support rule)** Noun or Adjectival modifier is a noun or adjectival phrase that modifies a noun phrase [30]. 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/support rule)** Prepositional Modifier is a prepositional phrase that modifies the meaning of a verb, adjective, noun or preposition [30]. 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", "in" and "under") to locate specific nouns (i.e., "value" and "field") or specific word in the prepositional phrase (i.e., "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 [30]. 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 [30]. 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 that is shown at the beginning of this section to illustrate how to extract one type of constraints (index-value pairs) using the grammar rules on the three sentences (S1, S2 and S3).

- **S1:** 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".
- **S2:** 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.
- **S3:** 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.

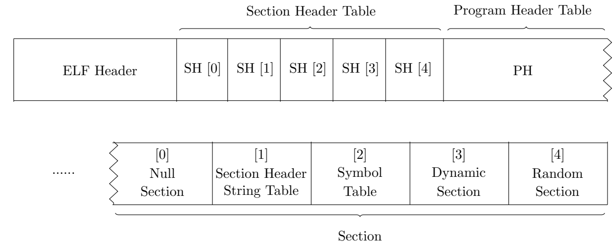

Fig. 4: DASE's ELF layout. SH is Section Header, and PH is Program Header. Numbers in brackets are array indices.

DASE aggregates the information from the three sentences. It then attempts to resolve the array name "e_ident" into a fully qualified name. Since there is enough information, it deducts "Elf32_Ehdr→e_ident" as the fully qualified name and generates two constraints.

In addition, DASE extracts constraints from man pages using regular expressions. Man pages often show a `struct` declaration, followed by the constraints (if available) for each field in the `struct`. The valid values for each struct field can be identified based on the indentation of the man page. Based on this layout, DASE first locates the name of the `struct`, and maps it to each of the constraints that are listed below it. The output of this analysis is also a list of constraints that can be directly used by the symbolic execution part of DASE.

B. Adding File Layout Constraints

ELF files follow a certain layout, which also defines valid ELF files. Therefore, in addition to extracting the file format constraints as described in § IV-A, we add file layout constraints for ELF by reading the ELF specification [31]. Our results show that both the file format and file layout constraints contribute to the improvement of DASE.

An ELF file always starts with an *ELF header* followed by the two header tables, *section header table* (SHT) and the *program header table* (PHT). SHT contains an array of *section header*s, PHT contains an array of *program header*s, and object files' real data are in the *section*s. In order to reduce the workload of the constraint solver and focus on important parts of ELF, we adopt a rigid layout as shown in Figure 4. SHT is set to have five section headers. The first section (at index 0) is a null section, followed by a string table, symbol table, dynamic section, and random section. The second and fifth section are set with a size of 8 bytes, and the third and fourth section are set with a size that is enough to hold two symbols. PHT is set to contain one random program header.

Note that our ELF layout is incomplete. We retain this incompleteness to give DASE the ability to explore close-to-valid inputs to explore boundary cases. In addition, input constraints can be slightly relaxed to include more close-to-valid inputs, which remains as our future work.

C. Extracting Valid Options

We automatically extract valid options only from man pages because we find that code comments do not describe valid options. Since man pages list the valid options in a standardized format, our parsers perform simple regular expression matching, which is effective and accurate. DASE takes a man page as input and outputs a list of valid command line options. It uses two regular expressions, one for *short* options (a single dash followed by a single letter), and one for *long* options (two dashes followed by multiple letters). If a short option has a long option equivalent, DASE keeps only the short option.

D. Using Options to Flatten Symbolic Execution

DASE takes the options extracted in §IV-C to trim and reorganize the dynamic symbolic execution tree as shown in Figure 3. Specifically, instead of having s symbolic arguments, DASE runs the program with $s-1$ symbolic arguments and a concrete valid option, which forms one execution branch. In this way, DASE creates n execution branches for a program with n valid options (one for each valid option). The aim is to balance the testing effort on each option (and the corresponding functionality), which should be of the similar semantic importance (at least not as skewed as $\frac{1}{2}$ versus $\frac{1}{2^n}$ as shown in Section II). The generated branches are then prioritized by search strategies. We can consider this technique as a "partition" of the execution tree. *The $s-1$ arguments remain symbolic, which can expand to any concrete options. Therefore, it is possible to cover combinations of command-line options such as "`-r -f`" (of `rm`) in our approach.* To ensure the completeness of this "partition", we add a branch for an invalid option and a branch for a null option.

V. METHOD

We use three coverage criteria reported by `gcov`, i.e., line, branch, and call coverage (% of executed function calls), as our main coverage metrics. The coverage criteria and `gcov` are widely used in literature [4], [9], [26], [32].

A. Evaluated Programs

We evaluate DASE on the following 88 programs from 5 popular and mature fundamental software suites for Unix-like systems. The sizes of these programs are at the *same scale* as the ones evaluated by previous work [9], [14], [26], [32].

COREUTILS 6.10. COREUTILS, also evaluated by KLEE, is a package of GNU programs that consists of basic file, shell, and text manipulation utilities. For a fair comparison with KLEE, our settings for DASE and KLEE for COREUTILS are identical; we set the same environment for COREUTILS as KLEE's authors; and we choose the same version, 6.10, and follow their parameters for both KLEE and DASE. The total source lines of code (LOC)[5] of the 82 stand-alone programs[6] that we tested in COREUTILS are 38,962 with a linked library size of 49,204 LOC. The program sizes range from 20–3,247 LOC. Here we show the program sizes and library sizes to give a better image of the scale of the programs. Following previous work [4], [32], coverage is measured against the programs excluding libraries as reported by `gcov` since a program typically use only part of libraries.

`diff` 3.3. `diff` compares files line by line and outputs the differences. The program has 1114 LOC with a library size of 43,324 LOC.

`grep` 2.18. `grep` searches files for given patterns. The program has 6,144 LOC with a library size of 38,663 LOC.

`objdump` & `readelf`(b) 2.24. These two programs are from BINUTILS, which is a set of GNU programs for processing binaries, libraries, object files, etc. `objdump` and `readelf` are used for displaying the contents of ELF files. They have 2,856 and 12,076 LOC respectively with a library size of 864,069 LOC. 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 provides similar tools as BINUTILS, but favors well-separated and well-documented libraries. They have 2,472 and 6,167 LOC respectively with a library size of 22,534 LOC.

B. Experimental Setup

All automatically extracted file format constraints and valid options (without manual examination for zero manual effort) are used as input constraints for all programs when applicable. DASE extracts file format constraints for ELF and uses them for the 4 ELF processing programs (`objdump`, `readelf`(b), `elfdump`, and `readelf`(e)) for path pruning. ELF is a boardly used main standard for binaries in Unix-like systems. *One can use the ELF model that we build to potentially improve test generation for all programs that read or write ELF binaries on a Unix-like platform.* In addition, DASE extracts valid options for the rest of the programs automatically and uses them to guide the symbolic execution on them.

To show the generality of our techniques of automatically extracting file format constraints, DASE extracts file format constraints for two additional standard file formats—Tar from `tar.h`, and the Common Object File Format (COFF) from `coff/internal.h`.

We run KLEE and DASE on each program until *no new instructions are covered in a certain amount of time*: 15 minutes for COREUTILS programs and 30 minutes for the rest due to their larger sizes. *This stop criterion allows both DASE and KLEE to run until they cannot make progress in coverage in a fixed time period, which is similar to that of the previous paper [32], but different from that of KLEE [4], in which each program is only allowed to run for one hour. In our experiments, the actual run time of each program varies from 6 seconds to 11.5 hours. We have also conducted experiments*

[5] Following previous work [26], [33], [34], all LOC counts in this paper are reported by `cloc` 1.62.

[6] `yes` is excluded because KLEE failed to terminate its execution. `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 to these programs in a sandbox to address this issue.

using the stop criterion from the KLEE paper, and DASE still achieves a similar amount of improvement over KLEE.

The other parameters are set by following the instructions from KLEE's authors [35]. The key parameters are:

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

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 §IV-B.

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.

VI. RESULTS

This section shows that DASE finds more previously unknown bugs, improves code coverage on top of different search strategies, complements developer tests, and extracts input constraints automatically. We also show that our results are statistically significant.

A. Detected Bugs

Using the constraints automatically inferred from documents (without any manual verification), DASE finds more bugs than KLEE. KLEE detects 3 previously unknown bugs from the 88 programs while DASE can uncover 13 previously unknown bugs (KLEE failed to detect 12 of them). Table I lists all of the detected previously unknown bugs.

DASE found 2 previously unknown bugs in COREUTILS and 3 in BINUTILS (`objdump` & `readelf`(b)), both of which have already been thoroughly tested by many symbolic execution tools. For example, COREUTILS has been tested by Veritesting [9], ZESTI [14] and KLEE [4], and BINUTILS has been tested by Veritesting [9], ZESTI [14], and KATCH [26]. Finding 5 new bugs in those extensively-tested suites demonstrates DASE's ability in finding new bugs and improving symbolic execution.

We explain a few example bugs to demonstrate DASE's bug finding capability. All these example bugs together with others (a total of 6) have already been confirmed and fixed by the developers after we reported the bugs to them.

`readelf`(b) fails with segmentation fault when the input file contains malformed attribute sections (of type `SHT_ARM_ATTRIBUTES`) [36]. The bug exists in the function `process_attributes()`, which is shown in Figure 5. Pointer `p` walks through the whole section. At line 19, 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 +`

TABLE I: 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	`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		✓

```
1  static int process_file_header(void) {
2    if (elf_header.e_ident[EI_MAG0] != ELFMAG0
3     || elf_header.e_ident[EI_MAG1] != ELFMAG1
4     || elf_header.e_ident[EI_MAG2] != ELFMAG2
5     || elf_header.e_ident[EI_MAG3] != ELFMAG3) {
6        error (_("Not an ELF file - ..."));
7        return 0;
8    } ...
9  }
10 ...
11 static int process_object(...) { ...
12   if (!process_file_header())
13       return 1; ...
14   process_arch_specific(file); /* calls
15       process_attributes() indirectly */ ...
16 }
17 ...
18 static int process_attributes(...) { ...
19   section_len = byte_get(p, 4);
20   p += 4;
21   ...
22   namelen = strlen((char *)p) + 1;
23   p += namelen;
24   section_len -= namelen + 4;
25
26   while (section_len > 0)
27   ...
28 }
```

Fig. 5: Buggy code in `readelf.c` from BINUTILS.

`4`, which causes an integer underflow at line 24. The variable `section_len`, 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.

Five other functions are ahead of `process_attributes()` in the call stack, namely, `main()`, `process_file()`, `process_object()`, `process_arch_specific()`, and `process_arm_specific()`. Each function reads and processes specific parts of the input ELF file. For example, to correctly invoke `process_attributes()`, the condition for the `if` statement at lines 2–5 must evaluate to false. The automatically extracted ELF constraints guide DASE to generate an ELF file that satisfies all these constraints to reach `process_attributes()` and expose the bug. This close-to-valid ELF file helps DASE detect this bug. `readelf`(e) contains a similar bug.

TABLE II: Coverage results with KLEE's default search strategy. "Line", "Br.", and "Call" show the total number of executable lines of code (ELOC), branches, and calls for each program, reported by gcov. "K" stands for KLEE and "D" is DASE. "Δ" is the improvement in percentage points of DASE over KLEE.

Program	**Line**	**K** %	**D** %	**Δ** pp	**Br.**	**K** %	**D** %	**Δ** pp	**Call**	**K** %	**D** %	**Δ** pp
COREUTILS	18329	66.2	75.6	**+9.4**	12674	69.9	77.3	**+7.4**	7008	56.6	67.5	**+10.9**
diff	526	59.1	67.9	**+8.8**	489	68.1	69.7	**+1.6**	150	46.6	59.3	**+12.7**
grep	932	37.3	58.4	**+21.1**	786	40.3	59.2	**+18.9**	266	33.5	53.6	**+20.1**
objdump	1687	19.4	25.6	**+6.2**	1270	16.9	22.8	**+5.9**	463	16.6	19.4	**+2.8**
readelf(b)	6998	6.9	15.2	**+8.3**	5410	6.2	16.6	**+10.4**	1959	6.9	13.5	**+6.6**
elfdump	1539	16.1	22.1	**+6.0**	1157	20.4	30.7	**+10.3**	533	16.5	23.6	**+7.1**
readelf(e)	3571	13.0	28.0	**+15.0**	2550	18.5	34.5	**+16.0**	1126	10.8	25.4	**+14.6**

The head program fails with memory exhaustion when invoked with options -c -1P, which tells head to print all but the last 1P bytes of the input file. Since P is a large unit of 1024^5, head tries to allocate a large amount of memory, which exceeds the total amount of available memory. 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 detect this bug.

Two bugs can be found by KLEE but not by DASE due to the following reason. The ELF file to trigger the bugs has a very large e_shoff value (SHT's offset from the beginning of the EFL file), which is incompatible with our ELF model. As shown in §IV-B, we manually fixed the offset to layout the SHT. Missing these two bugs shows the tradeoff involved in designing the ELF model. DASE focused on those more valid inputs to test the core logic.

Our results clearly demonstrate the benefits of our design choice: DASE finds 10 more bugs than KLEE. One can relax the constraints to explore fewer valid inputs and potentially cover these two bugs. Running KLEE and DASE together to gain benefits from both is also a good solution.

B. Code Coverage

Table II shows the overall code coverage achieved by KLEE and DASE. DASE outperforms KLEE on the 88 programs: it increases the line coverage, branch coverage, and call coverage by 14.2–120.3%, 2.3–167.7%, and 16.9–135.2% respectively, which are 6.0–21.1 pp, 1.6–18.9 pp, and 2.8–20.1 pp increases. For example, the line coverage boost on grep is 21.1 pp. Programs readelf(b), objdump, readelf(e), and elfdump are difficult to test because their inputs involve the complex ELF format. Despite the lower coverage, DASE detected new bugs in them that existing techniques did not detect as shown earlier. Figure 6 shows the coverage improvement of DASE over KLEE on readelf(b) over time. It shows that the improvement increases as time proceeds.

The coverage percentages for COREUTILS are different from those of the KLEE paper [4]. The difference is inevitable because the KLEE tool has evolved significantly since then, including major code changes of KLEE (e.g., removals of special tweaks), and an architecture change from 32-bit to 64-bit. We choose the latest version of KLEE at the time of the experiment because the original version used in the KLEE paper is not publicly available. For a fair comparison, the configurations for KLEE and DASE are identical.

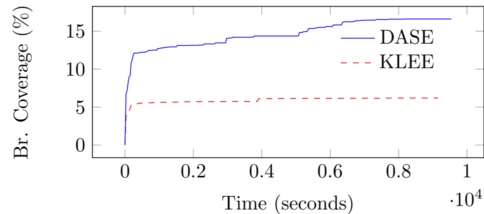

Fig. 6: Branch coverage on readelf(b) over time.

TABLE III: Number of instructions of generated test cases, showing that DASE explored deeper than KLEE. "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.

Program	**K-AVGi**	**D-AVGi**	**K-MAXi**	**D-MAXi**
COREUTILS	7132	8170	11682	13200
(82 programs)	49688	55672	320138	1189470
diff	18483	22427	35432	35976
grep	25942	26231	43424	53081
objdump	45915	62236	104479	206874
readelf(b)	11570	17800	24884	36196
elfdump	13827	28560	24433	319744
readelf(e)	18009	30069	29140	61233

DASE explored deeper than KLEE. Since DASE filters out "uninteresting" paths, we expect it to explore deeper. We count the number of executed instructions for each test case generated by KLEE and DASE to approximate the depth of the corresponding paths. The average and maximum numbers are shown in Table III, which shows that DASE generates test cases with much more instructions executed, indicating that DASE goes much deeper into the execution tree than KLEE. For the ELF processing programs, both the averages and maximums almost double their counterparts of KLEE. The difference is expected because while KLEE is still exploring at the early stage of the ELF sanity check, DASE has already penetrated through that part with the help of our ELF model.

DASE covered more functionalities and options than KLEE. To investigate DASE's coverage gain in detail, we manually check the coverage difference of KLEE and DASE on diff. KLEE explores only 27 out of the 55 distinct options[7], which are the shallower options, while DASE covers 46 options. The result agrees with our analysis in Figure 3. We have similar observations for the ELF processing programs. We manually examine 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. In addition,

[7]We count options that invoke the same code segment as one option.

TABLE IV: Coverage results with BFS.

Program	Line	K %	D %	Δ pp	Br.	K %	D %	Δ pp	Call	K %	D %	Δ pp
COREUTILS	1840	69.1	76.5	**+ 7.4**	1285	74.1	75.2	**+ 1.1**	728	53.8	69.5	**+ 15.7**
diff	526	40.9	71.1	**+ 30.2**	489	60.7	74.6	**+ 13.9**	150	33.3	64.0	**+ 30.7**
grep	932	33.4	59.6	**+ 26.2**	786	41.4	65.1	**+ 23.7**	266	25.2	54.9	**+ 29.7**
objdump	1687	2.9	3.6	**+ 0.7**	1270	5.3	5.6	**+ 0.3**	463	5.2	5.6	**+ 0.4**
readelf(b)	6998	0.7	0.8	**+ 0.1**	5410	1.6	1.6	**+ 0.0**	1959	1.1	1.2	**+ 0.1**
elfdump	1539	17.9	20.1	**+ 2.2**	1157	21.4	30.7	**+ 9.3**	533	19.3	21.4	**+ 2.1**
readelf(e)	3571	13.6	16.9	**+ 3.3**	2550	18.4	27.1	**+ 8.7**	1126	10.3	16.1	**+ 5.8**

TABLE V: Coverage of combining DASE with developer test cases, showing that DASE complements developer tests. "V" is developer tests. "D" is DASE combined with developer tests. Do note that readelf(e) has no developer test cases hence is missing.

Program	Line	V %	D %	Δ pp	Br.	V %	D %	Δ pp	Call	V %	D %	Δ pp
COREUTILS	18332	66.1	84.4	**+18.3**	12670	73.2	87.2	**+14.0**	7008	53.2	73.5	**+20.3**
diff	526	57.0	81.0	**+24.0**	489	72.2	87.3	**+15.1**	150	50.7	71.3	**+20.6**
grep	932	82.0	86.5	**+4.5**	786	87.0	89.8	**+2.8**	266	73.7	81.2	**+7.5**
objdump	1687	58.6	64.3	**+5.7**	1270	66.9	68.9	**+2.0**	463	51.2	57.2	**+6.0**
readelf(b)	7038	28.8	33.5	**+4.7**	5424	43.5	46.5	**+3.0**	1964	28.6	33.2	**+4.6**
elfdump	1084	71.6	75.0	**+3.4**	813	86.5	86.5	**+0.0**	264	46.2	52.9	**+6.7**

many other functions, such as print_symbol(), are missed by KLEE but covered by DASE.

The improvement of DASE generalizes to BFS. 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 II. Because it is too time-consuming (approximately 8 days) to run all the 88 programs, we randomly sample 5 programs from COREUTILS.

Table IV shows that when the search strategy is BFS, DASE still outperforms KLEE. Comparing Table IV with Table II, 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 input constraints 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.

C. DASE Complements Developer Tests

Since automated test generation aims to complement developer generated tests, we evaluate whether DASE finds bugs that developer tests cannot detect, and improves code coverage on top of developer tests. DASE *detected a total of 13 bugs on the evaluated programs that developer generated tests fail to detect* (§VI-A). Table V shows the coverage comparison. We can see that by adding DASE generated tests, the line coverage is improved by 3.4–24.0 pp. Together with Table II, we can see that for COREUTILS and diff, 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. The results demonstrate that DASE can be used by developers to find more bugs and further improve testing coverage even if manual tests exist.

D. Constraint Extraction Results

DASE automatically extracted input constraints from man pages and comments for command line options and three file formats with accuracies of 97.8–100%.

Specifically, for command-line options, DASE automatically extracted 776 valid options from man pages: 683 from the 82 COREUTILS programs, 46 from grep, and 47 from diff. The accuracy is 100%.

For ELF processing programs, we manually enforced 63 constraints to form the layout of our ELF model shown in Figure 4. By analyzing the ELF header file and ELF man page, DASE automatically extracts 60 values for 16 constraints regarding array index-value pairs, and 312 values for 20 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). In the case where a constraint exists from both the header file and man page, DASE combines all the values within both documents and creates a single new constraint with all the merged values. The ELF header file constraints is a superset of the man page constraints except for two constraints, which specify valid values for the EI_VERSION indices of the e_ident arrays. The accuracy of the extracted constraints is 97.8%.

The breakdown of the constraints extracted from the header file and the man page is as follows. From the man page, DASE automatically extracts 46 values for 16 constraints regarding array index-value pairs, and 72 values for 8 constraints regarding valid field values. The accuracy is 100%.

From the header file, DASE extracted 56 values for 14 constraints regarding array index-value pairs, and 312 values for 20 constraints regarding valid field values. Among the 312 values for 20 constraints regarding valid field values, 8 values are invalid, which affect 8 constraints. The accuracy is 97.8%. The inaccuracy results from a special kind of macro, *_NUM, in elf.h. This macro represents the total number of valid values, which is not a valid value. Among all the constraints, 10 constraints (consisting of 56 values) on special section types are not used because they are not applicable to our model. We can incorporate them when we improve our ELF model in the future.

DASE also extracted constraints from Tar and COFF's header files. It extracted 23 values for 2 constraints for the Tar file format, and 18 values for 2 constraints for the COFF file format. All of the extracted constraints are correct.

Impact of Incorrect Constraints. To understand the impact of incorrect constraints, we ran DASE with only the correct constraints (our main evaluation applies all constraints to minimize manual effort). The coverage and bug finding results are almost identical, suggesting that DASE is robust when a few incorrect constraints are provided.

Potential Effort Savings. Automated constraint extraction is important yet challenging [22], [23], and much work has been proposed to infer constraints from source code and execution traces automatically [37]–[39]. The proposed automated con-

straint extraction technique (takes 10–60 seconds to run) can save the effort of manually writing constraints. It is beneficial to automate the constraint extraction process to keep the constraints up to date since ELF, Tar and COFF file formats all have many revisions since their standardization.

DASE extracted almost all constraints in the header files and man pages. This can be expanded by analyzing more comprehensive file specifications such as the ELF specification [31]. In the future, we would like to extend the proposed NLP techniques to analyze other formats, e.g., TCP/IP packets and XML format.

E. Statistical Significance Test

Since there is randomness in KLEE's symbolic execution due to search strategies and the constraint solver, we conduct significance tests to check whether it is statistically significant that DASE outperforms KLEE. Since it takes a long time to run all 88 programs, we randomly sample 7 programs to perform the statistical significance tests: 5 COREUTILS programs, GREP, and ELFTOOLCHAIN `elfdump`. We run each program 3 time for KLEE and 3 time for DASE and perform Mann-Whitney U test (Wilcoxon rank-sum test) on each of the programs. The p-values are all smaller than 0.05, indicating statistical significance. Earlier results in Table II shows that DASE has a better coverage than KLEE; therefore it is *statistically significant that DASE achieves higher coverage than KLEE on these programs*.

VII. THREATS TO VALIDITY

While the natural language processing techniques are effective on the three evaluated file formats, the techniques may not generalize to other types of documentation. New grammar rules may be needed to support new types of sentence structures. In addition, the accuracy and quality of the extracted constraints depend on the quality of the documentation.

VIII. RELATED WORK

Symbolic Execution. Symbolic execution [7], [8] (alone or with concrete execution) has been widely used for automated testing [4], [26], [40]–[47]. To alleviate the path explosion problem, many strategies have been proposed [1], [4], [9], [16], [32], [44], [48]–[51]. Veritesting [9] leverages static symbolic execution to guide and improve dynamic symbolic execution. CUTE [44] and CREST [16] both use a bounded depth-first search (DFS) strategy. ZESTI [14] uses developer generated tests as "seeds" and explore paths similar to the seeds' paths. ZESTI's performance is affected by existing tests, while DASE does not suffer from this problem. DASE complements ZESTI: *DASE detected two previously unknown bugs in* COREUTILS *that were not detected by ZESTI (the same version of* COREUTILS *was used by DASE and ZESTI)*. Input space partitioning [52] has been used to improve symbolic execution [17], [53], [54]. For example, `FlowTest` [17] partitions the inputs into "non-interfering" blocks by analyzing the dependency among inputs.

The previous techniques rely on information from the code logic to guide the path exploration process. Different from them, DASE automatically extracts input constraints from documents and uses the constraints to prune execution paths. In addition, DASE focuses on valid and close-to-valid inputs while the above techniques have no knowledge about whether an execution path corresponds to valid or invalid input.

Input Constraints Guided Testing. Input constraints and specifications have been used for automated test generation [3], [18], [19], [24], [25]. These techniques require input constraints to be given manually. automatically extracting input constraints from documentation.

Lei Zhang developed the basis of this work as a Master's thesis [55]. This paper extends the thesis in several ways, including analyzing two additional file formats, tar and COFF, analyzing both manual pages and code comments for the file formats when applicable, and providing a clearer explanation of why DASE works.

Documentation Analysis. Many techniques analyze documents such as man pages to check for undocumented error codes [56], and code comments [28], [57], [58] and API documentation [59] for bug detection. These techniques do not improve symbolic execution based testing. In addition, DASE uses a new approach (typed dependencies) for document analysis and extracts different types of constraints.

Testing Effectiveness versus Code Coverage. A recent study [60] shows little correlation between code coverage and test suite effectiveness (measured by the number of mutants killed). However, only simple mutants generated by PIT [61] are used, which may not represent real bugs. Our results demonstrate that DASE improves testing effectiveness (i.e., detecting more new bugs) *and* code coverage over KLEE.

IX. CONCLUSIONS AND FUTURE WORK

This paper presents Document-Assisted Symbolic Execution (*DASE*)—a novel and general approach to extract input constraints from documents automatically to improve symbolic execution for automated bug detection and test generation. DASE prunes and flattens paths based on their *semantic* importance to help search strategies prioritize execution paths more effectively. DASE detected 12 previously unknown bugs that KLEE fails to detect, 6 of which have been confirmed by the developers on 88 mature programs. Compared to KLEE, DASE increases line coverage, branch coverage, call coverage by 6.0–21.1 pp, 1.6–18.9 pp, 2.8–20.1 pp, respectively. In the future, it would be promising to negate the input constraints to focus on testing error handling code.

ACKNOWLEDGMENT

The authors thank the statistical counseling service provided by the University of Waterloo and William Marshall for help with the statistical analysis of the results. The authors are grateful to Darko Marinov and Shan Lu for their feedback on the paper. This research is supported by the Natural Sciences and Engineering Research Council of Canada, a Google Faculty Research Award, and Ontario Ministry of Research and Innovation.

REFERENCES

[1] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2011, pp. 12–22.

[2] L. C. Briand and A. Wolf, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering*, 2007, pp. 85–103.

[3] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," *SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 123–133, 2002.

[4] C. Cadar, D. Dunbar, and D. 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*, 2008, pp. 209–224.

[5] P. Godefroid, M. Y. Levin, and D. Molnar, "Whitebox fuzzing for security testing," in *Proceedings of the Network and Distributed System Security Symposium*, 2008, pp. 20:20–20:27.

[6] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra, "Guided test generation for web applications," in *Proceedings of the International Conference on Software Engineering*, 2013, pp. 162–171.

[7] L. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 215–222, 1976.

[8] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[9] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1083–1094.

[10] A. Avancini and M. Ceccato, "Comparison and integration of genetic algorithms and dynamic symbolic execution for security testing of cross-site scripting vulnerabilities," *Information and Software Technology*, vol. 55, no. 12, pp. 2209–2222, 2013.

[11] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 1066–1071.

[12] M. K. Ganai, N. Arora, C. Wang, A. Gupta, and G. Balakrishnan, "BEST: A symbolic testing tool for predicting multi-threaded program failures," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 596–599.

[13] K. Li, C. Reichenbach, Y. Smaragdakis, Y. Diao, and C. Csallner, "SEDGE: Symbolic example data generation for dataflow programs," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, 2013, pp. 235–245.

[14] P. D. Marinescu and C. Cadar, "Make test-zesti: A symbolic execution solution for improving regression testing," in *Proceedings of the 2012 International Conference on Software Engineering*, 2012, pp. 716–726.

[15] P. Zhang, S. Elbaum, and M. B. Dwyer, "Automatic generation of load tests," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 43–52.

[16] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 443–446.

[17] R. Majumdar and R. Xu, "Reducing test inputs using information partitions," in *Proceedings of the 21st International Conference on Computer Aided Verification*, 2009, pp. 555–569.

[18] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, 2008, pp. 206–215.

[19] R. Majumda and R. Xu, "Directed test generation using symbolic grammars," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 134–143.

[20] C. Rubio-González and B. Liblit, "Defective error/pointer interactions in the linux kernel," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2011, pp. 111–121.

[21] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "Error propagation analysis for file systems," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 270–280.

[22] D. McClosky and C. D. Manning, "Learning constraints for consistent timeline extraction," in *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, 2012, pp. 873–882.

[23] K. Yoshikawa, S. Riedel, M. Asahara, and Y. Matsumoto, "Jointly identifying temporal relations with markov logic," in *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1*, 2009, pp. 405–413.

[24] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff, "A specification-based test case generation method for uml/ocl," in *Models in Software Engineering*. Springer, 2011, pp. 334–348.

[25] J. Offutt and A. Abdurazik, "Generating tests from uml specifications," in *Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard*, 1999, pp. 416–429.

[26] P. D. Marinescu and C. Cadar, "KATCH: High-coverage testing of software patches," in *Proceedings of the European Software Engineering Conference / ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2013, pp. 235–245.

[27] S. Bugzilla, "The Stanford natural language processing dependencies," http://nlp.stanford.edu/software/stanford-dependencies.shtml, 2015.

[28] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* iComment: Bugs or bad comments? */," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007, pp. 145–158.

[29] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language api documentation," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 307–318.

[30] M.-C. de Marneffe and C. D. Manning, "Stanford typed dependencies manual," http://nlp.stanford.edu/software/dependencies_manual.pdf, 2013.

[31] T. I. Standards, "Executable and linkable format," http://www.skyfree.org/linux/references/ELF_Format.pdf, 2015.

[32] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2013, pp. 19–32.

[33] T. Kuchta, C. Cadar, M. Castro, and M. Costa, "Docovery: Toward generic automatic document recovery," in *Proceedings of the International Conference on Automated Software Engineering*, 2014.

[34] P. D. Marinescu and C. Cadar, "High-coverage symbolic patch testing," in *Proceedings of the 19th International Conference on Model Checking Software*, 2012, pp. 7–21.

[35] T. K. Team, "OSDI'08 Coreutils Experiments," http://klee.github.io/docs/coreutils-experiments/, 2015.

[36] S. Bugzilla, "Readelf bug 16664 - Segmentation fault in process_attributes() of readelf.c," https://sourceware.org/bugzilla/show_bug.cgi?id=16664, 2014.

[37] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007.

[38] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 291–301.

[39] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001, pp. 57–72.

[40] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 265–278.

[41] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[42] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic pathfinder: Integrating symbolic execution with model checking for java bytecode analysis," *Automated Software Engineering*, vol. 20, no. 3, pp. 391–425, 2013.

[43] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2010, pp. 513–528.

[44] K. Sen, D. Marinov, and G. 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*, 2005, pp. 263–272.

[45] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *Tests and Proofs*. Springer, 2008, pp. 134–153.

[46] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: Techniques and tradeoffs," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2010, pp. 257–266.

[47] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proceedings of the 18th International Conference on Static Analysis*, 2011, pp. 95–111.

[48] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery," in *Proceedings of the 20th USENIX conference on Security*, 2011, pp. 10–10.

[49] K. Krishnamoorthy, M. S. Hsiao, and L. Lingappan, "Strategies for scalable symbolic execution-driven test generation for programs," *Science China Information Sciences*, vol. 54, no. 9, pp. 1797–1812, 2011.

[50] R. Santelices and M. J. Harrold, "Exploiting program dependencies for scalable multiple-path symbolic execution," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 195–206.

[51] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, "Carfast: Achieving higher statement coverage faster," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 35:1–35:11.

[52] E. J. Weyuker and T. J. Ostrand, "Theories of program testing and the application of revealing subdomains," *Software Engineering, IEEE Transactions on*, vol. SE-6, no. 3, pp. 236–246, 1980.

[53] M. Staats and C. Păsăreanu, "Parallel symbolic execution for structural test generation," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 183–194.

[54] D. Qi, H. D. Nguyen, and A. 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*, 2011, pp. 278–288.

[55] L. Zhang, "DASE: Document-assisted symbolic execution for improving automated test generation," Master's thesis, University of Waterloo, 2014.

[56] C. Rubio-González and B. Liblit, "Expect the unexpected: Error code mismatches between documentation and the real world," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2010, pp. 73–80.

[57] L. Tan, Y. Zhou, and Y. Padioleau, "aComment: Mining annotations from comments and code to detect interrupt-related concurrency bugs," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 11–20.

[58] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing javadoc comments to detect comment-code inconsistencies," in *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, 2012, pp. 260–269.

[59] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring specifications for resources from natural language api documentation," *Automated Software Engineering Journal*, vol. 18, no. 3-4, pp. 227–261, 2011.

[60] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the International Conference on Software Engineering*, 2014, pp. 435–445.

[61] H. Coles, "PIT mutation operators," http://pitest.org/quickstart/mutators/.