Grammar-based Whitebox Fuzzing

Patrice Godefroid
Microsoft Research
pg@microsoft.com

Adam Kieżun *
Massachusetts Institute of Technology
akiezun@mit.edu

Michael Y. Levin
Microsoft Center for Software Excellence
mlevin@microsoft.com

Abstract

Whitebox fuzzing is a form of automatic dynamic test generation, based on symbolic execution and constraint solving, designed for security testing of large applications. However, the effectiveness of whitebox fuzzing is limited when testing applications with highly-structured inputs, such as compilers and interpreters. These applications process their inputs in stages, such as lexing, parsing and evaluation. Due to the enormous number of control paths in the early processing stages, whitebox fuzzing rarely reaches parts of the application beyond the first stages.

In this paper, we study how to enhance whitebox fuzzing of complex structured-input applications with a grammar-based specification of their valid inputs. We present a novel dynamic test generation algorithm where symbolic execution directly generates grammar-based constraints whose satisfiability is checked using a custom grammar-based constraint solver. We have implemented this algorithm and evaluated it on a large security-critical application, the JavaScript interpreter of Internet Explorer 7. Results of experiments show that grammar-based whitebox fuzzing explores deeper program paths and avoids dead-ends due to non-parsable inputs. Compared to regular whitebox fuzzing, grammar-based whitebox fuzzing increased coverage of the code generation module of the IE7 JavaScript interpreter from 53% to 81% while using three times fewer tests.

1. Introduction

Blackbox fuzzing is a form of testing, heavily used for finding security vulnerabilities in software. It simply consists in randomly modifying well-formed inputs and testing the resulting variants [13, 3, 12]. Blackbox fuzzing sometimes uses *grammars* to generate the well-formed inputs, as well as to encode application-specific knowledge and test heuristics for guiding the generation of input variants [30, 1, 29].

A recently proposed alternative, *whitebox fuzzing* [16], combines fuzz testing with dynamic test generation [15, 6]. Whitebox fuzzing executes the program under test with an initial, well-formed input, both concretely and symbolically. During the execution of conditional statements, whitebox fuzzing creates constraints on program inputs. Those constraints capture how the program uses its inputs, and satisfying assignments for the negation of each constraint define new inputs that exercise different control paths. Whitebox fuzzing repeats this process for the newly created inputs, to exercise all feasible control paths of the program under test. In practice, the search is usually incomplete because the number of feasible control paths may be astronomical (even infinite) and because the precision of symbolic execution, constraint generation and solving is inherently limited. Nevertheless, whitebox fuzzing found new security vulnerabilities in several applications [16].

However, the effectiveness of whitebox fuzzing is limited when testing applications that require highly-structured inputs. Examples of such applications are compilers and interpreters. Such applications process the inputs in stages, such as lexing, parsing and evaluation. Due to the enormous number of control paths in the early processing stages, whitebox fuzzing rarely reaches parts of the application beyond the

```
1 //Reads and returns next token from file.
2 //Terminates on erroneous inputs.
3 Token nextToken(){
4   ...
5   readInputByte();
6   ...
7 }
8
9 //Parses the input file, returns parse tree.
10 //Terminates on erroneous inputs.
11 ParseTree parse(){
12   ...
13   Token t = nextToken();
14   ...
15 }
16
17 void main(){
18   ...
19   ParseTree t = parse();
20   ...
21   Bytecode code = codeGen(t);
22   ...
23 }
```

Figure 1. Sketch of an interpreter. The interpreter processes the inputs in stages: lexer (function `nextToken`), parser (function `parse`), and code generator (function `codeGen`). Next, the interpreter executes the generated bytecode (omitted here).

* The work of this author was done mostly while visiting Microsoft.

```
FunDecl   ::=  function id ( Formals ) FunBody
FunBody   ::=  { SrcElems }
SrcElems  ::=  ϵ
SrcElems  ::=  SrcElem SrcElems
Formals   ::=  id
Formals   ::=  id , Formals
...
```

Figure 2. Fragment of a context-free grammar for JavaScript. Nonterminals have names starting with uppercase. Symbol ϵ denotes the empty string. The starting nonterminal is FunDecl.

first stages. For instance, there are many possible sequences of blank-spaces/tabs/carriage-returns/etc. separating tokens in most structured languages, each corresponding to a different control path in the lexer. In addition to path explosion, symbolic execution itself may be defeated already in the first processing stages. For instance, lexers often detect language keywords by comparing their pre-computed, hard-coded hash values with the hash values of strings read from the input; this effectively prevents symbolic execution and constraint solving from ever generating input strings matching those keywords since hash functions cannot be inversed (i.e., given a constraint `x == hash(y)` and a value for `x`, one cannot compute a value for `y` satisfying this constraint).

In this paper, we present *grammar-based whitebox fuzzing*, which enhances whitebox fuzzing with a grammar-based specification of valid inputs. Grammar-based whitebox fuzzing is a dynamic test generation algorithm where symbolic execution directly generates grammar-based constraints whose satisfiability is checked using a custom grammar-based constraint solver. The algorithm has two key components:

1. Higher-level symbolic constraints, expressed in terms of symbolic grammar tokens returned by the lexer, instead of the traditional [15, 6, 16] symbolic bytes read as input.
2. Custom constraint solver that solves constraints on symbolic grammar tokens. The solver looks for solutions that satisfy the constraints *and* are accepted by a given (context-free) grammar. The grammar represents valid inputs for the program under test, and thus, by construction, all solutions found by the solver correspond to valid inputs.

Assuming the grammar accepts inputs only if they are parsable, the above algorithm never generates non-parsable inputs, i.e., it avoids dead-ends in the lexer or parser. Moreover, the grammar-based constraint solver can *complete* a partial set of token constraints into a fully-defined valid input, hence avoiding exploring many possible non-parsable completions.

As an example, consider the interpreter sketched in Figure 1 and the (JavaScript) grammar partially defined in Figure 2. By tracking the tokens returned by the lexer, i.e., the function `nextToken` (line 3) in Figure 1, and considering those as symbolic inputs, our technique generates constraints in terms of such tokens. For instance, running the interpreter on the valid input ``` ``function f(){ }'' ``` may correspond to the sequence of symbolic token constraints (of course, the precise form of the constraint depends on the actual source code of the parser, which is omitted from Figure 1) token_0 = `function`; token_1 = `id`; token_2 = `(`; token_3 = `)`; token_4 = {; token_5 = }. Negating the fourth constraint in this path constraint leads to the new sequence of constraints: token_0 = `function`; token_1 = `id`; token_2 = `(`; $\text{token}_3 \neq$ `)`. There are many ways to satisfy this constraints but most solutions lead to non-parsable inputs. In contrast, the grammar-based constraint solver directly concludes that the only way to satisfy this constraint *while generating a valid input* according to the grammar is to set token_3 = `id` *and* to complete the remainder of the input with, say, token_4 = `)`; token_5 = {; token_6 = }. The generated input that corresponds to this solution is ``` ``function f(id){ }'' ```, where `id` can be any identifier. Similarly, a grammar-based constraint solver can immediately prove that negating the third constraint token_2 = `(` in the above path constraint is unsolvable (i.e., there are no inputs that satisfy this constraint and are recognized by the grammar). Grammar-based whitebox fuzzing prunes the entire sub-tree of lexer executions corresponding to all possible non-parsable inputs matching this case.

By restricting the search space to valid inputs, grammar-based whitebox fuzzing can exercise deeper paths, and focus the search on the harder-to-test, deeper processing stages.

We have implemented grammar-based whitebox fuzzing and evaluated it on a large application, the JavaScript interpreter of the Internet Explorer 7 web-browser. Results of experiments show that grammar-based whitebox fuzzing outperforms whitebox fuzzing, blackbox fuzzing and grammar-based blackbox fuzzing in overall code coverage, while using fewer tests.

2. Grammar-based Whitebox Fuzzing

In this section, we describe whitebox fuzzing (Section 2.1) and introduce grammar-based whitebox fuzzing (Section 2.2). We then discuss how to check grammar-based constraints for context-free grammars (Section 2.3). Finally, we discuss additional aspects of our approach and some of its limitations (Section 2.4).

2.1 Whitebox Fuzzing

Algorithm 1 describes whitebox fuzzing [16], an algorithm for test input generation (the underlined text should be ignored for now.) This algorithm executes a sequential program *P* under test while performing a symbolic execution that tracks the effect of program inputs on conditional statements. The algorithm associates a symbolic variable with each byte of program input. The algorithm keeps a symbolic store that maps program variables to symbolic expressions composed of symbolic variables and constants. The algo-

rithm updates the symbolic store whenever the program manipulates input data (line 4). At every conditional statement that involves symbolic expressions, the algorithm extends the current path constraint *pc* with an additional conjunct *c* that represents the branch of the conditional statement taken in the current (concrete) execution (line 11). Each input *I* has an associated bound *I.bound* that denotes the depth of the parent path constraint that created *I* (initially 0). If the length $|pc|$ of the path constraint exceeds the bound *I.bound*, then the algorithm creates an *alternative* path constraint pc_{alt} by appending the *negation* of the conjunct *c* (line 13) to the path constraint. Any solution *s* to this alternative path constraint (line 14) corresponds to a program input that will drive the program execution exactly along the same control path until the branching point where the *opposite* branch will be taken (assuming symbolic execution has perfect precision, otherwise the actual execution may diverge from this path.) At the end of the execution, the new test inputs returned by Algorithm 1 are in turn executed, checked for runtime errors, and themselves expanded using Algorithm 1. The process is repeated until no new tests can be generated or until a time-limit expires.

2.2 Grammar-based Whitebox Fuzzing

Grammar-based whitebox fuzzing is an extension of the algorithm in Section 2.1. The underlined text in Algorithm 1 contains the three necessary changes. First, the new algorithm requires a grammar *G* that describes valid program inputs. Second, instead of marking the bytes in program inputs as symbolic (line 21), grammar-based whitebox fuzzing marks tokens returned from a tokenizing function such as `nextToken` in Figure 1 as symbolic (line 7); thus grammar-based whitebox fuzzing associates a symbolic variable with each token, and symbolic execution tracks the influence of the tokens on the control path taken by the program *P*. Third (line 14), the new algorithm uses the grammar to require that the new input not only satisfies the alternative path constraint but is also in the language accepted by the grammar. As the examples in the introduction illustrate, this additional requirement gives two advantages to grammar-based whitebox fuzzing: it allows pruning of the search tree corresponding to invalid inputs (i.e., inputs that are not accepted by the grammar), and it allows the direct completion of satisfiable token constraints into valid inputs.

2.3 Context-free Constraint Solver

The constraint solver invoked in line 14 of the grammar-based Algorithm 1 computes language intersection: it checks whether the language (set) $L(pc_{alt})$ of inputs satisfying the alternative path constraint pc_{alt} contains an input that is in the language accepted by the grammar. By construction, the language $L(pc_{alt})$ is always regular, as we show later in this section. If the grammar *G* is context-free, then language intersection with $L(pc_{alt})$ is decidable. If *G* is context-sensitive, then a sound and complete decision procedure for

input : Program *P*, input *I* , grammar *G*
output: New inputs, each for a different path in *P*

```
1  path constraint pc := true;
2  results := ∅;
3  foreach executed instruction inst do
4      update the symbolic store;
5      switch inst do
6          case return from tokenizing function
7              mark return token as symbolic;
8          end
9          case input-dependent conditional statement
10             c := expression for the executed branch;
11             pc := pc ∧ c;
12             if |pc| >I.bound then
13                 pc_alt := pc ∧ ¬c;
14                 s := Solve(L(pc_alt) ∩ L(G));
15                 s.bound := |pc_alt|;
16                 results := results ∪ {s};
17             end
18         end
19         otherwise
20             if inst is input reading ∧ false then
21                 mark input as symbolic;
22             end
23         end
24     end
25 end
26 return results
```

Algorithm 1: Grammar-based Whitebox fuzzing. Changes for grammar-based fuzzing are underlined.

computing language intersection may not exist (but approximations are possible). In what follows, we assume that *G* is context-free.

We assume that the set $\mathcal{T}$ of tokens that can be returned by the tokenization function is finite. Therefore, all token variables token_i have a finite range $\mathcal{T}$, and satisfiability of any constraint on a finite set of token variables is decidable. Given any such constraint pc_{alt}, one can sort its set of token variables token_i by their *index i*, representing the total order by which they have been created by the tokenization function, and build a regular expression (language) *R* representing $L(pc_{alt})$ for that constraint pc_{alt}.

A *context-free constraint solver* takes as inputs a context-free grammar *G* and a regular expression *R*, and returns either a string $s \in L(G) \cap L(R)$, or 'EMPTY', if the intersection is empty. The following procedure solves this problem in polynomial time:

1. convert *G* to a Push-Down Automaton (PDA),
2. convert *R* to a Finite-State Automaton (FSA),
3. compute the product PDA of the PDA and FSA,

input : context-free grammar G, grammar constraint R
output: s from $L(G) \cap L(R)$, or $EMPTY$ if $L(G) \cap L(R) = \varnothing$

1 $G' ::=$ duplicate productions for starting nonterminal S in G;
2 $G' ::=$ rename S to S' in G' (but not in the duplicated productions);
3 $n ::=$ highest index i of token$_i$ variable in R;
4 **for** $i = 0 \ldots n$ **do**
5 let constraint $c \in R \equiv token_i \in T$;
6 worklist $W ::=$ productions for S;
7 **while** W *not empty* **do**
8 $p ::=$ production from W;
9 **if** i^{th} *element in* p *is nonterminal* N **then**
10 add copies of p to W, with i^{th} element expanded using all productions for N;
11 **else**
12 remove p from W if i^{th} element in p is not in T (i.e., violates constraint);
13 **end**
14 **end**
15 **end**
16 **if** $L(G') = \varnothing$ **then**
17 **return** *EMPTY*
18 **else**
19 **return** *generate s from G′*
20 **end**

Algorithm 2: A simple context-free constraint solver.

4. check emptiness of the language accepted by the resulting PDA,
5. if non-empty, generate any string in that language, otherwise return 'EMPTY'.

Algorithm 2 is a simpler algorithm that we have implemented and used in our experiments. Given a context-free grammar and a regular expression, this algorithm computes a grammar representing exactly the strings in the intersection of both languages. This algorithm is simpler because it does not go through an explicit PDA translation and because it exploits the fact that, by construction, any regular language R always constrains only the first n tokens returned by the tokenization function, where n is the highest index i of a token variable token$_i$ appearing in the constraint represented by R. After the ith iteration of the **for** loop of the algorithm (line 4), the intermediate grammar satisfies prefixes $0 \ldots i$ of the grammar constraints, because step 12 in Algorithm 2 removes the productions that violate the i^{th} constraint. This simpler algorithm is not polynomial in general, but works well in practice for small values of n. Also, if the grammar is left-recursive, Algorithm 2 may not terminate. However, context-free grammars for file formats and programming languages are rarely left-recursive, and left-recursion can be efficiently removed [26].

S ::= (*let* ((*id* S)) S)
S ::= (Op S S)
S ::= *num*
S ::= *id*
Op ::= +
Op ::= −

Figure 3. Context-free grammar for simplified s-expressions. Figure 4 uses this grammar to illustrate Algorithm 2.

Figure 4 illustrates the algorithm on the example grammar from Figure 3. Starting with the initial grammar, the algorithm unrolls and prunes productions according to the grammar constraint. The left-most column shows the grammar after line 2 in the algorithm. Next, the main iteration begins. The first conjunct in the grammar constraint is "token$_0$ = (" , therefore the algorithm (step 12) removes the last two productions from the grammar. The second column show the result of this step. Next, the algorithm examines the second conjunct in the grammar constraint, 'token$_1$ = +"). The algorithm (step 10) expands the nonterminal Op in the production $S ::= (Op\ S'\ S')$. The production is replaced by two productions, $S ::= (+\ S'\ S')$ and $S ::= (-\ S'\ S')$, which are added to the worklist W. In the next iteration of the **for** loop, the second of the new productions is removed from the grammar (line 12) because it violates the grammar constraint. The third column in Figure 4 shows the grammar after this step. After 2 more iterations of the **for** loop, the algorithm arrives at the final grammar (right-most column).

The last step (line 19) of the algorithm generates a string s from the grammar G' for the intersection of G and R. For speed, our implementation uses a bottom-up strategy that generates a string with the lowest derivation tree for each nonterminal in the grammar, by combining the strings from the right-hand sides of productions for the nonterminal. This strategy is fast due to memoizing strings during string generation. Alternative generation strategies are possible, e.g., random generation (which we experimented with but gave inferior results). For the example grammar in the right-most column in Figure 4, the bottom-up strategy generates the string: "(+ (*let* ((*id num*)) *num*) *num*)". From this string of tokens, our tool generates a matching string of input bytes by applying an application-specific de-tokenization function.

2.4 Discussion and Limitations

Approximate grammars. Grammar-based whitebox fuzzing can be used with approximate grammars. If the grammar accepts all parsable inputs, i.e., over-approximates the set of parsable inputs, then Algorithm 1 is sound: it does not prune any of the feasible paths for which the parser successfully terminates.

$i = 0$	$i = 1$	$i = 2$	$i = 3$	**final**
$S' ::= (let\ ((id\ S'))\ S')$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
$S' ::= (Op\ S'\ S')$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
$S' ::= num$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
$S' ::= id$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
$Op ::= +$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
$Op ::= -$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
$S ::= (let\ ((id\ S'))\ S')$	$\rightarrow$	$S ::= (+\ S'\ S')$	$S ::= (+\ (let\ ((id\ S'))\ S')\ S')$	$S ::= (+\ (let\ ((id\ S'))\ S')\ S')$
$S ::= (Op\ S'\ S')$	$\rightarrow$		$S ::= (+\ (Op\ S'\ S')\ S')$	
$S ::= num$				
$S ::= id$				

Figure 4. Example illustrating Algorithm 2 on the grammar in Figure 3 and the regular grammar constraint: token_0 = (; token_1 = + ; token_2 = (; token_3 = (,),*num*,*id*,let ; Symbols $\rightarrow$ indicate that the production is unchanged from previous step. Consecutive columns show the grammar at the top of the `for` loop at line 4 in Algorithm 2. The last column shows the final grammar.

In practice, the set of valid inputs specified by a grammar is bound to be some approximation of the set of "parsable" inputs. Indeed, parsers typically implement additional validation (e.g., simple type-checking) that is not part of a typical grammar description of the language. Other grammars may have some "context-sensitive behaviors" (as in protocol description languages where a variable size parameter *k* is followed by *k* records), that are omitted or approximated in a context-free or regular manner. Other grammars, especially for network protocols, are simplified representations of valid inputs, and do not require the full power of context-sensitivity [28, 4, 9].

Domain knowledge. Grammar-based whitebox fuzzing requires a limited amount of domain knowledge, namely the formal grammar, identifying the tokenizing function, and providing the de-tokenization function. We believe this is not a severe practical limitation. Indeed, grammars are typically available for programming languages, and identifying the token-returning function is, in our experience, rather easy, even in unknown code, provided that the source code is available or the appropriate functions had standard names, such as `token`, `nextToken`, `scan`, etc. For instance, we found the tokenization function in the JavaScript interpreter of Internet Explorer 7 in a matter of minutes, by looking for commonly used names in the symbol table.

Lexer and parser bugs. Using a grammar to filter out invalid inputs may reduce coverage of the lexer and parser themselves, since the grammar explicitly prevents the execution of code-paths handling invalid inputs in those stages. For testing those stages, traditional whitebox fuzzing can be used. Moreover, our experiments indicate that grammar-based whitebox fuzzing does not decrease coverage in the lexer or parser.

Our approach uses the actual lexer and parser code of the program under test—removing these layers and using automatically generated software stubs simulating those parts may feed unrealistic inputs to the rest of the program.

3. Evaluation

We evaluate our approach experimentally and design experiments with the following goals:

- Compare grammar-based whitebox fuzzing to other approaches, grammar-*less* as well as *blackbox*. We compare how the various test generation strategies perform with a limited time budget and also examine their behavior over long periods of time (Section 3.5.1).
- Measure whether test inputs generated by our technique are effective in deeply penetrating the application, i.e., reaching beyond the lexer and parser. Section 3.5.1 gives the relevant experimental results.
- Measure how the set of inputs generated by each technique compares. In particular, do inputs generated by grammar-based whitebox fuzzing exercise the program in ways that other techniques do not? Section 3.5.2 presents the results.
- Measure the effectiveness of token-level constraints in preventing path explosion in the lexer. See Section 3.5.3 for the results.
- Measure the performance of the grammar constraint solver (Section 2.3) with respect to the size of test inputs. Section 3.5.4 discusses this point.
- Measure the effectiveness of the grammar-based approach in pruning the search tree. See Section 3.5.5.

The rest of this section describes our experiments and discusses the results. Naturally, because they come from a limited sample, these experimental results need to be taken with caution. However, our evaluation is extensive and performed with a large, widely-used JavaScript interpreter, a representative "real-world" program.

3.1 Subject Program

We performed the experiments with the JavaScript interpreter embedded in Internet Explorer 7. Our experimental setup runs the interpreter with no source modifications. The total size of the JavaScript interpreter is 113562 machine in-

strategy	seed inputs	random	tokens
blackbox	✓	✓	
grammar-based blackbox		✓	✓
whitebox	✓		
whitebox+tokens	✓		✓
grammar-based whitebox	✓		✓

Figure 5. Test input generation strategies evaluated and their characteristics. The **seed inputs** column indicates which strategies require initial seed inputs from which to generate new inputs. The **random** column indicates which strategies use randomness. The **tokens** column indicates which strategies use the lexical specification (i.e., tokens) of the input language. Each technique's name indicates whether the technique uses a grammar and whether is it white- or blackbox.

structions. In our experiments, we focus on the lexer, parser and code generator modules of the interpreter. Their respective sizes are 10410, 18535 and 3693 instructions. The code generator is the "deepest" of the examined modules, i.e., every input that reaches the code generator also reaches the other two modules (and not vice versa). The lexer and parser are equally deep—the parser always calls the lexer.

We use the official JavaScript grammar[1]. The grammar is quite large: 189 productions, 82 terminals (tokens), and 102 nonterminals.

3.2 Generation Strategies

We evaluate the following test input generation strategies, to compare them to grammar-based whitebox fuzzing.

blackbox generates test inputs by randomly modifying an initial input.

blackbox grammar generates test inputs by creating random strings from a given grammar. We use a strategy that generates strings of a given length uniformly at random [24], i.e., each string of a given length is equally likely.

whitebox generates test inputs using the whitebox fuzzing algorithm of Section 2.1.

whitebox+tokens enhances whitebox generation with the lexical part of the grammar, i.e., marks token identifiers as symbolic, instead of individual input bytes.

grammar-based whitebox is the grammar-based whitebox fuzzing algorithm of Section 2—it enhances whitebox fuzzing both with tokens and with the grammar.

Figure 5 tabulates the strategies we used in the evaluation and shows their characteristics. Other strategies are conceivable. For example, whitebox fuzzing could be used directly with the grammar, without tokens. Doing so requires transforming the grammar into a scannerless grammar [32].

[1] `http://www.ecma-international.org`

Another possible strategy is bounded exhaustive enumeration [35, 21]. We have not included the latter in our evaluation because, while all other strategies we evaluated are time-bounded (i.e., can be stopped at any time), exhaustive enumeration loses its meaning if it is terminated before completion, which makes it harder to fairly compare to time-bounded techniques.

3.3 Methodology

To avoid bias stemming from using arbitrary inputs, for techniques that require seed inputs (see Figure 5), we use 50 inputs of lengths 15–20 tokens generated randomly from the grammar. Section 3.4 provides more information about selecting the size of seeds inputs. To avoid any bias, we run all experiments inside the same test harness.

The **whitebox+tokens** and **grammar-based whitebox** strategies require specifying the function that is the source of tokens. Our framework allows doing so in a simple way, by overriding a single function.

For each of the examined modules (lexer, parser and code generator), we measure the reachability rate, i.e., the percentage of inputs that execute at least one instruction of the module. Deeper modules always have lower reachability rates.

We measure instruction coverage, i.e., the ratio of the number of unique, executed instructions to all instructions in the module of interest. This coverage metric is the most suitable for our needs—we want to estimate the bug-finding potential of the generated inputs, and blocks with more instructions are more likely to contain bugs than short blocks. In addition to the total coverage for the interpreter, we measure coverage in the lexer, parser and code generator modules.

We run each generation strategy for 2 hours. The 2-hour time includes *all* experimental tasks: program execution, symbolic execution (where applicable), constraint solving (where applicable), generation of new inputs and coverage measurements (To see whether giving more time changes the results, we also let each strategy run for much longer, until instruction coverage does not increase for 10 hours. See Section 3.5.1.)

For reference, we also include coverage data and reachability results obtained with a "manual" test suite, created over many years by the developers and testers of this JavaScript interpreter. The suite consists of more than 2800 hand-crafted inputs that exercise the interpreter thoroughly.

3.4 Seed Size Selection

Four of our generation strategies require seed inputs (Figure 5). To avoid bias stemming from using arbitrary inputs, we use inputs generated randomly from the JavaScript grammar. The length of the seed inputs may influence subsequent generation. To select the right length, we generate inputs of different sizes and measure the coverage achieved by each of those inputs as well as what percentage of inputs reaches the

size (tokens)	reach code gen. %	average coverage %	maximum coverage %
6	100	8.5	8.5
10	76.0	8.2	9.2
20	67.0	8.3	9.7
30	38.0	7.5	9.8
50	9.0	6.5	10.1
100	1.0	6.3	10.4
120	0.0	6.2	6.8
150	0.0	6.2	6.7
200	0.0	6.2	6.7

Figure 6. Coverage statistics for nine sets of 100 inputs each, generated randomly from the JavaScript grammar (using the same uniform generator as **grammar-based blackbox**.) The **reach code gen.** column displays the percentage of the generated inputs that reach the code generator module. The two right-most columns display the average and the maximum coverage (of the whole interpreter) for the generated inputs.

code generator. For each length, we generate 100 inputs and perform the measurements only for those inputs. Figure 6 presents the results.

The findings are not immediately intuitive—longer inputs achieve, on average, lower total coverage. The reason is that the official JavaScript grammar is only a partial specification of what constitutes syntactic validity. The grammar describes an over-approximation of the set of inputs acceptable by the parser (longer, randomly generated, inputs are more likely to be accepted by the grammar and *rejected* by the parser.) For example, the grammar specifies that `break` statements may occur anywhere in the function body, while the parser enforces that `break` statements may appear only in loops and `switch` statements. Enforcing this is possible by modifying the grammar but it would make the grammar much larger. Another example of over-approximation concerns line breaks and semicolons. The standard specifies that certain semicolons may be omitted, as long as the are appropriate line breaks in the file[2]. However, the grammar does not enforce this requirement and allows omitting all semicolons.

By analyzing the results, we select 15–20 as the size range, in tokens, of the seeds we use in other experiments. This length makes the seed inputs variable without sacrificing the penetration rate (i.e., reachability of the code generation module).

3.5 Results

3.5.1 Coverage and Penetration

Figure 7 tabulates the coverage and reachability results for the 2-hour runs. **Grammar-based whitebox** fuzzing achieves results that are closest to the manual suite (which, predictably, performed best). In particular, it achieves noticeably better coverage than **whitebox**. Of all examined strategies, **grammar-based whitebox** achieves the best total coverage as well as the best coverage in the deepest examined module, the code generator.

Grammar-based whitebox fuzzing performs also significantly better than `grammar-based blackbox`. Even though the latter strategy achieved good coverage in the code generator, whitebox strategies outperform blackbox ones in total coverage.

Moreover, **grammar-based whitebox** fuzzing achieves the highest coverage using the fewest inputs, which means that this strategy generates inputs of higher quality. Generating few, high-quality test inputs is important for regression testing.

The **blackbox** and **whitebox** techniques achieved similar results in all categories. This shows that, when testing applications with highly-structured inputs, whitebox fuzzing, with the power of dynamic analysis and symbolic execution, does not improve much over simple blackbox fuzzing. In fact, in the code generator, those grammar-*less* strategies do not improve coverage much above the initial set of seed inputs.

Reachability results show that almost all tested inputs reach the lexer. A few inputs generated by the **blackbox** and **whitebox** strategies contains invalid, e.g., non-ASCII, characters and the interpreter rejects them before using the lexer. To exercise the interpreter well, inputs must reach the deepest module, the code generator. The results show that **grammar-based whitebox** has the highest percentage of such deep-penetrating inputs.

To analyze the long generation-time behavior of the examined strategies, we let each strategy run for as long as it keeps covering new instructions at least every 10 hours. The results are that, after the initial 2 hours, each configuration reaches around 90% of coverage that it is eventually capable of reaching (this validates our selection of the 2-hour time limit for our experiments.) The long generation-time runs confirm the findings of the 2-hour runs. **grammar-based whitebox** fuzzing is the most effective of the examined techniques—it reaches the highest coverage and covers new code for longer than other techniques (97 hours vs. 84 hours for **whitebox** and 82 hours for **grammar-based blackbox**).

The results of the presented experiments validate our claim that grammar-based whitebox fuzzing is effective in penetrating the tested application more deeply and exercising the code more thoroughly than other techniques.

3.5.2 Relative Coverage

Figure 8 compares the instructions covered with **grammar-based whitebox** fuzzing and the other analyzed strategies. From the **only** *s* and the **gbw** columns for each strategy, we observe that inputs generated by **grammar-based whitebox** cover a larger number of unique instructions (for **grammar-based blackbox**, significantly so). Since (Section 3.5.1)

[2] Section 7.9 of the specification: `http://interglacial.com/javascript_spec/a-7.html#a-7.9`

strategy	inputs	total	lexer		parser		code generator	
		coverage %	reach %	coverage %	reach %	coverage %	reach %	coverage %
blackbox	8658	14.2	99.6	24.6	99.6	24.8	17.6	52.1
grammar-based blackbox	7837	11.9	100	22.1	100	24.1	72.2	61.2
whitebox	6883	14.7	99.2	25.8	99.2	28.8	16.5	53.5
whitebox+tokens	3086	16.4	100	35.4	100	39.2	15.5	53.0
grammar-based whitebox	2378	20.0	100	24.8	100	42.4	80.7	81.5
seed inputs	50	10.6	100	18.4	100	20.6	66.0	50.9
manual suite	2820	58.8	100	62.1	100	76.4	100	91.6

Figure 7. Coverage results for 2-hour runs. Manual test suite takes more than 2 hours to run and is included here for reference. The **seed inputs** row lists statistics for the seed inputs used by the generation strategies (see Figure 5). The **inputs** column gives the number of inputs tested by each strategy (i.e., those generated inputs for which our harness computes coverage information during the 2-hour time limit). The **total coverage** column gives the total instruction coverage percentage. Coverage statistics for lexer, parser and code generator modules are given in the corresponding columns. The **reach** columns give the percentage of inputs that reach the module's entry-point. The **coverage** columns give the instruction coverage for the module.

strategy *s*	only *s*	*s* and gbw	only gbw
blackbox	1176	14931	7747
grammar-based blackbox	507	13036	9642
whitebox	1761	14978	7700
whitebox+tokens	2782	15789	6889

Figure 8. Relative coverage compared to **grammar-based whitebox** (**gbw**). The column "**only** *s*" gives the total number of instructions covered by each strategy but *not* by **gbw**. The column "*s* **and gbw**" gives the total number of instructions covered by both strategies. The last column gives the total of instructions covered by "only **gbw**",

grammar-based whitebox fuzzing achieves the highest total coverage, highest reachability rate and coverage in the deepest module while using the smallest number of inputs, it creates tests inputs of the highest quality among the analyzed strategies.

3.5.3 Symbolic Executions

Figure 9 presents statistics for symbolic executions. The **whitebox** strategy creates most symbolic variables because it operates on characters, while the other two strategies work on tokens (cf. Figure 5). The **whitebox+tokens** strategy creates the fewest symbolic variables per execution. This is because **whitebox+tokens** generates many unparsable inputs (cf. Figure 7), which the parser rejects early and therefore no symbolic variables are created for the tokens after the parse error.

Figure 9 shows how constraint creation is distributed among the lexer, parser and code generator modules of the JavaScript interpreter. The two token-based strategies (**whitebox+tokens** and **grammar-based whitebox**) generate no constraints in the lexer. This helps to avoid path explosion in that module. Those strategies do explore the lexer (indeed, Figure 7 shows high coverage) but they do not get lost in the error paths.

All strategies create constraints in the deepest, code generator, module. However, there are few such constraints because the parser transforms the stream of tokens into an Abstract Syntax Tree (AST) and subsequent code operates on the AST. When processing the AST in later stages, symbolic variables associated with bytes or tokens are absent (also partially due to the incompleteness of the constraint theory), so symbolic execution does not create constraints for branches in these stages.

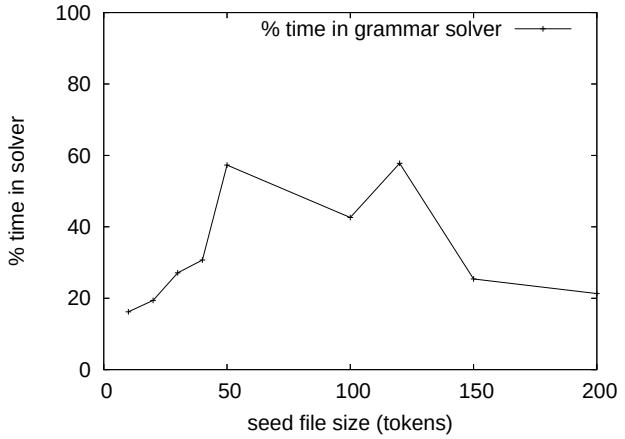

Figure 10. Grammar solver performance. The **% time in solver** line indicates what percentage of total 2-hour run time was spent in the grammar constraint solver.

3.5.4 Grammar Solver Performance

To measure the performance of the grammar constraint solver, we repeated the 2-hour **grammar-based whitebox** run 9 times with different sizes of seed inputs (between 10 and 200 tokens). The average number of solver calls per symbolic execution was between 23 and 53 (with no visible relationship between seed input size and the number of calls). Figure 10 shows how much of the total execution time was spent in the constraint solver. The results present no obvious correlation between seed size and solving time.

strategy	created constraints %			symbolic	average	average
	lexer	parser	code gen.	executions	symbolic variables	constraints
whitebox	66.6	33.1	0.3	131	57.1	297.7
whitebox+tokens	0.0	98.0	2.0	170	11.8	66.9
grammar-based whitebox	0.0	98.0	2.0	143	21.1	113.0

Figure 9. Constraint creation statistics for 2-hour runs of whitebox strategies. The **created constraints** columns shows the percentages of all symbolic constraints created in the three analyzed modules of the JavaScript interpreter. The **symbolic executions** column gives the total number of symbolic executions during each run. The two right-most columns give the average number of symbolic variables per symbolic execution and the average number of symbolic constraints per symbolic execution.

3.5.5 Grammar-based Search Tree Pruning

Grammar-based whitebox fuzzing is effective in pruning the search tree. In our experiments, 29.0% of grammar constraints are unsatisfiable. When a grammar constraint is unsatisfiable, the corresponding search tree is pruned because there is no input that satisfies the constraint and is valid according to the grammar.

4. Related Work

Automated approaches to systematic testing based on dynamic test generation, such as DART [15], CUTE [33], EXE [6] and SAGE [16] are popular because they find bugs without generating false alarms and require no domain knowledge. Our work enhances dynamic test generation by taking advantage of a formal grammar representing valid inputs, thus helping the generation of test inputs that penetrate the application deeper.

Miller's pioneering fuzzing tool [25] generated streams of random bytes, but most popular fuzzers today support some form of grammar representation, e.g., SPIKE [1], Peach [29], FileFuzz [12], Autodafé [2]. Work on grammar-based test input generation started in the 1970's [17, 31] and can be broadly divided into random [34, 23, 22, 8] and exhaustive generation, e.g., [19, 21]. Imperative generation [7, 27, 10] is a related approach in which a custom-made program generates the inputs (in effect, the program encodes the grammar). In systematic approaches, test inputs are created from a specification, given either a special piece of code (e.g., Korat [5]) or a logic formula (e.g., TestEra [18]). Grammar-based test input generation is an example of model-based testing (see Utting *et al.* for a survey [36]), which focuses on covering the specification (model) when generating test inputs to check conformance of the program with respect to the model. Our work also uses formal grammars as specifications. However, in contrast to those blackbox approaches, our approach analyses the code of the program under test.

Path explosion in systematic dynamic test generation can be alleviated by performing test generation compositionally [14], by testing functions systematically in isolation, encoding and memoizing test results as function summaries using function input preconditions and output postconditions, and re-using such summaries when testing higher-level functions. A grammar can be viewed as a special form of user-provided compact "summary" for the entire lexer/-parser, that may include over-approximations. Computing such a finite-size summary automatically may be impossible due to the presence of infinitely many paths or limited symbolic reasoning capability when analyzing the lexer/parser. Grammar-based whitebox fuzzing and test summaries are complementary techniques and could be used simultaneously.

Another approach to path explosion consists of abstracting lower-level functions using software stubs, marking their return values as symbolic, and then refining these abstractions to eliminate unfeasible program paths [20]. In contrast, grammar-based whitebox fuzzing is always grounded in concrete executions, and thus does not require the expensive step of removing unfeasible paths.

Emmi *et al.* [11] extend systematic testing with constraints that describe the state of the data for database applications. Our approach also solves path and data constraints simultaneously, but ours is designed for compilers and interpreters instead of database applications.

Majumdar and Xu's recent and independent work [21] is closest to ours. The authors combine grammar-based black-box fuzzing with dynamic test generation by exhaustively pre-generating strings from the grammar (up to a given length), and then performing dynamic test generation starting from those pre-generated strings, treating only variable names, number literals etc. as symbolic. Exhaustive generation inhibits scalability of this approach beyond very short inputs. Also, the exhaustive grammar-based generation and the whitebox dynamic test generation parts do not interact with each other in Majumdar and Xu's framework. In contrast, our grammar-based whitebox fuzzing approach is more elaborate and powerful as it exploits the grammar for solving constraints generated during symbolic execution to generate input variants that are guaranteed to be valid.

5. Conclusion

We introduced grammar-based whitebox fuzzing to significantly enhance the effectiveness of dynamic test input generation for applications with complex, highly-structured inputs, such as interpreters and compilers. Grammar-based

whitebox fuzzing tightly integrates constraint-based whitebox with grammar-based blackbox testing, and leverages their strengths.

As shown by our in-depth study with the IE7 JavaScript interpreter, grammar-based whitebox fuzzing generates higher-quality tests that exercise more code in the deeper, harder-to-test layers of the application under test (see Figure 7). In our experiments, it strongly outperformed traditional grammar-less whitebox fuzzing, the latter not being any better than random bit flipping done with blackbox fuzzing.

Since grammars are bound to be partial specifications of valid inputs, grammar-based blackbox approaches are fundamentally limited. Thanks to whitebox dynamic test generation, some of this incompleteness can be recovered, which explains why grammar-based whitebox fuzzing also outperformed grammar-based blackbox fuzzing in our experiments.

References

[1] D. Aitel. The Advantages of Block-Based Protocol Analysis for Security Testing. *Immunity Inc., February*, 2002.

[2] Autodafé. http://autodafe.sourceforge.net.

[3] D. Bird and C. Munoz. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[4] N. Borisov, D. Brumley, H. Wang, J. Dunagan, P. Joshi, and C. Guo. Generic application-level protocol analyzer and its language. In *NDSS*, 2007.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. *ISSTA*, 2002.

[6] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: automatically generating inputs of death. In *CCS*, 2006.

[7] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP*, 2000.

[8] D. Coppit and J. Lian. yagg: an easy-to-use generator for structured test inputs. *ASE*, 2005.

[9] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, 2007.

[10] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *FSE*, 2007.

[11] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, 2007.

[12] Filefuzz. http://labs.idefense.com/software/fuzzing.php.

[13] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*, Seattle, August 2000.

[14] P. Godefroid. Compositional Dynamic Test Generation. In *POPL*, 2007.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[16] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. Technical Report MS-TR-2007-58, Microsoft, 2007.

[17] K. Hanford. Automatic Generation of Test Cases. *IBM Systems Journal*, 9(4), 1970.

[18] S. Khurshid and D. Marinov. TestEra: Specification-Based Testing of Java Programs Using SAT. *ASE*, 11(4), 2004.

[19] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. *TestCom*, 2006.

[20] R. Majumdar and K. Sen. LATEST: Lazy dynamic test input generation. Technical Report UCB/EECS-2007-36, EECS Department, University of California, Berkeley, 2007.

[21] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *ASE*, 2007.

[22] B. Malloy and J. Power. An interpretation of Purdom's algorithm for automatic generation of test cases. *1st Annual International Conference on Computer and Information Science*, 2001.

[23] P. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4), 1990.

[24] B. McKenzie. Generating strings at random from a context free grammar. Technical Report TR-COSC 10/97, Department of Computer Science, University of Canterbury, 1997.

[25] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), 1990.

[26] R. C. Moore. Removing left recursion from context-free grammars. In *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*, 2000.

[27] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, 2007.

[28] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: a yacc for writing application protocol parsers. *Proceedings of the 6th ACM SIGCOMM on Internet measurement*, 2006.

[29] Peach. http://peachfuzz.sourceforge.net/.

[30] PROTOS: security testing of protocol implementations. http://www.ee.oulu.fi/research/ouspg/protos/.

[31] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3), 1972.

[32] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *PLDI*, 1989.

[33] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes*, 30(5), 2005.

[34] E. Sirer and B. Bershad. Using production grammars in software testing. *Proceedings of the 2nd conference on Domain-specific languages*, 1999.

[35] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *ISSTA*, 2004.

[36] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-Based Testing. *Department of Computer Science, The University of Waikato, New Zealand, Tech. Rep*, 4, 2006.