Grammar-Agnostic Symbolic Execution by Token Symbolization

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

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

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

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

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

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

KEYWORDS
Symbolic execution, Grammar, Parsing Program, Tokenization

ACM Reference Format:

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
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 a 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.

\[(\text{EXP}) \rightarrow (\text{NUM})(\text{OP})(\text{NUM})\]

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.
We use dynamic symbolic execution (DSE) \cite{30,11} to analyze complex programs. DSE (or concolic execution) combines traditional symbolic execution and concrete execution to analyze a program. Given a program \(P\), the initial input \(I\) and the input’s symbolization strategy, DSE executes \(P\) 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 \(PC(p) = \bigwedge_{i=1}^{n} C_i\) and \(C_i\) is the symbolic condition of the branch \(b_i\), the path condition of \(b_i's\) off-the-path branch (denoted as \(\neg b_i\)) is \(PC(\neg b_i) = (\bigwedge_{j=1}^{i-1} C_j) \land \neg C_i\), where \(1 \leq j \leq n\). 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 \(P\) 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.

### 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. \((\text{Expr})\) is the entry non-terminal. This grammar accepts an expression that can be a single-character name (\(\langle \text{ID} \rangle\)), a two-digit number (\(\langle \text{Number} \rangle\)) 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 \(\langle z' \rangle\). The true branch contains a bug (Line 7). parseExpr implements a recursive descent parser \cite{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. The path condition of the first iteration (denoted as \(PC_1\)) is as follows, where \(a[i]\) represents the \(i\)th character’s symbolic value, and the right-side numbers are the line numbers of the branch conditions that generate the constraints.

\[
\begin{align*}
    a[0] & \geq ' + ' \land a[0] > ' + ' \land a[0] < ' a' \land (36 & 38) \\
    a[0] & \geq ' 0 ' \land a[0] \leq ' 9 ' \land a[1] \geq ' 0 ' \land a[1] \leq ' 9 ' \land (40 & 42) \\
    a[2] & \geq ' + ' \land a[2] \leq ' + ' \land (36) \\
    a[3] & \geq ' 0 ' \land a[3] \leq ' 9 ' \land a[4] \geq ' 0 ' \land a[4] \leq ' 9 ' \land (40 & 42) \\
    a[4] & \neq ' z ' (6)
\end{align*}
\]

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 \(a[4] \neq ' z '\), the path condition for generating the new input would be \(PC_2\), except the last condition is changed to \(a[4] = ' z '\), which is as follows (denoted as \(PC_2\)).

\[
\begin{align*}
    a[0] & \geq ' + ' \land a[0] > ' + ' \land a[0] < ' a' \land (36 & 38) \\
    a[0] & \geq ' 0 ' \land a[0] \leq ' 9 ' \land a[1] \geq ' 0 ' \land a[1] \leq ' 9 ' \land (40 & 42) \\
    a[2] & \geq ' + ' \land a[2] \leq ' + ' \land (36) \\
    a[3] & \geq ' 0 ' \land a[3] \leq ' 9 ' \land a[4] \geq ' 0 ' \land a[4] \leq ' 9 ' \land (40 & 42) \\
    a[4] & = ' z ' (6)
\end{align*}
\]
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.

$$a[0] \geq 3 \land a[0] < 1' \land a[0] < a'$$
$$a[0] \geq 4 \land a[0] < 3' \land a[0] < 9' \land a[1] < 0' \land a[4] < a'$$
$$a[2] \geq 3' \land a[2] < 1' + a'$$

$PC_3$ is satisfiable. Suppose that solving $PC_3$ generates "12+12", 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 ‘\'. However, there do exist valid input strings that can trigger the bug, e.g., "12+\".

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, 

\textit{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 `\texttt{get\_NextToken}`.

In the beginning, the input string’s length is one. The paths explored by the DSE of `\texttt{get\_NextToken}` 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_NUM` and `T_ID`, respectively. Their path constraints are as follows, where $TC[T]$ represents the path constraint of token value $t$, where $t[j]$ represents the $j$th character’s symbolic value in the string represented by $t$.

\[
TC[T\_ID] = \begin{cases} 
0' \land t[0] \leq 'z' \\
9' \land t[0] \leq 'z' \\
'+' \land t[0] \leq 'z'
\end{cases}
\]

\[
TC[T\_OP] = \begin{cases} 
0' \land t[0] \leq 'z' \\
9' \land t[0] \leq 'z' \\
'+' \land t[0] \leq 'z'
\end{cases}
\]

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 `\texttt{get\_NextToken}` 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.

\[
TC[T\_NUM] = \begin{cases} 
0' \land t[0] \leq '9' \land t[1] \geq 0' \land t[1] \leq '9'
\end{cases}
\]

So, after the two times of DSE for `\texttt{get\_NextToken}` again, 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 $TC$ that records the explored token values and their input constraints. $TC$ actually gives a symbolic summary of the method `\texttt{getNextToken}`, \textit{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 $b_1$ from the application logic’s off-the-path branch set $OB_A$ and generates an input by solving the constraint composed by $PC(b_1)$ and the current token constraint $PC_T$, \textit{i.e.}, $PC_T \land PC(b_1)$. 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 $TC$ 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 $PC_T^C \land PC(b_1)$ to generate the new input. If there is no more branches in $OB_A$, the framework selects an off-the-path branch $b_1$ from the grammar checking’s off-the-path branch set $OB_T$ and solves the token constraint $PC(b_1)$ as before (i.e., $PC(b_1)$ 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 $T[i]$ represents the $i$th token’s symbolic value.

\[
\begin{align*}
PC_T &\lor PC_A
\end{align*}
\]

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 $a[4] = 'z'$. Hence, the new path constraint is as follows.

\[
\begin{align*}
PC_T &\lor PC(b_1)
\end{align*}
\]

To solve $PC_T \land PC(b_1)$, we solve $PC_T$ to get a token value sequence, based on which and $TC$ 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 $(T\_NUM,T\_OP,T\_NUM)$. Hence, $PC_T^C$ is as follows.

\[
\begin{align*}
a[0] &\geq 0' \land a[0] \leq 9' \land a[1] \geq 0' \land a[1] \leq 9' \land TC[T\_NUM] \\
a[2] &\geq 0' \land a[2] \leq 9' \land TC[T\_OP] \\
a[3] &\geq 0' \land a[3] \leq 9' \land TC[T\_NUM]
\end{align*}
\]
However, $PC_T^2 \land PC(b_a)$ is unsatisfiable, which means any inputs generating the current token sequence, i.e., $(T_{NUM}, T_{OP}, T_{NUM})$, cannot 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.

$$T[0] = T_{NUM} \land T[1] = T_{OP} \land T[2] \neq T_{NUM}$$

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 $T_C$.

$$\bigwedge_{i=1}^{3} T[i] \in \{T_{ID}, T_{NUM}, T_{OP}\}$$

Solving $PC(b_t) \land PC_R$ 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 $PC(b_t) \land PC_R$ generates the solution in which $T[2]$ is $T_{ID}$. The new token sequence is $(T_{NUM}, T_{OP}, T_{ID})$. Then, the byte-level constraint $PC_C(b_t)$ is as follows.

\[
\begin{align*}
 a[0] &\geq '0' \land a[0] \leq '9' \land a[1] \geq '0' \land a[1] \leq '9' \land T_C[T_{NUM}] \\
 a[2] &\geq '1' \land a[2] \leq '+' \land T_C[T_{OP}] \\
 a[3] &\geq 'a' \land a[3] \leq 'z' \land T_C[T_{ID}] \\
 a[3] &\neq 'z' \\
 a[3] &\neq '2' \\
\end{align*}
\]

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

$$T[0] = T_{NUM} \land T[1] = T_{OP} \land T[2] \neq T_{NUM} \land T[2] = T_{ID} \land
\begin{align*}
 PC_C(b_t)
 a[3] &\neq '2' \\
 a[3] &\neq 'z'
\end{align*}$$

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.

$$T[0] = T_{NUM} \land T[1] = T_{OP} \land T[2] \neq T_{NUM} \land T[2] = T_{ID} \land
\begin{align*}
 PC_T
 a[3] &\geq '0' \land a[0] \leq '9' \land a[1] \geq '0' \land a[1] \leq '9' \land T_C[T_{NUM}] \\
 a[2] &\geq '1' \land a[2] \leq '+' \land T_C[T_{OP}] \\
 a[3] &\geq 'a' \land a[3] \leq 'z' \land T_C[T_{ID}] \\
 a[3] &\neq 'z' \\
 PC_C(b_t)
 a[3] &\neq 'z'
\end{align*}$$

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

\[
\begin{align*}
 a[0] &\geq '0' \land a[0] \leq '9' \land a[1] \geq '0' \land a[1] \leq '9' \land T_C[T_{NUM}] \\
 a[2] &\geq '1' \land a[2] \leq '+' \land T_C[T_{OP}] \\
 a[3] &\geq 'a' \land a[3] \leq 'z' \land T_C[T_{ID}] \\
 a[3] &\neq 'z' \\
\end{align*}
\]

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 $P$ 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 $P$. Then, the algorithm maintains two worklists $W_t$ and $W_a$ 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 $P$ 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 $C_i$.

$$\{ \neg b_i \iff (\bigwedge_{i=1}^n C_i) \land \neg C_i \mid 1 \leq i \leq n \land \neg b_i \text{ is not explored} \} \quad (1)$$

Algorithm 1: Grammar-Agnostic Dynamic Symbolic Execution

**GADSE($P, I_0$)**

**Data:** $P$ is a program, $I_0$ is the initial input.

1. **begin**
2. $T_C \leftarrow $ GenTokenSummary($P, M_t$)
3. $W_t, W_a \leftarrow \emptyset, \emptyset$
4. $I \leftarrow I_0$
5. **while** true **do**
6. $(PC_T, PC_A) \leftarrow $ concolic_execute($P, I$)
7. $W_t \leftarrow W_t \cup \text{openBranches}(PC_T)$
8. $W_a \leftarrow W_a \cup \text{openBranches}(PC_A)$
9. **while** $W_a \neq \emptyset **do**$
10. $PC^*_a \leftarrow $ Select$_a(W_a)$
11. $I \leftarrow $ TokenSolve($T_C, PC_T, PC^*_a$)
12. $(PC_T, PC_A) \leftarrow $ concolic_execute($P, I$)
13. $W_a \leftarrow W_a \cup \text{openBranches}(PC_A)$
14. **end**
15. **if** $W_t = \emptyset **then**$
16. **return**
17. **end**
18. $PC^*_t \leftarrow $ Select$_t(W_t)$ //token-level path exploration
19. $I \leftarrow $ TokenSolve($T_C, PC^*_t, \text{true}$)
20. **end**
21. **end**
Then, we select an off-the-path branch from $\mathcal{W}_t$ (Line 10), where $\text{Select}_{t}$ represents the search heuristic used for path exploration in the application logic code. The selected branch will be removed from $\mathcal{W}_t$. Next, the algorithm selects the tokenized branch's condition and the current path token constraint by $\text{TokenSolve}$ (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 $\mathcal{W}_t$ (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 $\text{Select}_{t}$ 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 for 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 $\text{token\_concolic\_execution}$) 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 $M$ (denoted as $t \in M$), 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). $\text{SMTSolve}(PC)$ represents employing the underlying SMT solver to solve a path constraint $PC$ for generating an input. Finally, $M$ is returned as a summary of input and token output relation for the tokenization method.

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 $TC$ 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 $TS$, the algorithm composes the token constraint of each token value together to form the byte-level constraint $\Phi$ 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, $\Phi$ 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 $TC$ 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 \((T_{\text{NUM}}, T_{\text{OP}}, T_{\text{NUM}})\). For the second token, its constraint in \(TC\) is the following one.

\[ t[0] \geq '0' \land t[0] \leq '+' \]

Because there are already two bytes for the first token \(T_{\text{NUM}}\), we need to replace \(t[0]\) with \(a[2]\), and the real constraint added to \(\Phi\) is the following one.

\[ a[2] \geq '0' \land a[2] \leq '+' \]

We use \(a(TC[\text{value}])\) at Line 7 to represent the renamed constraint of \(TC[\text{value}]\).

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 other 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.

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

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, Grammar’s prototype does not support the input grammars. We create the generator for the input grammars according to Grammar’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 first question, we evaluate Gadse by comparing with Char, JQF and Grammar 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.

- 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 "8L", 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 JSIccParser. All the bugs are in the code generation 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 two bugs. One is a 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 the grammar checking is 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 \(N_{Gadse}\) and \(N_{Char}\) denote the numbers of statements or branches explored by Gadse and Char, respectively.

\[
\frac{N_{Gadse} - N_{Char}}{N_{Char}}
\]  

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 performs 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.
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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Project</th>
<th>Type</th>
<th>Gadse</th>
<th>Char</th>
<th>JQF</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug 1</td>
<td>J2latex</td>
<td>NumberFormatException</td>
<td>143s</td>
<td>&gt;1h</td>
<td>185s</td>
<td>No</td>
</tr>
<tr>
<td>Bug 2</td>
<td>CMMParser</td>
<td>NullPointerException</td>
<td>36s</td>
<td>&gt;1h</td>
<td>&gt;1h</td>
<td>Yes</td>
</tr>
<tr>
<td>Bug 3</td>
<td>CMMParser</td>
<td>NumberFormatException</td>
<td>41s</td>
<td>&gt;1h</td>
<td>&gt;1h</td>
<td>Yes</td>
</tr>
<tr>
<td>Bug 4</td>
<td>JsiJcc</td>
<td>NullPointerException</td>
<td>163s</td>
<td>77s</td>
<td>&gt;1h</td>
<td>No</td>
</tr>
<tr>
<td>Bug 5</td>
<td>JsiJcc</td>
<td>ClassCastException</td>
<td>44s</td>
<td>&gt;1h</td>
<td>&gt;1h</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Strategy</th>
<th>Char</th>
<th>Gadse</th>
<th>JQF</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clojure</td>
<td>BFS</td>
<td>1272</td>
<td>943</td>
<td>1247</td>
<td>939</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>1119</td>
<td>794</td>
<td>1278</td>
<td>952</td>
</tr>
<tr>
<td>FirstOrder</td>
<td>BFS</td>
<td>538</td>
<td>214</td>
<td>565</td>
<td>220</td>
</tr>
<tr>
<td>JsonParser</td>
<td>BFS</td>
<td>434</td>
<td>230</td>
<td>497</td>
<td>264</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>408</td>
<td>208</td>
<td>497</td>
<td>264</td>
</tr>
<tr>
<td>J2latex</td>
<td>BFS</td>
<td>1735</td>
<td>948</td>
<td>2433</td>
<td>1590</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>1710</td>
<td>925</td>
<td>2616</td>
<td>1724</td>
</tr>
<tr>
<td>SiXPath</td>
<td>BFS</td>
<td>1588</td>
<td>954</td>
<td>1704</td>
<td>1055</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>1569</td>
<td>896</td>
<td>1925</td>
<td>1073</td>
</tr>
<tr>
<td>AeJcc</td>
<td>BFS</td>
<td>313</td>
<td>113</td>
<td>335</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>323</td>
<td>119</td>
<td>335</td>
<td>125</td>
</tr>
<tr>
<td>JsiJcc</td>
<td>BFS</td>
<td>2553</td>
<td>1302</td>
<td>3172</td>
<td>1755</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>1999</td>
<td>880</td>
<td>10</td>
<td>3036</td>
</tr>
<tr>
<td>FastJSON</td>
<td>BFS</td>
<td>1237</td>
<td>475</td>
<td>1642</td>
<td>635</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>1144</td>
<td>436</td>
<td>1821</td>
<td>704</td>
</tr>
<tr>
<td>BING</td>
<td>BFS</td>
<td>408</td>
<td>151</td>
<td>413</td>
<td>157</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>385</td>
<td>140</td>
<td>413</td>
<td>157</td>
</tr>
<tr>
<td>Calculator</td>
<td>BFS</td>
<td>313</td>
<td>121</td>
<td>335</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>313</td>
<td>121</td>
<td>354</td>
<td>130</td>
</tr>
<tr>
<td>HtmlParser</td>
<td>BFS</td>
<td>565</td>
<td>342</td>
<td>579</td>
<td>351</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>504</td>
<td>262</td>
<td>553</td>
<td>316</td>
</tr>
<tr>
<td>UriParser</td>
<td>BFS</td>
<td>702</td>
<td>330</td>
<td>707</td>
<td>340</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>619</td>
<td>258</td>
<td>707</td>
<td>340</td>
</tr>
<tr>
<td>JsonMun</td>
<td>BFS</td>
<td>779</td>
<td>443</td>
<td>842</td>
<td>444</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>699</td>
<td>344</td>
<td>845</td>
<td>445</td>
</tr>
<tr>
<td>QaJava</td>
<td>BFS</td>
<td>2138</td>
<td>1910</td>
<td>3862</td>
<td>1908</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>2241</td>
<td>1945</td>
<td>395</td>
<td>3287</td>
</tr>
<tr>
<td>JavaParser</td>
<td>BFS</td>
<td>2213</td>
<td>1190</td>
<td>3464</td>
<td>2033</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>2123</td>
<td>1014</td>
<td>3146</td>
<td>1882</td>
</tr>
<tr>
<td>CMMParser</td>
<td>BFS</td>
<td>793</td>
<td>469</td>
<td>1239</td>
<td>839</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>995</td>
<td>598</td>
<td>1273</td>
<td>839</td>
</tr>
<tr>
<td>Curta</td>
<td>BFS</td>
<td>1313</td>
<td>616</td>
<td>1262</td>
<td>579</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>1182</td>
<td>548</td>
<td>1290</td>
<td>596</td>
</tr>
<tr>
<td>SqlParser</td>
<td>BFS</td>
<td>472</td>
<td>224</td>
<td>491</td>
<td>230</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>477</td>
<td>228</td>
<td>491</td>
<td>230</td>
</tr>
<tr>
<td>JsonRaupachz</td>
<td>BFS</td>
<td>410</td>
<td>189</td>
<td>423</td>
<td>194</td>
</tr>
<tr>
<td></td>
<td>DFS</td>
<td>424</td>
<td>194</td>
<td>423</td>
<td>194</td>
</tr>
</tbody>
</table>

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
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,
**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]. Godefrois 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 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.
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]. Havrićk 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. 616902015, 62002107, 62032024, and 6190203).

REFERENCES