Grammar-based Whitebox Fuzzing

Patrice Godefroid
Microsoft Research
Redmond, WA, USA
pg@microsoft.com

Adam Kieżun
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
Cambridge, MA, USA
akiezun@mit.edu

Michael Y. Levin
Microsoft Center for Software Excellence
Redmond, WA, USA
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. Unfortunately, the current 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 early processing stages, whitebox fuzzing rarely reaches parts of the application beyond those 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 (IE7). Results of our 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.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Algorithms, Reliability

Keywords Software Testing, Automatic Test Generation, Grammars, Program Verification

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 [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 [1, 37].

A recently proposed alternative, *whitebox fuzzing* [16], combines fuzz testing with dynamic test generation [6, 14]. Whitebox fuzzing executes the program under test with an initial, well-formed input, both concretely and symbolically. During the execution of conditional statements, symbolic execution 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, with the goal of exercising many different control paths of the program under test and finding bugs as fast as possible using various search heuristics. 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 has been shown to be very effective in finding new security vulnerabilities in several applications.

Unfortunately, the current effectiveness of whitebox fuzzing is limited when testing applications with highly-structured inputs. Examples of such applications are 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 early processing stages, whitebox fuzzing rarely reaches parts of the application beyond these 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 that match 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` that satisfies this constraint).

```
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).

In this paper, we present *grammar-based whitebox fuzzing*, which enhances whitebox fuzzing with a grammar-based specification of valid inputs. We present 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. Generation of higher-level symbolic constraints, expressed in terms of symbolic grammar tokens returned by the lexer, instead of the traditional [6, 14, 16] symbolic bytes read as input.
2. A 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.

Assuming the grammar accepts inputs only if they are parsable, our algorithm never generates non-parsable inputs, i.e., it avoids dead-ends in the lexer and 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. 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.

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 dynamic test generation algorithm 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

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, our grammar-based constraint solver can directly conclude 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 = }

Thus, the generated input that corresponds to this solution is

```
function f(id){ }
```

where `id` can be any identifier.

Similarly, the grammar-based constraint solver can immediately prove that negating the third constraint in the previous path constraint, thus leading to the new path constraint

token_0 = `function`
token_1 = `id`
$\text{token}_2 \neq$ `(`

is *unsolvable*, i.e., there are no inputs that satisfy this constraint and are recognized by the grammar. Grammar-based whitebox fuzzing prunes in one iteration the entire subtree of lexer executions corresponding to all possible non-parsable inputs matching this case.

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

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.

2. Grammar-based Whitebox Fuzzing

In this section, we recall the basic notions behind 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 shows the whitebox fuzzing algorithm [16] (the underlined text should be ignored for now). Given a sequential deterministic program $\mathcal{P}$ under test and an initial program input $\mathcal{I}$, this dynamic test generation algorithm generates new test inputs by negating constraints generated during the symbolic execution of program $\mathcal{P}$ with input $\mathcal{I}$. These new inputs exercise different execution paths in $\mathcal{P}$. This process is repeated and the algorithm executes the program with new inputs multiple times—each newly generated input may lead to the generation of additional inputs. The algorithm terminates when a testing time budget expires or no more inputs can be generated.

The algorithm starts by checking whether running program $\mathcal{P}$ with input $\mathcal{I}$ triggers a runtime error (line 3). The algorithm associates an attribute *bound* with each input, initially set to 0 (line 4), as an optimization that avoids generating the same path constraint multiple times. Variable *worklist* represents a priority queue of inputs that have yet to be explored. The queue is initialized to a singleton containing the initial input (line 6). The algorithm then enters an iterative phase that continues as long as there are more inputs to explore, and as long as the time budget allows (line 7). As long as this is the case, an input is taken out of the *worklist* (line 8). Then, the program is executed symbolically on the input (line 9). The result of symbolic execution (explained below) is a *path constraint*, which is a conjunction of constraints on the program's input parameters, $c_1 \wedge \ldots \wedge c_n$, that are all satisfied on the current execution path. The algorithm then creates new test inputs by modifying the path constraint (lines 10–16), as follows. For each prefix of the path constraint (longer than the *bound* associated with the last input), the algorithm negates the last conjunct (line 11). A solution, if it exists, to such an alternative path constraint corresponds to an input that will execute the program along the prefix of the original execution path, but take the *opposite* branch of the conditional statement corresponding to the last constraint in that prefix (assuming symbolic execution and constraint solving have perfect precision, otherwise the actual execution may diverge from this path). The algorithm calls the constraint solver to find a concrete input that satisfies the alternative path constraint (line 12). If such a value exists (line 13), and can be found by the constraint solver, this new test input is run and checked (line 14), its *bound* is set to value i (line 15), and it is added to the queue (line 16).

```
parameters: Program $\mathcal{P}$, input $\mathcal{I}$$\underline{\text{, grammar } \mathcal{G}}$
result    : Bugs in $\mathcal{P}$
1  Procedure grammarBasedWhiteboxFuzzing($\mathcal{P}, \mathcal{I}\underline{, \mathcal{G}}$):
2  bugs := $\emptyset$
3  bugs := bugs $\cup$ Run&Check($\mathcal{P}, \mathcal{I}$)
4  $\mathcal{I}$.bound := 0
5  worklist := emptyQueue()
6  enqueue(worklist, $\mathcal{I}$)
7  while not empty(worklist) and not timeExpired() do
8      input := dequeue(worklist)
9      $c_1 \wedge \ldots \wedge c_n$ := executeSymbolic($\mathcal{P}$, input)
10     for $i$ := input.bound,...,$n$ do
11         pc := $c_1 \wedge \ldots \wedge c_{i-1} \wedge \neg c_i$
12         newInput := solve(pc$\underline{, \mathcal{G}}$)
13         if newInput $\neq \perp$ then
14             bugs := bugs $\cup$ Run&Check($\mathcal{P}$, newInput)
15             newInput.bound := $i$
16             enqueue(worklist, newInput)
17 return bugs

18 Procedure executeSymbolic($\mathcal{P}, \mathcal{I}$):
19 path constraint pc := true
20 foreach instruction inst executed by $\mathcal{P}$ with $\mathcal{I}$ do
21     update the symbolic store
22     switch inst do
23         case $\underline{\text{return from tokenization function}}$
24             $\underline{\text{mark token as symbolic variable}}$
25         case input-dependent conditional statement
26             c := expression for the executed branch
27             pc := pc $\wedge$ c
28         otherwise
29             if $\underline{\text{false} \wedge}$ inst reads byte from $\mathcal{I}$ then
30                 mark input byte as symbolic variable;
31 return pc
```

Algorithm 1: Grammar-based whitebox fuzzing. Changes from whitebox fuzzing are $\underline{\text{underlined}}$. The auxiliary procedure *executeSymbolic* (lines 18–31) changes for grammar-based fuzzing. Grammar-based whitebox fuzzing requires the constraint solver (auxiliary procedure *solve*) to handle grammar constraints (Algorithm 2).

Symbolic execution, implemented in procedure *executeSymbolic* in Algorithm 1, is a well established technique [20]. Here we consider a particular form of symbolic execution which is carried out *dynamically*, while the program is running on a particular input. Dynamic execution allows any imprecision in symbolic execution to be alleviated using concrete values and randomization: whenever symbolic execution does not know how to generate a constraint for a program statement depending on some inputs, one can always simplify this constraint using the concrete values of those inputs [14]. Symbolic execution takes the program and the input and records how the program's input affects the control flow in the program. The result of symbolic execution is a path constraint that is a logic formula that is satisfied by the currently executed concrete input and any other concrete input that will drive the program's execution along the same control path. Symbolic variables in the path constraint refer to bytes in the program's input. The algorithm keeps a symbolic store that maps program variables to symbolic expressions composed of symbolic variables and constants. The algorithm updates the symbolic store whenever the program manipulates input data (line 21). 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 execution (line 27). At every instruction that reads a byte from the input, a symbolic variable is associated with the input byte (line 30).

```
parameters: Path constraint $pc$, grammar $\mathcal{G}$ with start
        symbol $S$
result    : $s \in L(pc) \cap L(\mathcal{G})$ or $\perp$
1  Procedure $solve(pc, \mathcal{G})$:
2  $R := buildConstraint(pc)$
3  $\mathcal{G}' := \mathcal{G}$
4  $\mathcal{G}'$ := duplicate productions for starting nonterminal $S$
5  $\mathcal{G}'$ := rename $S$ to $S'$ (but not in the duplicated
   productions)
6  $n$ := highest index $i$ of token$_i$ variable in $R$
7  for $i := 1 \ldots n$ do
8      let $c_i$ denote the constraint in $R$ on variable $token_i$
9      worklist $W$ := productions for $S$ in $\mathcal{G}'$
10     while not empty($W$) do
11         $prod := dequeue(W)$
12         $S_i := i^{th}$ symbol in $prod.rhs$
13         if $S_i$ is nonterminal $N$ then
14             add copies of $prod$ to $W$ and $\mathcal{G}'$, with $S_i$
               expanded using all productions for $N$ in $\mathcal{G}'$
               (unroll)
15         else
16             remove $prod$ from $\mathcal{G}'$ if $S_i$ does not satisfy $c_i$
               (prune)
17 if $L(\mathcal{G}') = \emptyset$ then
18     return $\perp$
19 else
20     return generate $s$ from $\mathcal{G}'$
```

Algorithm 2: Procedure $solve(pc, \mathcal{G})$ implements a context-free constraint solver. The auxiliary function *buildConstraint*(pc) converts the path constraint pc to a regular expression. Notation *prod.rhs* denotes the right hand side of the production *prod*.

2.2 Grammar-based Extension to Whitebox Fuzzing

Grammar-based whitebox fuzzing is an extension of the algorithm in Section 2.1. The underlined text in Algorithm 1 contains the necessary changes.

- The new algorithm requires a grammar $\mathcal{G}$ that describes valid program inputs (line 1).
- Instead of marking the bytes in program inputs as symbolic (line 30), grammar-based whitebox fuzzing marks tokens returned from a tokenization function such as `nextToken` in Figure 1 as symbolic (line 24); thus grammar-based whitebox fuzzing associates a symbolic variable with each token[1], and symbolic execution tracks the influence of the tokens on the control path taken by the program $\mathcal{P}$.
- The algorithm uses the grammar $\mathcal{G}$ to require that the new input not only satisfies the alternative path constraint but is also in the language accepted by the grammar (line 12). 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.

[1] Symbolic variables could also be associated with other values returned by the tokenization function for specific types of tokens, such as the string value associated with each identifier, the numerical value associated with each number, etc.

2.3 Context-free Constraint Solver

The constraint solver invoked in line 12 of Algorithm 1 implements the procedure *solve* and computes language intersection: it checks whether the language $L(pc)$ of inputs satisfying the path constraint pc contains an input that is in the language accepted by the grammar $\mathcal{G}$. By construction, the language $L(pc)$ is always regular, as we discuss later in this section. If $\mathcal{G}$ is context-free, then language intersection with $L(pc)$ is decidable. If $\mathcal{G}$ is context-sensitive, then a sound and complete decision procedure for computing language intersection may not exist (but approximations are possible). In what follows, we assume that $\mathcal{G}$ 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, 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 $L(pc)$ for that constraint pc.

A *context-free constraint solver* takes as inputs a context-free grammar $\mathcal{G}$ and a regular expression R, and returns either a string $s \in L(\mathcal{G}) \cap L(R)$, or $\perp$ if the intersection is empty.

Algorithm 2 presents a decision procedure for such a constraint solver. The algorithm 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. The algorithm starts by converting the path constraint into a regular expression R (line 2). This is straightforward and involves grouping the constraints in pc by the token variable index. The next 3 lines (lines 3–5) are technical steps to eliminate recursion for the start symbol S. The algorithm employs a simple unroll-and-prune approach: in the i^{th} iteration of the main loop (line 7), the algorithm unrolls the right-hand sides of productions to expose a $0 \ldots i$ prefix of terminals (line 14), and prunes those productions that violate the constraint c_i on the i^{th} token variable token$_i$ in the regular expression R (line 16). During each round of unrolling and pruning, the algorithm uses the worklist W to store productions that have not yet been unrolled and examined for conformance with the regular expression.

After the unrolling and pruning, the algorithm checks emptiness [18] of the resulting language $L(\mathcal{G}')$ and generates a string s from the intersection grammar $\mathcal{G}'$ (line 20). 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 nonterminals. This strategy is fast due to memoizing strings during generation. Section 2.4 discusses alternatives and limitations of Algorithm 2.

Solving example. We illustrate the algorithm on an example, a simplified S-expression grammar. Starting with the initial grammar, the algorithm unrolls and prunes productions given a regular path constraint. The grammar is (S is the start symbol, nonterminals are uppercase)

$$\begin{array}{lcl} S & ::= & (let\ ((id\ S))\ S) \mid (Op\ S\ S) \mid num \mid id \\ Op & ::= & + \mid - \end{array}$$

and the regular path constraint is

$$\begin{array}{lll} token_1 & \in & \{(\} \\ token_2 & \in & \{+\} \\ token_3 & \in & \{(\} \\ token_4 & \in & \{(,),num,id,let\} \end{array}$$

Before the main iteration (line 7), the grammar is:

$$\begin{array}{lll} S' & ::= & (let\ ((id\ S'))\ S') \mid (Op\ S'\ S') \mid num \mid id \\ Op & ::= & + \mid - \\ S & ::= & (let\ ((id\ S'))\ S') \mid (Op\ S'\ S') \mid num \mid id \end{array}$$

Next, the main iteration begins. The first conjunct in the grammar constraint is $token_1 \in \{(\}$, therefore the algorithm (line 16) removes the last two productions from the grammar. The result is the following grammar (execution is now back at the top of the loop in line 7).

$$\begin{array}{lll} S' & ::= & (let\ ((id\ S'))\ S') \mid (Op\ S'\ S') \mid num \mid id \\ Op & ::= & + \mid - \\ S & ::= & (let\ ((id\ S'))\ S') \mid (Op\ S'\ S') \end{array}$$

In the next iteration of the **for** loop, the algorithm examines the second conjunct in the regular path constraint, $token_2 \in \{+\}$. The algorithm prunes the first production rule from S since *let* does not match + (line 16), and then expands the nonterminal *Op* in the production $S ::= (Op\ S'\ S')$ (line 14). The production is replaced by two productions, $S ::= (+\ S'\ S')$ and $S ::= (-\ S'\ S')$, which are added to the worklist W. The grammar $\mathcal{G}'$ is then

$$\begin{array}{lll} S' & ::= & (let\ ((id\ S'))\ S') \mid (Op\ S'\ S') \mid num \mid id \\ Op & ::= & + \mid - \\ S & ::= & (+\ S'\ S') \mid (-\ S'\ S') \end{array}$$

In the next iteration of the **while** loop, the second of the new productions is removed from the grammar (line 16) because it violates the grammar constraint. After the removal, the execution is now again at the top of the loop in line 7.

$$\begin{array}{lll} S' & ::= & (let\ ((id\ S'))\ S') \mid (Op\ S'\ S') \mid num \mid id \\ Op & ::= & + \mid - \\ S & ::= & (+\ S'\ S') \end{array}$$

After 2 more iterations of the **for** loop, the algorithm arrives at the final grammar

$$\begin{array}{lll} S' & ::= & (let\ ((id\ S'))\ S') \mid (Op\ S'\ S') \mid num \mid id \\ Op & ::= & + \mid - \\ S & ::= & (+\ (let\ ((id\ S'))\ S')\ S') \end{array}$$

As the last two steps, the algorithm checks that $L(\mathcal{G}') \neq \emptyset$ (line 17) and generates a string s from the final grammar $\mathcal{G}'$ for the intersection of $\mathcal{G}$ and R (line 20). Our 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 detokenization function.

2.4 Discussion and Limitations

Computing language intersection. Computing the intersection of a context-free grammar with a regular expression is a well-known problem. A standard polynomial-time algorithm consists in translating the grammar into a pushdown automaton, translating the regular expression into a finite-state automaton, computing the product of these two automata to obtain another pushdown automaton, and finally translating the resulting pushdown automaton back into a context-free grammar. Alternatively, the intersection can be computed without the explicit automata conversion [39], by an adaptation of the context-free reachability algorithm [27].

The unroll-and-prune algorithm we present in Section 2.3 is simpler because it does not go through an explicit pushdown automaton translation and exploits the structure of the regular language that describes the path constraint on 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. This algorithm is not polynomial in general, but performs satisfactorily in practice for small values of n (around 50–60). 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 [29].

Approximate grammars. Grammar-based whitebox fuzzing can be used with approximate grammars. Let us call an input *parsable* if the parser successfully terminates when run on that input. If the grammar accepts all parsable inputs or 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.

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 [4, 9, 31].

Domain knowledge. Grammar-based whitebox fuzzing requires a limited amount of domain knowledge, namely the formal grammar, identifying the tokenization function to be instrumented, and providing a de-tokenization function to generate input byte strings from input token strings generated by a context-free constraint solver. We believe this is not a severe practical limitation. Indeed, grammars are typically available for many input formats, and identifying the tokenization function is, in our experience, rather easy, even in unknown code, provided that the source code is available or that the tokenization functions has a standard name, 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 code coverage in 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 (Section 3) indicate that grammar-based whitebox fuzzing does not decrease coverage in the lexer or parser.

Grammar-based whitebox fuzzing approach uses the actual lexer and parser code of the program under test. Indeed, 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 grammar-based whitebox fuzzing 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 in Section 3.5.1.
- Measure whether test inputs generated by our technique are effective in exercising deep execution paths in 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 of 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 the Internet Explorer 7 Web-browser. Our experimental setup runs the interpreter with no source modifications. The total size of the JavaScript interpreter is 113562 machine instructions. In our experiments, we also measure coverage in the lexer, parser and code generator modules of the interpreter. Their respective sizes are 10410, 18535 and 3693 machine 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 (but the converse does not hold). The lexer and parser are equally deep, because the parser always calls the lexer.

We use the official JavaScript grammar[2]. The grammar is quite large: 189 productions, 82 terminals (tokens), and 102 nonterminals.

3.2 Test 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. We use a Microsoft-internal widely-used blackbox fuzzing tool.

grammar-based blackbox 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 [26], i.e., each string of a given length is equally likely.

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

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

Figure 3. 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 randomization. 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 whitebox or blackbox.

whitebox generates test inputs using the whitebox fuzzing algorithm of Section 2.1. We use an existing tool, named SAGE, that implements this algorithm for x86 Windows applications [16].

whitebox+tokens extends whitebox fuzzing with only the lexical part of the grammar, i.e., marks token identifiers as symbolic, instead of individual input bytes, but does not use a grammar. This strategy was implemented as an extension of SAGE.

grammar-based whitebox is the grammar-based whitebox fuzzing algorithm of Section 2, which extends whitebox fuzzing both using symbolic tokens and an input grammar. This strategy was also implemented as an extension of the SAGE tool.

Figure 3 tabulates the strategies used in the evaluation and shows their characteristics.

Other strategies are conceivable. For example, whitebox fuzzing could be combined directly with the grammar, without tokens. Doing so requires transforming the grammar into a scannerless grammar [33]. Another possible strategy is bounded exhaustive enumeration [23, 36]. We have not included the latter in our evaluation because, while all other strategies we evaluated can be time-bounded (i.e., can be stopped at any time), exhaustive enumeration up to some input length is biased if terminated before completion, which makes it hard to fairly compare to time-bounded techniques.

3.3 Methodology

To avoid bias when using test generation strategies that require seed inputs (see Figure 3), we use 50 seed inputs with 15 to 20 tokens generated randomly from the grammar. Section 3.4 provides more information about selecting the size of seed inputs. Also, to avoid bias across all strategies, we run all experiments inside the same test harness.

The *whitebox+tokens* and *grammar-based whitebox* strategies require identifying the tokenization function that creates grammar tokens. Our implementation 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

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 4. 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.

in the module of interest. This coverage metric seems the most suitable for our needs, since 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 instruction coverage for the interpreter, we also measure coverage in the lexer, parser and code generator modules.

We run each test input 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 much longer, until instruction coverage does not increase during the last 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 several years by the developers and testers of this JavaScript interpreter. The suite consists of more than 2,800 hand-crafted inputs that exercise the interpreter thoroughly.

3.4 Seed Size Selection

Four of our generation strategies require seed inputs (Figure 3). 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 test input 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 code generator. For each length, we generate 100 inputs and perform the measurements only for those inputs. Figure 4 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 there are appropriate line breaks in the file[3]. However, the grammar does not enforce this requirement and allows omitting all semicolons.

By analyzing the results in Figure 4, we select 15 to 20 as the size range, in tokens, of the input 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 5 tabulates the coverage and reachability results for the 2-hour runs with each of the five automated test generation strategies previously discussed. For comparison, results obtained with the manually-written test suite are also included, even though running it requires more than 2 hours (as those 2,820 input JavaScript programs are typically much larger and takes more time to be processed).

Among all the automated test generation strategies considered, *grammar-based whitebox* achieves the best total coverage as well as the best coverage in the deepest examined module, the code generator. It achieves results that are closest to the manual test suite, which predictably provides the best coverage. The manual suite is diverse and extensive, but was developed with the cost of many man-months of work. In contrast, *grammar-based whitebox* requires minimal human effort, and quickly generates relatively good test inputs.

We can also observe the following.

- *Grammar-based whitebox* fuzzing achieves much better coverage than regular *whitebox* fuzzing.
- *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.
- *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* strategies achieved similar results in all categories. This shows that, when testing applications with highly-structured inputs in a limited amount of time (2 hours), whitebox fuzzing, with the power of 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-reaching 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

[3] See 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 test suite	2820	58.8	100	62.1	100	76.4	100	91.6

Figure 5. Coverage results for 2-hour runs. The *seed inputs* row lists statistics for the 50 seed inputs used by some of the test generation strategies (see Sections 3.3 and 3.4). The *manual test suite* takes more than 2 hours to run and is included here for reference. The "inputs" column gives the number of inputs tested by each strategy . 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.

strategy *S*	only *S*	*S* and GBW	only GBW
blackbox	4.9	62.5	32.6
grammar-based blackbox	2.2	56.2	41.6
whitebox	7.2	61.3	31.5
whitebox+tokens	10.9	62.0	27.1

Figure 6. Relative coverage in % 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".

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, as it reaches the highest coverage and keeps discovering new code for the longest than the other techniques (97 hours for *grammar-based whitebox* vs. 84 hours for *whitebox* and 82 hours for *grammar-based blackbox*).

In summary, the results of these experiments validate our claim that grammar-based whitebox fuzzing is effective in reaching deeper into the tested application and exercising the code more thoroughly than other automated test generation strategies.

3.5.2 Relative Coverage

Figure 6 compares the instructions covered with *grammar-based whitebox* fuzzing and the other analyzed strategies. The numbers show that the inputs generated by *grammar-based whitebox* cover most of the instructions covered by the inputs generated by the other strategies (see the small numbers in the column "only *S*"), while covering many other instructions (see the large numbers in the column "only GBW").

Combined with the results of Section 3.5.1, this shows that *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. In other words, *grammar-based whitebox* creates tests inputs of the highest quality among the analyzed strategies.

3.5.3 Statistics on Symbolic Executions

Figure 7 presents various statistics related to the symbolic executions performed during the 2 hours runs of each of the three whitebox strategies evaluated. We make the following observations.

- All three whitebox strategies perform roughly the same number of symbolic executions.
- However, the *whitebox* strategy creates a larger average number of symbolic variables because it operates on characters, while the other two strategies work on tokens (cf. Figure 3).
- The *whitebox+tokens* strategy creates the smallest average number of symbolic variables per execution. This is because *whitebox+tokens* generates many unparsable inputs (cf. Figure 5), which the parser rejects early and therefore no symbolic variables are created for the tokens after the parse error.

Figure 7 also 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 avoiding path explosion in that module. Those strategies do explore the lexer (indeed, Figure 5 shows high coverage) but they do not get lost in its error-handling 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, like the code generator, operates on the AST. When processing the AST in later stages, symbolic variables associated with input bytes or tokens are largely absent, so symbolic execution does not create constraints from code branches in these stages. The number of symbolic constraints in those deeper stages could be increased by associating symbolic variables with other values returned by the tokenization function such as string and integer values associated with some tokens.

3.5.4 Context-Free Constraint 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 obvious correlation between seed input size and the average number of solver calls). The results are that up to 58% of total execution time was spent in the constraint solver (with no obvious correlation between seed size and solving time). Such solving times are typical in dynamic test generation (e.g., a recent report [15] indicates up to 62% of the test generation time spent in the constraint solver).

strategy	created constraints %			symbolic executions	average number of symbolic variables	average number of constraints
	lexer	parser	code gen.			
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 7. Symbolic execution 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 2-hour 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

Systematic dynamic test generation [6, 14] is becoming increasingly popular [2, 16, 34] because it finds bugs without generating false alarms and requires 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 [28] generated streams of random bytes, but most popular fuzzers today support some form of grammar representation, e.g., SPIKE[4], Peach[5], FileFuzz[6], Autodafé[7]. Sutton *et al.* present a survey of fuzzing techniques and tools [37]. Work on grammar-based test input generation started in the 1970's [17, 32] and can be broadly divided into random [8, 24, 25, 35] and exhaustive generation [21, 23]. Imperative generation [7, 10, 30] 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 [19]). Grammar-based test input generation is an example of model-based testing (see Utting *et al.* for a survey [38]), 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 blackbox approaches, our approach analyses the code of the program under test and derives new test inputs from it.

Path explosion in systematic dynamic test generation can be alleviated by performing test generation compositionally [13], 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 infinitely many paths or limited symbolic reasoning capability when analyzing the lexer/parser. Grammar-based whitebox fuzzing and test summaries are complementary techniques which 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 [22]. 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 [23] is closest to ours. These authors combine grammar-based blackbox 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 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 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 testing with grammar-based blackbox testing, and leverages the strengths of both.

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 5). In our experiments, grammar-based whitebox fuzzing strongly outperformed both whitebox fuzzing and blackbox fuzzing. Code generator coverage improved from 61% to 81% and deep reachability improved from 72% to 80%. Deep parts of the application are the hardest to test automatically and our technique shows how to address this.

Since grammars are bound to be partial specifications of valid inputs, grammar-based blackbox approaches are fundamentally limited. Thanks to whitebox dynamic test gener-

[4] `http://www.immunitysec.com/resources-freesoftware.shtml`

[5] `http://peachfuzz.sourceforge.net/`

[6] `http://labs.idefense.com/software/fuzzing.php`

[7] `http://autodafe.sourceforge.net`

ation, some of this incompleteness can be recovered, which explains why grammar-based whitebox fuzzing also outperformed grammar-based blackbox fuzzing in our experiments.

Acknowledgments

We thank Michael D. Ernst and Danny Dig for comments on drafts of this paper. We also thank the anonymous reviewers for their suggestions to improve the presentation. The work of Adam Kieżun was done mostly while visiting Microsoft.

References

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

[2] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic Web applications. Technical Report MIT-CSAIL-TR-2008-006, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, Feb. 2008.

[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. In *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. In *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] 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.

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

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

[15] P. Godefroid, M. Levin, and D. Molnar. Active property checking. Technical Report MSR-TR-2007-91, Microsoft, 2007.

[16] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

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

[18] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley Series in Computer Science, 1979.

[19] S. Khurshid and D. Marinov. TestEra: Specification-Based Testing of Java Programs Using SAT. In *ASE*, 2004.

[20] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

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

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

[24] B. Malloy and J. Power. An interpretation of Purdom's algorithm for automatic generation of test cases. In *ICIS*, 2001.

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

[26] 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.

[27] D. Melski and T. Reps. Interconvertbility of set constraints and context-free language reachability. In *PEPM*, 1997.

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

[29] 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.

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

[31] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: a yacc for writing application protocol parsers. In *IMC*, 2006.

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

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

[34] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *FSE*, 2005.

[35] E. Sirer and B. Bershad. Using production grammars in software testing. In *DSL*, 1999.

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

[37] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[38] 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.

[39] G. Wassermann and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *PLDI*, 2007.