Grammar-Agnostic Symbolic Execution by Token Symbolization

Weiyu Pan
College of Computer, National University of Defense Technology
Changsha, China
panweiyu@nudt.edu.cn

Zhenbang Chen*
College of Computer, National University of Defense Technology
Changsha, China
zbchen@nudt.edu.cn

Guofeng Zhang
College of Computer, National University of Defense Technology
Changsha, China
zhangguofeng16@nudt.edu.cn

Yunlai Luo
College of Computer, National University of Defense Technology
Changsha, China
luoyl@nudt.edu.cn

Yufeng Zhang
College of Computer Science and Electronic Engineering, Hunan University
Changsha, China
yufengzhang@nudt.edu.cn

Ji Wang
College of Computer, State Key Laboratory of High Performance Computing, National University of Defense Technology
Changsha, China
wj@nudt.edu.cn

ABSTRACT

Parsing code exists extensively in software. Symbolic execution of complex parsing programs is challenging. The inputs generated by the symbolic execution using the byte-level symbolization are usually rejected by the parsing program, which dooms the effectiveness and efficiency of symbolic execution. Complex parsing programs usually adopt token-based input grammar checking. A token sequence represents one case of the input grammar. Based on this observation, we propose grammar-agnostic symbolic execution that can automatically generate token sequences to test complex parsing programs effectively and efficiently. Our method's key idea is to symbolize tokens instead of input bytes to improve the efficiency of symbolic execution. Technically, we propose a novel two-stage algorithm: the first stage collects the byte-level constraints of token values; the second stage employs token symbolization and the constraints collected in the first stage to generate the program inputs that are more possible to pass the parsing code.

We have implemented our method on a Java Pathfinder (JPF) based concolic execution engine. The results of the extensive experiments on real-world Java parsing programs demonstrate the effectiveness and efficiency in testing complex parsing programs. Our method detects 6 unknown bugs in the benchmark programs and achieves orders of magnitude speedup to find the same bugs.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Symbolic execution, Grammar, Parsing Program, Tokenization

ACM Reference Format:

Weiyu Pan, Zhenbang Chen, Guofeng Zhang, Yunlai Luo, Yufeng Zhang, Ji Wang. 2021. Grammar-Agnostic Symbolic Execution by Token Symbolization. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460319.3464845

1 INTRODUCTION

Parsing [1] is usually the first step in software. Many programs need to parse the input files or strings in the initial stage of execution. Suppose the inputs are not valid with respect to some specific formats. In that case, the program may exit and throw an exception or output an error message, which indicates the invalidness of the inputs. There is usually an input grammar [16] that specifies the requirements of valid inputs. Automatic testing of these complex parsing programs without the input grammars is challenging [10].

Symbolic execution [11, 19] provides a general framework for exploring the program's path space. In symbolic execution, the program is executed in a symbolic manner. Symbolic execution maintains a path condition (*i.e.*, a quantifier-free first-order logic formula [20], denoted as *PC*) for each program path. When executing a branch statement, symbolic execution explores both branches of the statement after checking each branch's feasibility by solving the branch's path condition. If a branch's *PC* is unsatisfiable [20], the exploration does not continue, *i.e.*, the path to this branch is unreachable; otherwise, the execution of the statements inside the branch continues, and the path's *PC* is updated by adding the branch's condition. In this way, the path space of the program is systematically explored. Symbolic execution provides a base technique for efficiently testing programs in an automatic manner. We can solve the *PC* of each program path to generate a program input. There are already many successful symbolic execution based automatic testing tools, such as KLEE [4], Pex [32], and SPF [28], to name a few.

From the view of programmers, there are grammars in their minds for checking the validity of inputs. However, these grammars may not be available to the third party. We notice that these

*Weiyu Pan and Zhenbang Chen contributed equally to this work and are co-first authors. Zhenbang Chen and Ji Wang are the corresponding authors.

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464845

grammars are often embedded in token-based implementation. Usually, the complex parsing program's execution can be divided into three stages: *tokenization*, *grammar checking* and *application logic*. In the first stage, the input is tokenized into a sequence of tokens, and each token represents a sub-sequence of the characters or bytes in the input. After the first stage, grammar checking checks whether the tokenized input, *i.e.*, the sequence of tokens, satisfies the grammar rules. After this step, the input is considered a valid input, which will then be processed by the application logic code. For example, suppose that we have an evaluator program for the binary expression of numbers. The program's input grammar is as follows, where ⟨NUM⟩ and ⟨OP⟩ represent a number and an operator, respectively, and their tokens are `T_NUM` and `T_OP`.

$$\langle \mathsf{EXP} \rangle \rightarrow \langle \mathsf{NUM} \rangle \langle \mathsf{OP} \rangle \langle \mathsf{NUM} \rangle$$

If the input is "`11 + 22`", in the first stage, "`11`", "`+`" and "`22`" are tokenized to three tokens `T_NUM`, `T_OP` and `T_NUM`, respectively. The token sequence composed by these three tokens satisfies the grammar. Then, the evaluation converts the two number strings to two integers and calculates the result as 33. However, if the input is "`1a + 22`", the input cannot pass the tokenization code because "`1a`" is not a number string; besides, if the input is "`+ + 22`", it can be tokenized but the token sequence does not satisfy the grammar, *i.e.*, the input is also rejected.

It is challenging for symbolic execution to analyze token-based parsing programs. If we symbolize the program inputs blindly, *e.g.*, symbolizing every byte of the inputs, it will be very hard for the symbolic execution to analyze the code in the third stage or even part of the second stage. The tokenizer or the grammar checker may reject many inputs generated by symbolic execution. This problem challenges the automatic testing of complex parsing programs based on symbolic execution. There is existing work to tackle this problem in symbolic execution [8, 10, 23]; however, the existing work requires to provide the input grammar, which is often unavailable and hard to infer [24].

We observe that token abstracts the inputs of complex parsing programs. Different inputs may be tokenized to be the same token. Besides, input grammar checking is often implemented by checking the token sequence of the input instead of the character sequence. Hence, different token sequences are more effective for testing the complex parsing program. Suppose that we can symbolize the tokens during symbolic execution and generate new token sequences. In that case, the grammar checking code will be tested more efficiently, which also directly improves the effectiveness of testing the application logic. Different token sequences generated with respect to the grammar checking code abstract the different cases of the valid input requirements or even the application logic.

Based on this observation, we propose grammar-agnostic symbolic execution, *i.e.*, a framework for effective symbolic execution of complex parsing programs based on token symbolization *without the need of input grammars*. Our framework does not collect the byte-level constraint in the tokenization stage but collects the token constraints in the grammar checking stage. Then, our framework can generate new token sequences using the token constraints. Two technical problems challenge our framework: (1) how to generate the input of a token sequence? (2) how to analyze the code in application logic in priority?

For the first problem, we propose to do the symbolic execution of tokenization code first and collect the constraints describing the possible values of tokens. Then, when generating the input from a token sequence, our framework uses these constraints to generate the program input. For the second problem, we propose maintaining the constraints collected in application logic separately and exploring the corresponding unexplored paths in priority under the specific token sequence. In this way, our framework tests the code in application logic in priority and automatically generates the inputs for different input grammar cases.

In principle, our method can be viewed as an instance of compositional symbolic execution [9][18], which usually uses function-level summaries to reduce the program's path space and improve symbolic execution's efficiency. The symbolic execution of tokenization code extracts the summary of tokenization. Then, when doing the symbolic execution of the parsing program, we only collect the token constraints in the grammar checking code but ignore the byte-level constraints in the tokenization code, and the token-level path exploration is the system-level symbolic execution in compositional symbolic execution for complex parsing programs. When generating the byte-level inputs, we use the tokenization summaries and the token-level constraint to construct the byte-level constraint, which also corresponds to the stitching of system-level constraints and function-level summaries in compositional symbolic execution.

As far as we know, *our work is the first parsing-oriented symbolic execution framework that does not need the input grammar*. We have implemented our method in a prototype for Java programs based on Symbolic PathFinder (SPF) [27]. The results of the extensive experiments on real-world benchmark programs indicate the effectiveness and efficiency of our approach.

Our main contributions are as follows.

- We propose the framework of grammar-agnostic symbolic execution that symbolizes tokens to generate valid program inputs more efficiently.
- We propose a two-stage algorithm that collects the token constraints in the first stage and then generates valid inputs to quickly cover the grammar checking code and application logic code in the second stage.
- We have implemented our method in a prototype based on JPF and carried out extensive experiments on real-world open-source Java parsing programs (121531 lines of code in total).
- Our method detects 6 unknown bugs and improves both statement coverage and branch coverage. Compared with byte-level symbolization and fuzzing methods, our method achieves orders of magnitude speedups to find the same bugs.

The remainder of this paper is organized as follows. Section 2 briefly introduces dynamic symbolic execution and gives a motivation example. Section 3 depicts our framework in details. Section 4 gives the implementation and evaluation. Section 5 discusses the limitations of our approach. Section 6 reviews the related work and compares them with our method. Section 7 concludes the paper.

```
1 public void entry(String a) throws ParseException{
2  // inputReader's type is Reader
3  inputReader = new StringReader(a);
4  parseExpr();
5  // application logic starts
6  if (a.charAt(a.length() - 1) == 'z') {
7   assert(false); //bug
8  }
9 }
10
11 void parseExpr() throws ParseException {
12  int token = getNextToken();
13  if (token == T_NUM){
14   parseOp();
15   return;
16  } else if (token == T_ID){
17   if (getNextToken() == T_EOF) return;
18  }
19  throw new ParseException();
20 }
21
22 void parseOp() throws ParseException {
23  int token = getNextToken();
24  if (token == T_OP){
25   parseExpr();
26  } else if (token == T_EOF){
27   return;
28  }
29  throw new ParseException();
30 }
31
32 int getNextToken() throws ParseException {
33  int res = inputReader.read();
34  if (res == -1) return T_EOF;
35  char c = (char) res;
36  if (c >= '*' && c <= '+'){
37   return T_OP;
38  } else if (c >= 'a' && c <= 'z'){
39   return T_ID;
40  } else if (c >= '0' && c <= '9'){
41   char next_c = (char) inputReader.read();
42   if (next_c >= '0' && next_c <= '9'){
43    return T_NUM;
44   }
45  }
46  throw new ParseException();
47 }
```

Figure 1: An example parsing program.

2 ILLUSTRATION

2.1 Dynamic Symbolic Execution

We use dynamic symbolic execution (DSE) [11, 30] to analyze complex parsing programs. DSE (or concolic execution) combines traditional symbolic execution and concrete execution to analyze a program. Given a program $\mathcal{P}$, the initial input I and the input's symbolization strategy, DSE executes $\mathcal{P}$ with I concretely, which generates a path p. At the same time, DSE also carries out symbolic execution along p and records the unexplored off-the-path branches along p. An off-the-path branch corresponds to the negation of a branch along p. For example, if p's path condition is $PC(p) = \wedge_{i=1}^{n} C_i$ and C_i is the symbolic condition of the branch b_i, the path condition of b_j's off-the-path branch (denoted as $\neg b_j$) is $PC(\neg b_j) = (\wedge_{i=1}^{j-1} C_i) \wedge \neg C_j$, where $1 \le j \le n$. When the concrete execution terminates, DSE selects an off-the-path branch b and solves the path condition of b to generate a new input to do the concolic execution of $\mathcal{P}$ again. The off-the-path branch selection is determined by the search heuristic, such as depth-first search (DFS) and breadth-first search (BFS), which controls the style of path exploration. This procedure continues until timeout or there is no unexplored off-the-path branches.

$$
\begin{array}{rcl}
\langle \text{Expr} \rangle & ::= & \langle \text{ID} \rangle \mid \langle \text{Number} \rangle \mid \langle \text{Number} \rangle\ \langle \text{Op} \rangle\ \langle \text{Expr} \rangle \\
\langle \text{ID} \rangle & ::= & \texttt{'a'} \mid \texttt{'b'} \mid \dots \mid \texttt{'y'} \mid \texttt{'z'} \\
\langle \text{Number} \rangle & ::= & \texttt{'00'} \mid \texttt{'01'} \mid \dots \mid \texttt{'98'} \mid \texttt{'99'} \\
\langle \text{Op} \rangle & ::= & \texttt{'*'} \mid \texttt{'+'}
\end{array}
$$

Figure 2: Grammar in the example program, where ⟨ID⟩ must be a one-character identity and ⟨Number⟩ must be a two-digital number.

2.2 Motivation Example

This subsection uses a motivation example to illustrate our method. Figure 1 shows a Java parsing program extracted from real-world programs. The program implements the parser for the grammar in Figure 2. ⟨Expr⟩ is the entry non-terminal. This grammar accepts an expression that can be a single-character name (⟨ID⟩), a two-digit number (⟨Number⟩) or a composite expression whose left is a number and right is an expression. In Figure 1, `entry` function accepts an input string `a` and initializes the `inputReader` object. Then, `parseExpr` is used to parse the input string. If the parsing is successful, `entry` checks whether the last character is `'z'`. The true branch contains a bug (Line 7). `parseExpr` implements a recursive descent parser [1]. `getNextToken` reads the next character `c` and checks `c` to return a token. There are four token values in total.

2.2.1 Original DSE. Suppose that the initial input string is `"12+13"` and we symbolize each character. Then, the path condition of the first iteration (denoted as PC_1) is as follows, where $a[i]$ represents the ith character's symbolic value, and *the right-side numbers are the line numbers of the branch conditions that generate the constraints.*

$$
\begin{array}{lr}
a[0] \ge \texttt{'*'} \wedge a[0] > \texttt{'+'} \wedge a[0] < \texttt{'a'} \wedge & (36\&38) \\
a[0] \ge \texttt{'0'} \wedge a[0] \le \texttt{'9'} \wedge a[1] \ge \texttt{'0'} \wedge a[1] \le \texttt{'9'} \wedge & (40\&42) \\
a[2] \ge \texttt{'*'} \wedge a[2] \le \texttt{'+'} \wedge & (36) \\
a[3] \ge \texttt{'*'} \wedge a[3] > \texttt{'+'} \wedge a[3] < \texttt{'a'} \wedge & (36\&38) \\
a[3] \ge \texttt{'0'} \wedge a[3] \le \texttt{'9'} \wedge a[4] \ge \texttt{'0'} \wedge a[4] \le \texttt{'9'} \wedge & (40\&42) \\
a[4] \ne \texttt{'z'} & (6)
\end{array}
$$

There are 17 off-the-path branches along the first path. If we use DFS to select the next off-the-path branch, *i.e.*, the one corresponding to the last branch whose condition is $a[4] \ne \texttt{'z'}$, the path condition for generating the new input would be PC_1 except the last condition is changed to $a[4] = \texttt{'z'}$, which is as follows (denoted as PC_2).

$$
\begin{array}{l}
a[0] \ge \texttt{'*'} \wedge a[0] > \texttt{'+'} \wedge a[0] < \texttt{'a'} \wedge \\
a[0] \ge \texttt{'0'} \wedge a[0] \le \texttt{'9'} \wedge a[1] \ge \texttt{'0'} \wedge a[1] \le \texttt{'9'} \wedge \\
a[2] \ge \texttt{'*'} \wedge a[2] \le \texttt{'+'} \wedge \\
a[3] \ge \texttt{'*'} \wedge a[3] > \texttt{'+'} \wedge a[3] < \texttt{'a'} \wedge \\
a[3] \ge \texttt{'0'} \wedge a[3] \le \texttt{'9'} \wedge a[4] \ge \texttt{'0'} \wedge a[4] \le \texttt{'9'} \wedge \\
\boxed{\text{a[4] = 'z'}}
\end{array}
$$

However, PC_2 is unsatisfiable because of $a[4]$'s three constraints. Then, we select the next off-the-path branch that is generated at Line 42 and the path condition (denoted as PC_3) is as follows.

$$\begin{array}{l} a[0] \geq \texttt{'*'} \wedge a[0] > \texttt{'+'} \wedge a[0] < \texttt{'}a\texttt{'} \wedge \\ a[0] \geq \texttt{'0'} \wedge a[0] \leq \texttt{'9'} \wedge a[1] \geq \texttt{'0'} \wedge a[1] \leq \texttt{'9'} \wedge \\ a[2] \geq \texttt{'*'} \wedge a[2] \leq \texttt{'+'} \wedge \\ a[3] \geq \texttt{'*'} \wedge a[3] > \texttt{'+'} \wedge a[3] < \texttt{'}a\texttt{'} \wedge \\ a[3] \geq \texttt{'0'} \wedge a[3] \leq \texttt{'9'} \wedge a[4] \geq \texttt{'0'} \wedge \boxed{\mathrm{a[4]} > \text{'9'}} \end{array}$$

PC_3 is satisfiable. Suppose that solving PC_3 generates "12+1z", which will be rejected by the parsing program, because the last character is not a number character. In this way, we need 6 iterations to cover Line 17 and Line 39 under DFS. If the DSE employs BFS, it still needs 6 iterations to cover Line 17 and Line 39. In summary, the DSE with byte-level symbolization generates many invalid inputs that will be rejected by the parsing program and do not contribute to the testing of the program.

Besides, ***it is impossible for DSE to detect the bug at Line 7 under the initial input*** "12+13". The reason is that there exist no 5-length strings that satisfy the grammar [1] and whose last character is 'z'. However, there do exist valid input strings that can trigger the bug, *e.g.*, "12+z".

2.2.2 Grammar-Agnostic DSE. Our grammar-agnostic DSE is a two-stage procedure. In the first stage, we just do the DSE of the tokenization code, which collects the constraint of each token value, *i.e.*, the symbolic summary [9] of tokenization code. Our framework starts with a one-size input and gradually increases the input size to collect the token values and their constraints. After the first stage, each collected token has a concrete value (usually an integer value) and its corresponding byte-level constraint. For the example program in Figure 1, our framework does the DSE of `getNextToken`. In the beginning, the input string's length is one. The paths explored by the DSE of `getNextToken` are two normally terminated paths. The others are all paths with a parsing exception. The two normally terminated paths correspond to the token values `T_OP` and `T_ID`, respectively. Their path constraints are as follows, where $\mathcal{TC}[t]$ represents the path constraint of token value t, where $t[j]$ represents the ith character's symbolic value in the string represented by t.

$$\begin{array}{lcl} \mathcal{TC}[\texttt{T\_ID}] & = & t[0] \geq \texttt{'}a\texttt{'} \wedge t[0] \leq \texttt{'}z\texttt{'} \\ \mathcal{TC}[\texttt{T\_OP}] & = & t[0] \geq \texttt{'*'} \wedge t[0] \leq \texttt{'+'} \end{array}$$

The parsing exception paths are ignored. Hence, we have collected the constraints of two token values. Then, we increase the input size to two and do the DSE of `getNextToken` again. We will collect three normally terminated paths, in which there is a new token value `T_NUM`, and the constraints for `T_ID` and `T_OP` are the same as those generated by one-size input. The constraint of `T_NUM` is as follows.

$$\mathcal{TC}[\texttt{T\_NUM}] = t[0] \geq \texttt{'0'} \wedge t[0] \leq \texttt{'9'} \wedge t[1] \geq \texttt{'0'} \wedge t[1] \leq \texttt{'9'}$$

So, after the two times of DSE for `getNextToken`, we get all the token values and the constraints. Then, if we increase the input size to three, there will be no new token value generated and no new constraint for each already generated token value. This first stage terminates. In practice, we set a *threshold* to terminate the first stage (Section 2.2.1). The result of the first stage is a map $\mathcal{TC}$ that records the explored token values and their input constraints. $\mathcal{TC}$ actually gives a symbolic summary of the method `getNextToken`, *i.e.*, the relation between the inputs and the return values.

After the first stage, our framework starts the second stage, in which we symbolize both the token generated and each byte in the input. Our framework maintains two path conditions: one for the symbolized tokens (denoted as PC_T) and the other for the branches in application logic code (denoted as PC_A). More specifically, the framework maintains two sets OB_T and OB_A of the off-the-path branches for the grammar checking code and the application logic code, respectively. Notably, the framework does not collect the constraints of the branches in the tokenization code. Similar to the system-level symbolic execution in compositional symbolic execution, the path exploration at the token-level is more effective for testing the parsing program.

Then, after exploring a path, the framework first selects an off-the-path branch b_a from the application logic's off-the-path branch set OB_A and generates an input by solving the constraint composed by $PC(b_a)$ and the current token constraint PC_T, *i.e.*, $PC_T \wedge PC(b_a)$. The solving of this new path condition contains three steps: first, we solve PC_T to get a sequence of token values; second, based on these values and the token constraint map $\mathcal{TC}$ generated in the first stage, we generate the byte-level constraint for PC_T (denoted as PC_T^C), and PC_T^C reuses the summary of the tokenization code in a similar way of stitching the system-level constraint and the function-level constraints in compositional symbolic execution; finally, we solve $PC_T^C \wedge PC(b_a)$ to generate the new input. If there is no more branches in OB_A, the framework selects an off-the-path branch b_t from the grammar checking's off-the-path branch set OB_T and solves the token constraint $PC(b_t)$ as before (*i.e.*, $PC(b_a)$ is $true$) to generate a new input. This procedure iterates until there are no branches in the grammar checking's off-the-path branch set OB_T or timeout.

For the example program, suppose that the initial input of the second stage is also "12+13". In the second stage, after the first execution, our framework collects the following path condition, where $\mathbf{T}[i]$ represents the ith token's symbolic value.

$$\underbrace{\mathbf{T}[0] = \texttt{T\_NUM} \wedge \mathbf{T}[1] = \texttt{T\_OP} \wedge \mathbf{T}[2] = \texttt{T\_NUM}}_{PC_T} \wedge \underbrace{a[4] \neq \texttt{'z'}}_{PC_A}$$

There are three off-the-path branches in the grammar checking's off-the-path branch set OB_T and one in the application logic code's off-the-path branch set OB_A. We select the branch in OB_A whose path condition is $a[4] = \texttt{'z'}$. Hence, the new path constraint is as follows.

$$\underbrace{\mathbf{T}[0] = \texttt{T\_NUM} \wedge \mathbf{T}[1] = \texttt{T\_OP} \wedge \mathbf{T}[2] = \texttt{T\_NUM}}_{PC_T} \wedge \underbrace{a[4] = \texttt{'z'}}_{PC(b_a)}$$

To solve $PC_T \wedge PC(b_a)$, we solve PC_T to get a token value sequence, based on which and $\mathcal{TC}$ we generate a byte-level constraint PC_T^C by mapping the token values to their constraints. Solving the above PC_T generates the token sequence $\langle \texttt{T\_NUM}, \texttt{T\_OP}, \texttt{T\_NUM} \rangle$. Hence, PC_T^C is as follows.

$$\begin{array}{ll} a[0] \geq \texttt{'0'} \wedge a[0] \leq \texttt{'9'} \wedge a[1] \geq \texttt{'0'} \wedge a[1] \leq \texttt{'9'} \wedge & \mathcal{TC}[\texttt{T\_NUM}] \\ a[2] \geq \texttt{'*'} \wedge a[2] \leq \texttt{'+'} \wedge & \mathcal{TC}[\texttt{T\_OP}] \\ a[3] \geq \texttt{'0'} \wedge a[3] \leq \texttt{'9'} \wedge a[4] \geq \texttt{'0'} \wedge a[4] \leq \texttt{'9'} & \mathcal{TC}[\texttt{T\_NUM}] \end{array}$$

However, $PC_T^C \wedge PC(b_a)$ is unsatisfiable, which means any inputs generating the current token sequence, *i.e.*, $\langle$T_NUM, T_OP, T_NUM$\rangle$, can not trigger the bug. Now, there is no branch in OB_A, which means DSE finishes the path exploration of the application logic under the current token sequence. Then, we select an off-the-path branch b_t from the grammar checking's off-the-path branch set OB_T. Suppose that we also employ DFS and the path condition of b_t, *i.e.*, $PC(b_t)$, is as follows.

$$\mathbf{T}[0] = \texttt{T\_NUM} \wedge \mathbf{T}[1] = \texttt{T\_OP} \wedge \mathbf{T}[2] \neq \texttt{T\_NUM}$$

Besides $PC(b_t)$, we also add the following range constraint PC_R for all the token variables (omitted for the last step), where the values are the key values of $\mathcal{TC}$.

$$\bigwedge_{i=1}^{3} \mathbf{T}[i] \in \{\texttt{T\_ID}, \texttt{T\_NUM}, \texttt{T\_OP}\}$$

Solving $PC(b_t) \wedge PC_R$ explores a new path at the token level, which can be considered as exploring a new system-level path in compositional symbolic execution to improve the efficiency of the symbolic execution. Suppose that solving $PC(b_t) \wedge PC_R$ generates the solution in which $\mathbf{T}[2]$ is T_ID. The new token sequence is $\langle$T_NUM, T_OP, T_ID$\rangle$. Then, the byte-level constraint $PC_c(b_t)$ is as follows.

$$\begin{array}{ll} a[0] \geq \texttt{'0'} \wedge a[0] \leq \texttt{'9'} \wedge a[1] \geq \texttt{'0'} \wedge a[1] \leq \texttt{'9'} \wedge & \mathcal{TC}[\texttt{T\_NUM}] \\ a[2] \geq \texttt{'*'} \wedge a[2] \leq \texttt{'+'} \wedge & \mathcal{TC}[\texttt{T\_OP}] \\ a[3] \geq \texttt{'a'} \wedge a[3] \leq \texttt{'z'} & \mathcal{TC}[\texttt{T\_ID}] \end{array}$$

Suppose that solving $PC_c(b_t)$ generates the input string "11+a". The concolic execution of the example program under "11+a" covers Lines 17&39 and collects the following path constraints

$$\underbrace{\mathbf{T}[0] = \texttt{T\_NUM} \wedge \mathbf{T}[1] = \texttt{T\_OP} \wedge \mathbf{T}[2] \neq \texttt{T\_NUM} \wedge \mathbf{T}[2] = \texttt{T\_ID}}_{PC_T} \wedge \underbrace{a[3] \neq \texttt{'z'}}_{PC_A}$$

Then, same as before, we select the branch in the application logic's off-the-path branch set OB_A and generate the following constraint.

$$\underbrace{\mathbf{T}[0] = \texttt{T\_NUM} \wedge \mathbf{T}[1] = \texttt{T\_OP} \wedge \mathbf{T}[2] \neq \texttt{T\_NUM} \wedge \mathbf{T}[2] = \texttt{T\_ID}}_{PC_T} \wedge \underbrace{a[3] = \texttt{'z'}}_{PC(b_a)}$$

This constraint corresponds to the following byte-level constraint, which is satisfiable.

$$\begin{array}{ll} a[0] \geq \texttt{'0'} \wedge a[0] \leq \texttt{'9'} \wedge a[1] \geq \texttt{'0'} \wedge a[1] \leq \texttt{'9'} \wedge & \mathcal{TC}[\texttt{T\_NUM}] \\ a[2] \geq \texttt{'*'} \wedge a[2] \leq \texttt{'+'} \wedge & \mathcal{TC}[\texttt{T\_OP}] \\ a[3] \geq \texttt{'a'} \wedge a[3] \leq \texttt{'z'} \wedge & \mathcal{TC}[\texttt{T\_ID}] \\ a[3] = \texttt{'z'} & PC(b_a) \end{array}$$

Suppose that the solving generates "11+z", which is accepted by the grammar and triggers the bug at Line 7.

In summary, by employing grammar-agnostic DSE, we can cover Lines 17&39 at the 2nd execution and trigger the bug at Line 7 at the 3rd execution.

3 METHOD

This section presents the details of grammar-agnostic DSE. The framework will be introduced first. Then, the collection and solving of token constraints will be presented in the following two subsections. Finally, we discuss our approach.

3.1 Framework

Algorithm 1 shows the details of the grammar-agnostic DSE framework. The inputs are a parsing program $\mathcal{P}$ and an initial input I_0. The algorithm first employs GenTokenSummary (Algorithm 2) to extract the summary of the tokenization method, *i.e.*, collecting the token value constraints (Line 2), where M_t is the tokenization method in $\mathcal{P}$. Then, the algorithm maintains two worklists $\mathcal{W}_t$ and $\mathcal{W}_a$ to store the off-the-path branches for grammar checking code and the application logic code, respectively.

The main loop is a worklist based procedure. The algorithm first carries out the concolic execution of $\mathcal{P}$ under the current input I (Line 6). This execution returns two path constraints: PC_T and PC_A, *i.e.*, the token path constraint and the byte-level path constraint collected in the application logic code. Then, we save the open off-the-path branches of each path constraint to the corresponding worklist (Lines 7&8). openBranches(PC) is defined as follows, where $PC = \bigwedge_{i=1}^{n} C_i$ and b_i is the branch of each C_i.

$$\{\neg b_i \mapsto (\bigwedge_{j=1}^{i-1} C_j) \wedge \neg C_i \mid 1 \leq i \leq n \wedge \neg b_i \textit{ is not explored}\} \quad (1)$$

Algorithm 1: Grammar-Agnostic Dynamic Symbolic Execution

GADSE($\mathcal{P}, I_0$)
Data: $\mathcal{P}$ is a program, I_0 is the initial input.
1 **begin**
2 $\quad \mathcal{TC} \leftarrow$ GenTokenSummary($\mathcal{P}, M_t$)
3 $\quad \mathcal{W}_t, \mathcal{W}_a \leftarrow \emptyset, \emptyset$
4 $\quad I \leftarrow I_0$
5 $\quad$ **while** *true* **do**
6 $\quad\quad (PC_T, PC_A) \leftarrow$ concolic_execute($\mathcal{P}, I$)
7 $\quad\quad \mathcal{W}_t \leftarrow \mathcal{W}_t \cup$ openBranches(PC_T)
8 $\quad\quad \mathcal{W}_a \leftarrow \mathcal{W}_a \cup$ openBranches(PC_A)
9 $\quad\quad$ **while** $\mathcal{W}_a \neq \emptyset$ **do**
10 $\quad\quad\quad PC_a^c \leftarrow$ Select$_a$($\mathcal{W}_a$)
11 $\quad\quad\quad I \leftarrow$ TokenSolve($\mathcal{TC}, PC_T, PC_a^c$)
12 $\quad\quad\quad (PC_T, PC_A) \leftarrow$ concolic_execute($\mathcal{P}, I$)
13 $\quad\quad\quad \mathcal{W}_a \leftarrow \mathcal{W}_a \cup$ openBranches(PC_A)
14 $\quad\quad$ **end**
15 $\quad\quad$ **if** $\mathcal{W}_t = \emptyset$ **then**
16 $\quad\quad\quad$ **return**
17 $\quad\quad$ **end**
18 $\quad\quad PC_t^c \leftarrow$ Select$_t$($\mathcal{W}_t$) //token-level path exploration
19 $\quad\quad I \leftarrow$ TokenSolve($\mathcal{TC}, PC_t^c, true$)
20 $\quad$ **end**
21 **end**

Algorithm 2: Tokenization Code Summary Generation

GenTokenSummary($\mathcal{P}, M_t$)
Data: $\mathcal{P}$ is a program, M_t is the tokenization method.

1 **begin**
2 $\quad \mathcal{W} \leftarrow \emptyset$
3 $\quad \mathcal{M} \leftarrow \emptyset$
4 $\quad$ **for** $i \in [1..K]$ **do**
5 $\quad\quad I \leftarrow$ RandomInput(i)
6 $\quad\quad$ **while** *true* **do**
7 $\quad\quad\quad (t, PC) \leftarrow$ token_concolic_execution($\mathcal{P}, M_t, I$)
8 $\quad\quad\quad \mathcal{M}[t] \leftarrow t \hat{\in} \mathcal{M}$? $(\mathcal{M}[t] \vee PC) : PC$
9 $\quad\quad\quad \mathcal{W} \leftarrow \mathcal{W} \cup$ openBranches(PC)
10 $\quad\quad\quad$ **if** $\mathcal{W} = \emptyset$ **then**
11 $\quad\quad\quad\quad$ **break**
12 $\quad\quad\quad$ **end**
13 $\quad\quad\quad PC_n \leftarrow \text{Select}_c(\mathcal{W})$
14 $\quad\quad\quad I \leftarrow$ SMTSolve(PC_n)
15 $\quad\quad$ **end**
16 $\quad$ **end**
17 $\quad$ **return** $\mathcal{M}$
18 **end**

Then, we select an off-the-path branch from $\mathcal{W}_a$ (Line 10), where Select_a represents the search heuristic used for path exploration in the application logic code. The selected branch will be removed from $\mathcal{W}_a$. Next, the algorithm solves the selected branch's path condition and the current token path constraint by TokenSolve (Algorithm 3). The new input is used for the next concolic execution of $\mathcal{P}$, and the algorithm only saves the off-the-path branches collected in the application logic code to $\mathcal{W}_a$ (Line 13) because of the same token path constraint.

After the path exploration of the application logic under the current token path constraint PC_t, the algorithm selects an off-the-path branch from $\mathcal{W}_t$ (Line 18). It generates a new input from a new token sequence (Line 19), where Select_t denotes the search heuristic of the token-based exploration for grammar checking code. This procedure continues until there is no off-the-path branch in $\mathcal{W}_t$ or timeout (omitted in both loops for the sake of brevity).

3.2 Tokenization Code Summary Generation

Algorithm 2 shows the details of the first stage for extracting the summary of the tokenization method by collecting token value constraints. The inputs are the program $\mathcal{P}$ and its tokenization method. The output is a map that gives the byte-level constraints for each token value.

The algorithm analyzes $\mathcal{P}$'s tokenization code under different input sizes. The algorithm starts from one size input and generates a random input of the current input size (Line 5). Then, the algorithm uses the input as the initial one for doing DSE. The concolic execution of $\mathcal{P}$ for collecting token value constraints (denoted as token_concolic_execution) terminates when the tokenization method M_t returns and collects the returned concrete value t and the current path condition PC. The algorithm then records t and PC (Line 7). If the token value already exists in $\mathcal{M}$ (denoted as $t \hat{\in} \mathcal{M}$), *e.g.*, generated by the before inputs, the algorithm makes a disjunction between the existing constraint and the current path constraint, which denotes the multiple cases of the same token value. The DSE for collecting the constraints continues until the path exploration under specifically sized input is finished or timeout (omitted for brevity). SMTSolver(PC) represents employing the underlying SMT solver to solve a path constraint PC for generating an input. Finally, $\mathcal{M}$ is returned as a summary of input and token output relation for the tokenization method.

Algorithm 3: Token Constraint Solving

TokenSolve($\mathcal{TC}, PC_T, PC_A$)
Data: $\mathcal{TC}$ is the token constraint map, PC_T is the token constraint, and PC_A is the constraint in application logic.

1 **begin**
2 $\quad V_t \leftarrow$ TokenVars(PC_T)
3 $\quad \mathcal{S} \leftarrow$ SMTSolve(PC_T)
4 $\quad \Phi \leftarrow$ *true*
5 $\quad$ **for** *each* $t_i \in V_t$ **do**
6 $\quad\quad value_t \leftarrow \mathcal{S}[t_i]$
7 $\quad\quad \Phi \leftarrow \Phi \wedge \alpha(\mathcal{TC}[value_t])$
8 $\quad$ **end**
9 $\quad I \leftarrow$ SMTSolve($\Phi \wedge PC_A$)
10 $\quad$ **return** I
11 **end**

In compositional symbolic execution [9][18], the completeness of the function-level summaries directly influences the efficiency of symbolic execution; however, extracting more detailed function-level summaries may introduce more overhead. Similarly, in Algorithm 2, if K is larger, the collection of the token values and the constraints is more complete; however, the first stage's overhead would be larger. There is a trade-off between the first stage's overhead and the whole framework's effectiveness, and K controls this trade-off. Consider the example program in Section 2. If K is 1, we only get the token values `T_ID` and `T_OP`, but we cannot get `T_NUM` that requires the input of size two.

3.3 Token Constraint Solving

Algorithm 3 shows the details of solving token constraints together with the constraint in application logic code. The inputs are the token constraint map $\mathcal{TC}$ generated at the first stage, the token path condition PC_T and the path condition PC_A in the application logic code. The output is the generated input.

The key idea is to solve the token path constraint PC_T to get the token values first (Line 2). Then, based on the token sequence and each token's constraint in $\mathcal{TS}$, the algorithm composes the token constraint of each token value together to form the byte-level constraint Φ to generate the input (Lines 5-8), which is in a similar way of composing system-level constraints and function-level summaries in compositional symbolic execution. Finally, Φ and PC_A will be solved to generate the new program input.

Notably, the conjunction at Line 7 needs to consider the byte index. The token constraint in $\mathcal{TC}$ is just a template constraint for generating the token value. We need to replace the byte variables in the template constraint with the byte variables in the token sequence's new input. For example, for the example program in Section 2, suppose we need to generate the input for the token sequence $\langle$T_NUM, T_OP, T_NUM$\rangle$. For the second token, its constraint in $\mathcal{TC}$ is the following one.

$$t[0] \geq \text{'}*\text{'} \wedge t[0] \leq \text{'+'}$$

Because there are already two bytes for the first token T_NUM, we need to replace $t[0]$ with $a[2]$, and the real constraint added to Φ is the following one.

$$a[2] \geq \text{'}*\text{'} \wedge a[2] \leq \text{'+'}$$

We use $\alpha(\mathcal{TC}[value_t])$ at Line 7 to represent the renamed constraint of $\mathcal{TC}[value_t]$.

3.4 Discussion

In principle, token symbolization is the key to our grammar-agnostic DSE. Token provides a balanced abstraction for the symbolic execution of complex parsing programs. On the one hand, compared with byte-level symbolization, token symbolization-based path constraints can be used to generate different token sequences, which is more effective for testing grammar checking code. On the other hand, tokenization is widely adopted in parsing programs (*e.g.*, the benchmark program in Section 4).

In the second stage of our framework, the path explorations of the grammar checking code and the application logic code are interleaved. We explore the paths of application logic code in priority under the condition of a specific token sequence. After exploring all the application logic paths under the token sequence, the framework generates a new token sequence, which may cover new application logic code. This interleaving divides the program's path space with respect to the input grammar.

Different aspects influence the effectiveness and efficiency of grammar-agnostic DSE. First, the first stage's completeness of collecting token constraints has a direct impact. Some tokens may need a larger-size input, which may introduce a huge overhead for ensuring completeness, but this situation is rare in practice. Second, the search strategies of different stages may also have an influence. Third, the initial input's size (or the length of initial token sequences) also directly influences the DSE's results. As demonstrated by Section 2, our grammar-agnostic DSE can explore more paths than byte-level symbolization because the path space of the same token length is larger than that of the same input size. However, if some program behavior can only be triggered by a specific length of tokens, our approach may fail. Gradually increasing the length of the token sequence can help this situation.

Our method can be understood as an instance of compositional symbolic execution [9][18] targeting parsing programs. Usually, compositional symbolic execution extracts a summary (*e.g.*, input-output relation) of a method first. It then reuses the summary when invoking the method during symbolic execution to avoid entering the method multiple times. This reusing can effectively reduce the program's path space. Our method's first stage collects the input constraint for token values, which extracts a summary of the tokenization method. Similar to compositional symbolic execution's avoiding the multiple executions of a method, the second stage also does not collect the byte-level constraints of the tokenization method. The path exploration at the token level can also be understood as the system-level path exploration in compositional symbolic execution. Besides, the solving method for token constraints stitches the token-level constraints and the tokenization summary to form the byte-level constraint. However, we do not summarize the functions in the grammar checking code or the application logic code. We believe that the compositional symbolic execution in these two parts can further improve the efficiency.

4 EVALUATION

We have implemented our method on the JPF-based DSE engine [17, 27, 38] for Java programs. We have extended the engine to maintain two symbolic execution trees for token-based path space and the byte-level symbolization-based path space in application logic, respectively. We employ JPF-nhandler [31] to handle the invocations of Java Native Interface (JNI), which improves the engine's ability to analyze real-world Java programs. We have improved JPF's environment model libraries for collecting the path constraints better. The engine records the inputs generated during the DSE procedure for the coverage calculation. Besides the input values, we also record the time of generating the inputs.

We conducted extensive experiments to answer the following two research questions.

- **RQ1**: effectiveness, *i.e.*, how effective is our method to test a parsing program compared with byte-level symbolization method and the state-of-the-art fuzzing methods? Here, effectiveness means the number of detected unknown bugs or the statement/branch coverage.
- **RQ2**: efficiency, *i.e.*, how efficient is our method compared with the byte-level symbolization method and the fuzzing methods? Here, we use the time to achieve the same code coverage or find the same bugs to measure efficiency.

4.1 Experimental Setup

Benchmarks. Table 1 lists the benchmark programs used for evaluation. All the benchmark programs are open-source programs that are parsers or have a parsing component. The input grammars of most programs are complex, and the parsing code contains tokenization and grammar checking. The input grammars of these programs are diverse. There are 11 types of grammars, and the number of tokens ranges from 5 to 128 .

Baseline. We compare our method (denoted as **Gadse**) with the baseline DSE method employing byte-level symbolization (denoted as **Char**) under two search heuristics, *i.e.*, DFS and BFS. We use the search strategy in both token constraint collection (Algorithm 2) and the later DSE for grammar checking and application logic code. The value of K (*i.e.*, maximum size of the characters in a token) in Algorithm 2 is set to 3. To evaluate our method further, we also compare our method with two *state-of-the-art* fuzzing methods: coverage-guided fuzzing [26] (denoted as **JQF**) and grammar-guided black-box fuzzing [14] (denoted as **Gramma**).

Evaluation metric. We first record the inputs generated by DSE-based methods or fuzzing methods and then execute the program under the inputs to calculate the statement coverage and branch coverage. We use JaCoCo [15] for coverage calculation. We carry

Table 1: The benchmark Java programs.

Subject	SLOC	Brief Description
`Clojure`	3269	A Clojure parser
`FirstOrder`	2103	A parser for first-order logic
`JsonParser`	4428	JavaCC-built JSON Parser
`J2Latex`	9723	A compiler from Java to Latex
`SiXpath`	6313	An XPath parser
`Aejcc`	3269	Arithmetic Expression interpreter
`Jsijcc`	6313	Javascript interpreter
`FastJSON`	19307	Alibaba JSON parser
`Bling`	3269	parser for arithmetic expressions
`Calculator`	3420	arithmetic expression evaluator
`HtmlParser`	2737	A HTML parser
`UriParser`	2720	An URI parser
`Jsonmwn`	3371	A JSON parser
`OaJava`	15907	A Java code parser
`JavaParser`	22372	Java 1-15 Parser
`CMMParser`	3420	A parser for a subset of C
`Curta`	4428	A expression evaluator
`SqlParser`	1791	A SQL parser
`JsonRaupachz`	3371	A JSON parser
Total	121531	**19 open source Java programs**

out each test generation task for 1 hour and collect the trend of coverage, except the grammar-guided method, which only needs a little time to generate the inputs with respect to the grammar. For some programs, **Gramma**'s prototype does not support the input grammars. We create the generator for the input grammars according to **Gramma**'s document [13]. Because **Grammar** is a black-box grammar-based fuzzer and does not analyze the program, we do not compare with **Grammar** when evaluating the efficiency. For each grammar in the benchmark programs, **Grammar**'s input generation uses less than 20 minutes.

All the experiments were carried out on a server with 64 GB memory and 16 3.1GHz cores. The operating system is Ubuntu 14.04.

4.2 Experimental Results

Answer to RQ1. To answer the the first question, we evaluate **Gadse** by comparing with **Char**, **JQF** and **Gramma** in two aspects: unknown bug detection and code coverage. Next, we give the experimental results.

Unknown bugs. **Gadse** detects 6 *unknown bugs* in the benchmark programs. Table 2 shows the results of bug detection. We only show whether **Grammar** can find the bug because **Grammar** is a black-box grammar fuzzing tool and does not need to analyze the program. All the bugs are caused by runtime exceptions[1].

- **Bug 1**: **Gadse** detected a bug in `J2latex` that causes the runtime exception `NumberFormatException` at the `unary` function in the project's `C1` class. **Gadse** generates the input that contains `"0L"`, which is interpreted by the translator as an octal number and use `Integer.parseInt` to parse the string.
- **Bugs 2&3**: **Gadse** detected two bugs in `CMMParser` that cause `NullPointerException` and `NumberFormatException` exceptions. The first one is in the `polynomial` function of the `CMMParser` class. The reason is that the input statement string passes the grammar checking, but the statement uses an undefined variable, resulting in `NullPointerException`. The second one is in the `term` function of the `CMMParser` class, and the reason is that the generated input causes the parser to convert a string to a floating-point object. However, the string is the concatenation of `"-"` and the null pointer, *i.e.*, `"-null"`, which results in the exception. An undefined variable also causes the null pointer.
- **Bugs 4&5&6**: **Gadse** detected three bugs in `JSInterpreter`. All the bugs are in the evaluator class `EvaluationVisitor` of the program. The first one causes `NullPointerException` in the visit function of an assignment expression. **Gadse** generates an input in which there is an assignment that assigns an undefined variable. The second one causes the `ClassCastException` in the visit function of additive expression. The fault is a programming mistake. The last one also causes the `ClassCastException`. The bug is in the `getDouble` method in `JavascriptType` class. The reason is that the generated input makes the interpreter convert a double value from a non-numerical object.

Char and **JQF** can find only one bug in one hour. **Grammar** can find only three bugs. There is one bug (*i.e.*, Bug 4) that is only found by **Gadse**. **Char** and **JQF** generate many invalid inputs, and **Grammar** can generate valid inputs but does not do well in exploring the path space of application logic. All the bugs can be triggered by the inputs that are valid with respect to the input grammars, which indicate that passing the grammar checking is very important for testing complex parsing programs. Besides, only passing the grammar checking is not enough, and the exploration of the paths in application logic code is also important. These results indicate that **Gadse** is effective in bug detection.

Code coverage. Table 3 shows the detailed coverage results. Figures 3&4 show the comparison results of new statements and branches in DFS between **Char** and **Gadse**, respectively. The X-axis shows the benchmark programs ordered by the values in Y-axis. The Y-axis shows the relative increasing of the covered statements or branches, which is defined as follows, where $N_{\textbf{Gadse}}$ and $N_{\textbf{Char}}$ denote the numbers of statements or branches explored by **Gadse** and **Char**, respectively.

$$\frac{N_{\textbf{Gadse}} - N_{\textbf{Char}}}{N_{\textbf{Char}}} \tag{2}$$

As shown by the figures, under DFS, **Gadse** can explore more statements than **Char** in 17 (89.47%) programs. On average, the relative increasing of statements achieved by **Gadse** is 31.18% (-0.24%~59.18%). For branch coverage, **Gadse** preforms better in the same number of programs as statement coverage, and achieves the relative increasing of branches as 48.41% (0.0%~93.3%) on average. It indicates that **Gadse** improves the effectiveness of DSE. Besides, the improvements of statements and branches are co-related.

[1]All the buggy programs and the inputs that generated by **Gadse** to trigger the bugs are available at https://github.com/gadse-bug/bugs .

Table 2: The results of unknown detected bugs. The number is the time for finding the bug in seconds. >1h means that the method fails to find the bug within 1 hour. Yes in the column GRAMMAR represents that GRAMMAR finds the bug, and No means that GRAMMAR fails to find the bug.

Name	Project	Type	GADSE	CHAR	JQF	GRAMMAR
Bug 1	`J2latex`	`NumberFormatException`	143s	>1h	185s	No
Bug 2	`CMMParser`	`NullPointerException`	36s	>1h	>1h	Yes
Bug 3	`CMMParser`	`NumberFormatException`	41s	>1h	>1h	Yes
Bug 4	`Jsijcc`	`NullPointerException`	456s	>1h	>1h	No
Bug 5	`Jsijcc`	`ClassCastException`	163s	77s	>1h	No
Bug 6	`Jsijcc`	`ClassCastException`	44s	>1h	>1h	Yes

Table 3: Experimental Results of Code Coverage (#S: the number of statements, #B: the number of branches, #P: the number of paths).

Program	Strategy	CHAR			GADSE			JQF			GRAMMAR		
		#S	#B	#P	#S	#B	#P	#S	#B	#P	#S	#B	#P
`Clojure`	BFS	1272	943	16838	1247	939	22628	1217	890	1210504	1182	833	2115
	DFS	1119	794	7879	1278	952	17426						
`FirstOrder`	BFS	538	214	18534	565	220	15586	549	220	2629163	497	179	8571
	DFS	545	208	1710	565	220	11722						
`JsonParser`	BFS	434	230	27211	497	264	30359	429	223	1989811	455	229	373
	DFS	408	192	8684	497	264	30471						
`J2Latex`	BFS	1755	948	230	2433	1500	14343	2677	1704	1381899	2334	1435	54005
	DFS	1710	925	660	2616	1724	13924						
`SiXpath`	BFS	1588	954	10193	1702	1055	4139	1048	484	2380125	1879	1125	675
	DFS	1569	896	1941	1734	1073	6802						
`Aejcc`	BFS	313	113	22302	335	125	22318	314	119	2731671	193	53	7
	DFS	323	119	12266	335	125	18387						
`Jsijcc`	BFS	2553	1302	2435	3172	1755	17426	2738	1538	424095	3674	2080	72887
	DFS	1999	880	10	3036	1701	15792						
`FastJSON`	BFS	1239	475	12296	1642	635	3325	1602	561	2026154	1567	512	373
	DFS	1144	436	7375	1821	704	3146						
`Bling`	BFS	408	151	25591	413	157	28922	422	160	2326562	311	100	12
	DFS	385	140	15960	413	157	27356						
`Calculator`	BFS	335	121	321	354	130	134	321	122	125495	194	59	1
	DFS	335	121	321	354	130	149						
`HtmlParser`	BFS	565	342	12166	579	351	32044	506	269	123352	360	151	68
	DFS	504	262	6710	553	316	31347						
`UriParser`	BFS	702	350	12808	707	340	2615	802	368	40956	795	345	1372
	DFS	619	258	1917	707	340	2531						
`Jsonmwn`	BFS	779	443	22745	842	444	29355	871	517	1694878	665	298	373
	DFS	699	344	369	845	445	29274						
`OaJava`	BFS	2138	1041	13461	3862	1908	32315	3562	1954	1424084	3839	1890	54005
	DFS	2241	945	395	3287	1596	17347						
`JavaParser`	BFS	2213	1190	6821	3464	2033	16337	3285	2032	988016	3932	2329	54005
	DFS	2123	1014	883	3146	1882	16265						
`CMMParser`	BFS	793	469	751	1239	839	9352	912	520	34739	1252	801	2802
	DFS	995	598	704	1273	859	4608						
`Curta`	BFS	1313	616	26150	1262	579	27868	1244	591	1875713	1048	424	3287
	DFS	1182	548	5920	1290	596	13260						
`SqlParser`	BFS	472	224	9390	491	230	6082	490	242	1825278	373	166	45229
	DFS	477	228	888	491	230	6082						
`JsonRaupachz`	BFS	410	189	19782	423	194	27946	420	197	1756342	413	190	373
	DFS	424	194	3843	423	194	24676						

Similar to DFS, Figures 5&6 show the results under BFS. **GADSE** achieves better results for statement coverage and branch coverage in 17 and 16 programs under BFS, respectively. On average, **GADSE** achieves 27.29% (-3.88%∼ 80.64%) relative increasing of statements, and 32.80% (-6.01%∼83.29%) relative increasing of branches. These

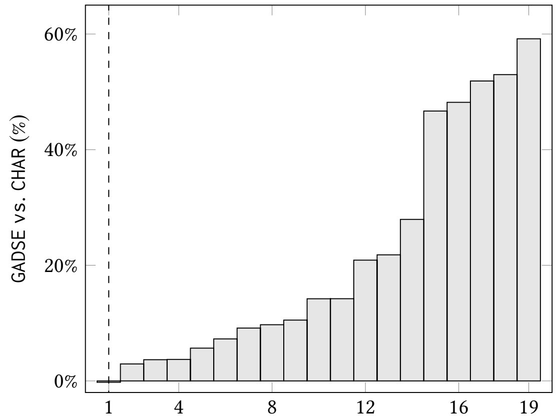

Figure 3: Relative increasing of statement coverage in DFS.

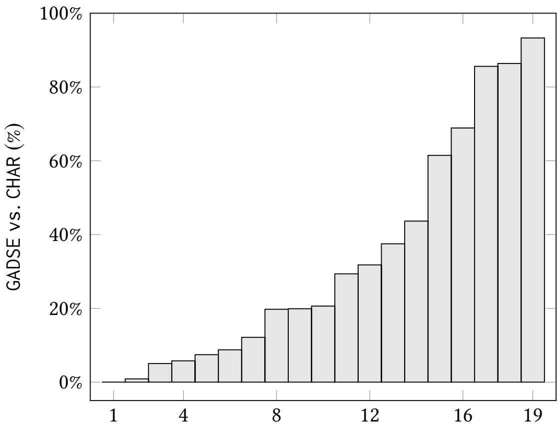

Figure 4: Relative increasing of branch coverage in DFS.

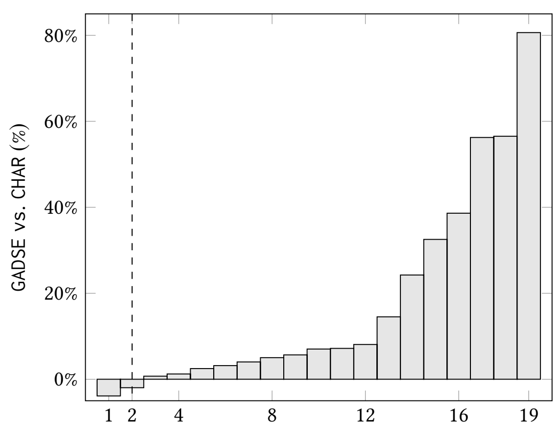

Figure 5: Relative increasing of statement coverage in BFS.

results indicate that **Gadse** is also effective under BFS. Besides, for the benchmark programs, **Gadse** is more effective under DFS.

Gadse also outperforms the two fuzzing methods (*i.e.*, **JQF** and **Grammar**) in many benchmark programs. Compared with **JQF**,

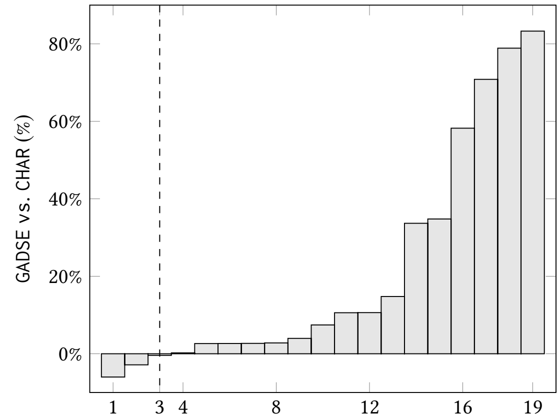

Figure 6: Relative increasing of branch coverage in BFS.

Gadse under DFS on average increases the numbers of statements and branches by 5.36% (-11.85%∼65.46%) and 6.27% (-18.32%∼121.69%), respectively. Compared with **Grammar**, these two results of statements and branches are 17.94% (-17.36%∼82.47%) and 37.36% (-18.22%∼135.84%), respectively.

Similar to DFS, under BFS, **Gadse** also on average performs better than **JQF** and **Grammar**. On average, **Gadse** increases 7.77% (-11.85%∼62.40%) and 18.31% (-13.66%∼82.47%) statements for **JQF** and **Grammar**, respectively. The relative increasings of branches are 7.76% (-14.12%∼117.98%) and 36.96% (-15.62%∼135.84%), respectively.

> *Answer to RQ1: Our method finds more unknown bugs in the benchmark programs. Besides, our method increases the numbers of covered statements and branches.*

Answer to RQ2. To answer the second research question, we carried out the experiments of running **Char** and **JQF** much longer for finding the unknown bugs. The results indicate that both of **Char** and **JQF** fail to find the bugs in 6 hours, *i.e.* the same results as those of 1 hour. **Gadse** finds each bug in less than 8 minutes. These results indicate that **Gadse** is efficient for bug finding.

Besides, we record the time of generating inputs and evaluate our method's efficiency by the time to cover the same amount of statements or branches. We synthesize the global trends of the statement and branch coverages of all the benchmark programs. Figures 7&8 show the trends of statement and branch coverages for all the benchmark programs, respectively. The X-axis shows the analysis time. The Y-axis displays the accumulated number of new statements or branches. We do not consider **Grammar** because it is a black-box approach and requires the input grammar.

As shown by Figure 7, **Gadse** under BFS achieves the best results for statement coverage. **Gadse** (BFS) covers 23409 statements (*i.e.*, the amount of the statements covered by **Char** (BFS) in one hour) at 9s and achieves 6.67x speedup. For **JQF**, this speedup is 2.61x. Similar to statement coverage, as shown by Figure 8, **Gadse** (BFS) also achieves the best result on branch coverage. Compared with **Char** (BFS) and **JQF**, **Gadse** (BFS) achieves 30x and 2.61x speedup to have the same coverage, respectively. These results indicate that

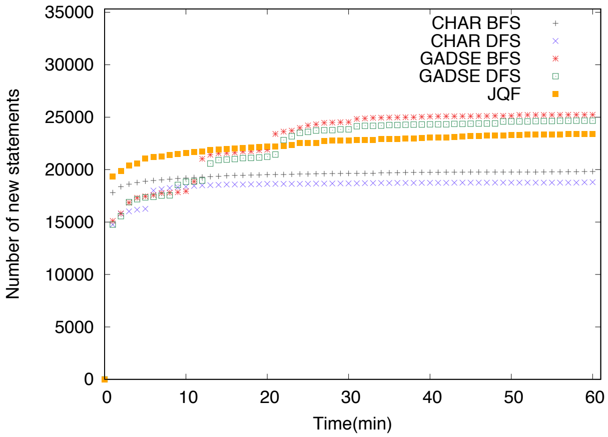

Figure 7: Trends of statement coverage.

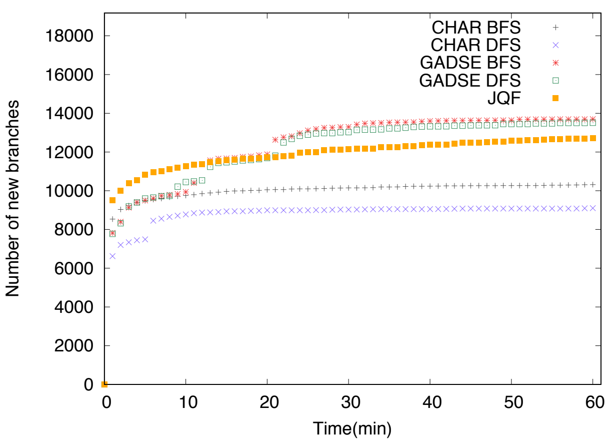

Figure 8: Trends of branch coverage.

GADSE is highly efficient for automatic testing. Therefore, we have the following conclusion for **RQ2**.

In addition, as shown by the figures, compared with byte-level symbolization, **JQF** achieves a better coverage. Besides, **JQF** performs best in the beginning (*i.e.*, before 20 minutes). The reason is that fuzzing is fast and runs the program many times (on average 1420465) in 1 hour to improve the statement or branch coverage. Moreover, these results and the bug finding results also indicate that the coverage improvement is not co-related to bug finding.

Answer to RQ2: Our method finds the unknown bugs in less than 8 minutes; whereas, byte-level symbolization-based DSE or coverage-guided fuzzing fails to find the bugs in 6 hours. Compared with byte-level symbolization, our method, on average, achieves 6.67x and 30x speedups to achieve the same statement and branch coverages, respectively.

4.3 Threats to Validity

The threats to the validity are mainly external. The benchmark Java programs and the grammars are limited. We plan to apply our method to more complex programs in the next step. We alleviate the experimental errors by running each task three times and use the average value as the result. For internal threats, which mainly come from implementation errors, we designed some manually written simple grammar parsing programs (such as the motivation example) to test our prototype.

5 LIMITATIONS

Our grammar-agnostic DSE is limited in the following aspects:

- Our method is not applicable if the parsing program does not employ token-based input grammar checking, *i.e.*, URL parsing, which usually employs regular expressions for parsing and does not use tokenization.
- The separation of the parsing program into different stages needs manual help. Besides, we need the entry information of the tokenization code.
- Our method is limited in its handling stateful tokens. Stateful tokens influence the byte-level constraints of the tokens in the first stage, which may cause path divergence.
- Our method is limited in handling the parsing program with the context-free input grammars. Especially, we may generate the token sequence that does not satisfy the matching requirements in context-free grammars, *e.g.*, `'('` and `')'` should be matched.
- If the application logic code is tightly weaved into the parsing code, our method's advantage may be doomed, especially the ability to explore the paths of application logic code in priority.

The first one is inevitable. For the second one, we can employ a lightweight static analysis method to suggest the separation and the tokenization code of the parsing program. The third one can be supported by employing multiple tokens-based summary during the first stage, which may introduce more overhead. The fourth one is because our method does not need grammar. We suggest developing a search heuristics to select the token constraints that tend to generate valid token sequences. The last one needs more abstractions for improving symbolic execution's efficiency further.

6 RELATED WORK

Our work is related to many research areas, including symbolic execution, fuzzing, grammar inference, *etc.* Next, we review the related work and compare our method with them.

There exist work of leveraging input grammar to improve the efficiency of symbolic execution for parsing programs [10, 23]. Godefroid *et al.* [10] propose grammar-based white-box fuzzing, which also suggests employing token symbolization during the symbolic execution. The token constraint is then solved based on an input grammar. CESE [23] also uses an input grammar to improve the DSE of the grammar's parsing program. CESE generates the initial inputs based on the symbolic grammar generated from the input grammar. These inputs are then used for the DSE of the parsing program to explore the deeper paths. In contrast, our grammar-agnostic DSE does not need to provide an input grammar. We use the token's byte-level constraints collected in the first stage for solving the token path constraints. David *et al.* [8] propose a language for specifying input symbolization, which is critical for the efficiency of symbolic execution. In principle, specifying how to symbolize input usually considers the input grammar.

There exists the work of search heuristics for improving the efficiency of symbolic execution. Different search heuristics are proposed for different targets, such as code coverage [4, 36], reaching a statement [22] and generating a specific program path [40]. Besides, there is also work of pruning program paths [7, 21, 37] to improve efficiency, which prunes the redundant paths with respect to the target, *e.g.*, the paths that do not contribute code coverage or will not trigger bugs. The existing work of search heuristic and path pruning are complementary with our grammar-agnostic DSE. We can employ different heuristics in the different stages of grammar-agnostic DSE. On the other hand, our token symbolization and constraint solving can be considered as exploring the path in application logic code and the valid input-related paths in priority and pruning invalid input paths.

Our method is also related to compositional symbolic execution [2, 9, 18, 29]. To improve DSE's scalability, Godefroid [9] proposes SMART that uses DSE to generate the input-output relation summaries for low-level functions first, and then directly uses the summaries when invoking the functions during the DSE of higher-level functions (*i.e.*, caller functions). Anand *et al.* [2] improves SMART by a demand-driven compositional symbolic execution method, which tries to reduce the explored paths by a lazy summary method based on the encoding using uninterpreted functions [20]. FOCAL [18] advances demand-driven compositional symbolic execution by employing a Craig interpolants [6] based function summary refinement. FOCAL employs a backward analysis to generate a system-level input for a failure target and composes the constraints of the contexts in the target's invoking chain from the entry function. Gillian [29] provides a language-independent compositional symbolic execution framework, in which a bi-abductive symbolic analysis [5] is employed to support compositional testing. Our method is an instance of compositional symbolic execution targeting parsing programs. We only summarize the tokenization code, which balances the generalization and the efficiency for analyzing parsing programs. It is interesting to leverage the result in these work to further improve the efficiency of our method, *e.g.*, in the analysis of the application logic code.

Fuzzing [39] is also related to our work. The existing grammar-oriented fuzzing work can be divided into grammar-directed black-box fuzzing [14], grammar-directed gray-box fuzzing [24, 25], grammar and coverage directed gray-box fuzzing [26]. Havrikov and Zeller [14] use an input grammar to generate program inputs and propose the notion of *token coverage* to guide the generation procedure. Mathis *et al.* [24] propose parser-directed fuzzing, which provides a lightweight approach for recording the character comparisons during parsing and generates the valid input to pass parsing code. To handle the problem of the token comparison in grammar checking, LFuzzer employs a two-stage procedure for fuzzing the parser [25]. LFuzzer collects tokens and their corresponding inputs in the first stage and uses these tokens in the second stage to help the fuzzer generate the inputs that can pass the validity checking of the parser. Superion [34] provides a grammar-aware coverage-based gray-box fuzzing method, in which the grammar is used to minimize and mutate the inputs for improving the fuzzing's efficiency. Zest [26] combines coverage-oriented gray-box fuzzing and grammar-based black-box fuzzing to mutate the inputs more efficiently. Compared with these fuzzing approaches, our approach is symbolic execution-based, which suffers from symbolic computation overhead and enjoys more efficient path exploration. The empirical comparison between our approach and Zest (without grammar generator) in Section 4 indicates that our approach is more effective and efficient for bug finding and code coverage.

Our work is also related to input grammar inference. Glade [3] provides an algorithm that synthesizes a context-free input grammar form the input-output examples of the program. Then, the inferred grammar can be used to improve fuzzing. REINAM [35] improves Glade by tackling the problem of over-generalization. REINAM generates a probabilistic context-free input grammar. Skyfire [33] proposes to learn a probabilistic context-sensitive grammar (PCSG) to represent the distribution of valid inputs. Then the PCSG is used to generate seeds for efficient fuzzing. Different from these approaches, *Mimid* [12] learns a readable context-free input grammar in a white-box manner. The input characters are tracked for their access to aid the grammar inference. How to infer the grammar based on symbolic execution (which provides more information) is interesting and left to be the future work.

7 CONCLUSION

Symbolic execution of complex parsing programs is challenging. This paper presents grammar-agnostic symbolic execution, *i.e.*, a framework that uses token symbolization to improve symbolic execution's efficiency. Our framework does not need to provide input grammar. We automatically collect the input constraints of token values, based on which valid inputs can be generated to test complex parsing programs efficiently. We have implemented our framework for Java programs based on JPF. The extensive experiments indicate that our approach is effective and efficient for testing complex parsing programs.

The next step lies in several directions: 1) improve the prototype to carry out more extensive experiments; 2) investigate the method for generating the inputs of complex grammars; 3) study more advanced symbolic abstraction for testing parsing programs.

ACKNOWLEDGEMENTS

This research was supported by National Key R&D Program of China (No. 2017YFB1001802) and NSFC Program (No. 61632015, 62002107, 62032024, and 61690203).

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

[2] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-Driven Compositional Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 367–381. https://doi.org/10.1007/978-3-540-78800-3_28

[3] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona,*

Spain, June 18-23, 2017, Albert Cohen and Martin T. Vechev (Eds.). ACM, 95–110. https://doi.org/10.1145/3062341.3062349

[4] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224.

[5] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 289–300. https://doi.org/10.1145/1480881.1480917

[6] William Craig. 1957. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic* 22, 3 (1957), 269–285. https://doi.org/10.2307/2963594

[7] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. 2013. Verifying systems rules using rule-directed symbolic execution. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, 329–342. https://doi.org/10.1145/2451116.2451152

[8] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. 2016. Specification of concretization and symbolization policies in symbolic execution. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 36–46. https://doi.org/10.1145/2931037.2931048

[9] Patrice Godefroid. 2007. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 47–54. https://doi.org/10.1145/1190216.1190226

[10] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 206–215. https://doi.org/10.1145/1375581.1375607

[11] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 213–223. https://doi.org/10.1145/1065010.1065036

[12] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 172–183. https://doi.org/10.1145/3368089.3409679

[13] Nikolas Havrikov. 2019. tribble 1.0.0. https://github.com/havrikov/tribble.

[14] Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 189–199. https://doi.org/10.1109/ASE.2019.00027

[15] Marc R Hoffmann, B Janiczak, and E Mandrikov. 2014. JaCoCo, version 0.6.5.201403032054. https://github.com/jacoco/jacoco.

[16] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to automata theory, languages, and computation, 3rd Edition*. Addison-Wesley.

[17] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. 2009. jFuzz: A Concolic Whitebox Fuzzer for Java. In *First NASA Formal Methods Symposium - NFM 2009, Moffett Field, California, USA, April 6-8, 2009 (NASA Conference Proceedings)*, Ewen Denney, Dimitra Giannakopoulou, and Corina S. Pasareanu (Eds.), Vol. NASA/CP-2009-215407. 121–125.

[18] Yunho Kim, Shin Hong, and Moonzoo Kim. 2019. Target-driven compositional concolic testing with function summary refinement for effective bug detection. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 16–26. https://doi.org/10.1145/3338906.3338934

[19] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[20] Daniel Kroening and Ofer Strichman. 2016. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Springer. https://doi.org/10.1007/978-3-662-50497-0

[21] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 193–204. https://doi.org/10.1145/2254064.2254088

[22] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science)*, Eran Yahav (Ed.), Vol. 6887. Springer, 95–111. https://doi.org/10.1007/978-3-642-23702-7_11

[23] Rupak Majumdar and Ru-Gang Xu. 2007. Directed test generation using symbolic grammars. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer (Eds.). ACM, 134–143. https://doi.org/10.1145/1321631.1321653

[24] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 548–560. https://doi.org/10.1145/3314221.3314651

[25] Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning input tokens for effective fuzzing. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 27–37. https://doi.org/10.1145/3395363.3397348

[26] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 329–340. https://doi.org/10.1145/3293882.3330576

[27] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. 2008. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, Barbara G. Ryder and Andreas Zeller (Eds.). ACM, 15–26. https://doi.org/10.1145/1390630.1390635

[28] Corina S. Pasareanu and Neha Rungta. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 179–180. https://doi.org/10.1145/1858996.1859035

[29] José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian: Compositional Symbolic Execution for All. *CoRR* abs/2001.05059 (2020). https://arxiv.org/abs/2001.05059

[30] Koushik Sen, Darko Marinov, and Gul Agha. 2005. 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, Lisbon, Portugal, September 5-9, 2005*, Michel Wermelinger and Harald C. Gall (Eds.). ACM, 263–272. https://doi.org/10.1145/1081706.1081750

[31] Nastaran Shafiei and Franck van Breugel. 2014. Automatic handling of native methods in Java PathFinder. In *2014 International Symposium on Model Checking of Software, SPIN 2014, Proceedings, San Jose, CA, USA, July 21-23, 2014*, Neha Rungta and Oksana Tkachuk (Eds.). ACM, 97–100. https://doi.org/10.1145/2632362.2632363

[32] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex-White Box Test Generation for .NET. In *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings (Lecture Notes in Computer Science)*, Bernhard Beckert and Reiner Hähnle (Eds.), Vol. 4966. Springer, 134–153. https://doi.org/10.1007/978-3-540-79124-9_10

[33] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 579–594. https://doi.org/10.1109/SP.2017.23

[34] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[35] Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. 2019. REINAM: reinforcement learning for input-grammar inference. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 488–498. https://doi.org/10.1145/3338906.3338958

[36] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009*. IEEE Computer Society, 359–368. https://doi.org/10.1109/DSN.2009.5270315

[37] Hengbiao Yu, Zhenbang Chen, Ji Wang, Zhendong Su, and Wei Dong. 2018. Symbolic verification of regular properties. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June*

03, 2018, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 871–881. https://doi.org/10.1145/3180155.3180227

[38] Hengbiao Yu, Zhenbang Chen, Yufeng Zhang, Ji Wang, and Wei Dong. 2017. RGSE: a regular property guided symbolic executor for Java. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 954–958. https://doi.org/10.1145/3106237.3122830

[39] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The Fuzzing Book. In *The Fuzzing Book*.

[40] Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular Property Guided Dynamic Symbolic Execution. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 643–653. https://doi.org/10.1109/ICSE.2015.80