IFuzzer: An Evolutionary Interpreter Fuzzer using Genetic Programming

Spandan Veggalam[1], Sanjay Rawat[2,3], Istvan Haller[2,3] and Herbert Bos[2,3]

[1] International Institute of Information Technology, Hyderabad, India
[2] Computer Science Institute, Vrije Universiteit Amsterdam, Netherlands
[3] Amsterdam Department of Informatics
veggalam.s@research.iiit.ac.in, s.rawat@vu.nl, i.haller@student.vu.nl, herbertb@cs.vu.nl

Abstract. We present an automated *evolutionary fuzzing* technique to find bugs in JavaScript interpreters. Fuzzing is an automated black box testing technique used for finding security vulnerabilities in the software by providing random data as input. However, in the case of an interpreter, fuzzing is challenging because the inputs are piece of codes that should be syntactically/semantically valid to pass the interpreter's elementary checks. On the other hand, the fuzzed input should also be *uncommon enough* to trigger exceptional behavior in the interpreter, such as crashes, memory leaks and failing assertions. In our approach, we use evolutionary computing techniques, specifically *genetic programming*, to guide the fuzzer in generating uncommon input code fragments that may trigger exceptional behavior in the interpreter. We implement a prototype named **IFuzzer** to evaluate our technique on real-world examples. IFuzzer uses the language grammar to generate valid inputs. We applied IFuzzer first on an older version of the JavaScript interpreter of Mozilla (to allow for a fair comparison to existing work) and found 40 bugs, of which 12 were exploitable. On subsequently targeting the latest builds of the interpreter, IFuzzer found 17 bugs, of which four were security bugs.

Keywords: Fuzzing, System Security, Vulnerability, Genetic Programming, Evolutionary Computing

1 Introduction

Browsers have become the main interface to almost all online content for almost all users. As a result, they have also become extremely sophisticated. A modern browser renders content using a wide variety of interconnected components with interpreters for a growing set of languages such as JavaScript, Flash, Java, and XSLT. Small wonder that browsers have turned into prime targets for attackers who routinely exploit the embedded interpreters to launch sophisticated attacks [1]. For instance, the JavaScript interpreter in modern browsers (e.g., SpiderMonkey in Firefox) is a widely used interpreter that is responsible for many high-impact vulnerabilities [2]. Unfortunately, the nature and complexity of these interpreters is currently well beyond state-of-the-art bug finding techniques, and therefore, further research is necessary [3]. In this paper, we propose a novel evolutionary fuzzing technique that explicitly targets interpreters.

Fuzz testing is a common approach for finding vulnerabilities in software [4–8]. Many fuzzers exist and range from a simple random input generator to highly sophisticated testing tools. For instance, in this paper, we build on evolutionary fuzzing which has proven particularly effective in improving fuzzing efficiency [5, 9, 10] and makes use of evolutionary computing to generate inputs that exhibit vulnerabilities. While fuzzing is an efficient testing tool in general, applying it to interpreters brings its own challenges. Below, we list a few of the issues that we observed in our investigations:

1. Traditionally, fuzzing is about mutating *input* that is manipulated by a software. In the case of the interpreter, the input is *program (code)*, which needs to be mutated.
2. Interpreter fuzzers must generate syntactically valid inputs, otherwise, inputs will not pass the elementary interpreter checks (mainly the parsing phase) and testing will be restricted to the input checking part of the interpreter. Therefore, the input grammar is a key consideration for this scenario. For instance, if the JavaScript interpreter is the target, the fuzzed input *must* follow the syntax specifications of the JavaScript language, lest the inputs be discarded early in the parsing phase.
3. An interpreter may use a somewhat different (or evolved) version of the grammar than the one publicly known. These small variations are important to consider when attempting fuzzing the interpreter fully.

Genetic Programming is a variant of evolutionary algorithms, inspired by biological evolution and brings transparency in making decisions. It follows Darwin's theory of evolution and generates new individuals in the eco-system by recombining the current characteristics from individuals with the highest fitness. Fitness is a value computed by an objective function that directs the evolution process. Genetic Programming exploits the modularity and re-usability of solution fragments within the problem space to improve the fitness of individuals. This approach has been shown to be very appropriate for generating code fragments [11–13], but hasn't been used for fuzz-testing in general as program inputs are typically unstructured and highly inter-dependent. However, our key insight is that, as described before, interpreter fuzzing is a special case. Using code as input, Genetic Programming seems like a natural fit!

In this paper, we introduce a framework called *IFuzzer*, which generates code fragments using Genetic Programming [14]—allowing us to test interpreters by following a black-box fuzzing technique and mainly looks for vulnerabilities like memory corruptions. IFuzzer takes a language's context-free grammar as input for test generation. It uses the grammar to generate parse trees and to extract code fragments from a given test-suite. For instance, IFuzzer can take the JavaScript grammar and the test-suite of the SpiderMonkey interpreter as input and generate parse trees and relevant code fragments for in-depth testing. IFuzzer leverages the fitness improvement mechanism within Genetic Programming to improve the quality of the generated code fragments.

Figure 1 describes the overview of IFuzzer. The fuzzer takes as input a test suite, a language grammar and sample codes. The parser module uses the lan-

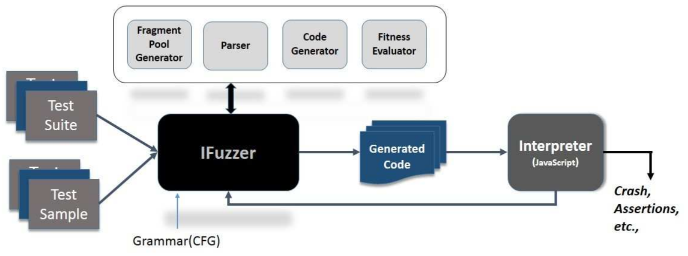

Fig. 1: Overview of IFuzzer Approach

guage grammar to parse the program and generates an abstract syntax tree. The fragment pool extractor generates a pool of code fragments extracted from a set of sample code inputs for different nodes (Non-Terminals) in the grammar. The code generator generates new code fragments by performing genetic operations on the test suite. The interpreter executes all the generated code fragments. Based on the feedback from the interpreter, the fragments are evaluated by the fitness evaluator and accordingly used (or discarded) for future generations of inputs. We evaluated IFuzzer on two versions of Mozilla JavaScript interpreter. Initially, we configured it to target SpiderMonkey 1.8.5 in order to have a comparison with LangFuzz [3], a state-of-art mutation fuzzer for interpreter testing. In another experiment, we configured IFuzzer to target the latest builds of SpiderMonkey. Apart from finding several bugs that were also found by LangFuzz, IFuzzer found *new exploitable* bugs in these versions.

In summary, this paper makes the following contributions:

1. We introduce a fully automated and systematic approach for code generation for interpreter testing by *mapping the problem of interpreter fuzz testing onto the space of evolutionary computing for code generation*. By doing so, we establish a path for applying advancements made in evolutionary approaches to the field of interpreter fuzzing.
2. We show that Genetic Programming techniques for code generation result in a diverse range of code fragments, making it a very suitable approach for interpreter fuzzing. We attribute this to inherent randomness in Genetic Programming.
3. We propose a *fitness function* (objective function) by analyzing and identifying different code parameters, which guide the fuzzer to generate inputs which can trigger uncommon behavior within interpreters.
4. We implement these techniques in a full-fledged (to be) open sourced fuzzing tool called *IFuzzer* that can target any language interpreter with minimal configuration changes.
5. We show the effectiveness of IFuzzer empirically by finding new bugs in Mozilla's JavaScript engine SpiderMonkey—including several exploitable security vulnerabilities.

The rest of the paper is organized as follows. Section 2 presents the motivation for choosing Genetic Programming for code generation. We explain the implementation of IFuzzer in section 3. Section 4 discusses the experimental set-up and evaluation step of IFuzzer and section 7 concludes the work with comments on possible future work.

2 Genetic Programming

Evolutionary algorithms build up a search space for finding solutions to optimization problems, by evolving a population of individuals. An *objective function* evaluates the fitness of these individuals and provides feedback for next generations of individuals. These algorithms build on the Darwinian principle of natural selection and biologically inspired genetic operations. In prior work, Genetic Algorithms proved successful in the generation of test cases [13, 15].

Genetic programming (GP) [14, 16] achieves the goal of generating a population by following a similar process as that of most genetic algorithms, but it represents the individuals it manipulates as tree structures. Out of the many variants of GP in the literature, we follow Grammar-based Genetic Programming (GGP). In GGP, we consider programs, that are generated based on the rules formulated in the grammar (context free), as the individuals and represent them by parse trees. This procedure is a natural fit for the interpreters. All the individuals in a new generation are the result of applying the genetic operators—crossover and mutation—on the parse tree structures.

Search Space: The search space is the set of all feasible solutions. Each point in the space represents a solution defined by the fitness values or some other values related to an individual. Based on fitness constraints, the individual with highest fitness is considered the best feasible solution.

Bloating: Bloating [16] is a phenomenon that adversely affects input generation in evolutionary computing. There are two types of bloating: structural and functional bloating.

- *Structural Bloating*: While iterating over generations, after a certain number of generations, the average size of individuals (i.e. the code) grows rapidly due to uncontrolled growth [17]. This results in inefficient code, while the growth hardly contributes to the improvement of fitness. Moreover, large programs require more computation to process.

- *Functional Bloating*: In functional bloating [18], a range of fitness values become narrow and thereby reduces the search space. However, it is common to have different individuals with the same fitness, because after some time bloating makes everything look linear. As a result, it becomes hard to distinguish individuals.

As the process of fuzzing may run for a very long period, neglecting or failing to handle the bloating problem may lead to very unproductive results.

2.1 Representation of the Individuals

We consider input code to be the individuals manipulated by GP. Each individual is represented by its parse tree, generated using the corresponding language grammar. IFuzzer performs all its genetic operations on these parse trees and generates new individuals (input code) from the resulting parse trees. Figure 2 illustrates an example of valid program for the simple language grammar (Listing 1.1) and the corresponding parse tree derived.

⟨*statement*⟩ ::= ⟨*variable Statement*⟩*
⟨*variable Statement*⟩ ::= `var` ⟨*identifier*⟩ ⟨*initializer*⟩?
⟨*initializer*⟩ ::= `=` ⟨*numLiteral*⟩ | ⟨*identifier*⟩
⟨*identifier*⟩ ::= [a-zA-Z0-9]*
⟨*numLiteral*⟩ ::= [0-9]*

Listing 1.1: Example of a Simple Language Grammar

```
var s=10
var b
var a=b
```

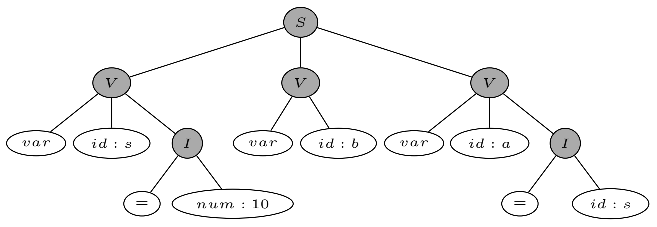

Fig. 2: Example of a syntactically valid program and its derived parse tree

2.2 Fragment Pool

The fragment pool is a collection of code fragments for each non-terminal in the grammar specification. We can tag each possible code fragment in a program with a non-terminal in the grammar specification. Using the parser, IFuzzer parses all the input files in the test suite and extracts the corresponding code fragments for different non-terminals. With a sufficient number of input files, we can generate code fragments for all non-terminals in the language grammar. It stores these code fragments in tree representations with the corresponding non-terminal as root. At a later stage, it uses these code fragments for mutation and code generation. The same process of generating parse trees is followed in the crossover operation for identifying code fragments for selected common non-terminal between the participating individuals. An example of a fragment pool for the derived parse tree, summarized in Figure 2, is shown in the box below.

```
(S) <statement> = {var s=10,var b,var a=s}
(V) <variableStatement>
        = {var s=10,var b,var a=s}
(I) <initializer> = {= 10,=s}
terminals = {id:s,id:a,id:b,num:10,var,=}
```

3 Implementation

We implement IFuzzer as a proof-of-concept based on the methods discussed in the previous sections. It works as described in the overview diagram of Figure 1 and in the following, we elaborate on IFuzzer's individual components.

3.1 Code Generation

In this section, we explain various genetic operators that IFuzzer uses for input generation. After each genetic operation, the objective function, discussed in section 3.3, evaluates the fitness of the offspring.

IFuzzer uses the ANTLR parser for the target language and generates the parser using the ANTLR parser generator framework [19] with the language grammar as input. The initial population, the fragment pool generation (discussed in section 2), and the crossover and mutation operations all make use of parse tree returned by the parser.

Initial Population. The initial population of individuals consists of random selection of programs, equal to the population size, from the input test samples. This forms the first generation. After each generation, individuals from the parent set undergo genetic operations and thereby evolve into offspring.

Mutation. During mutation, IFuzzer selects random code fragments of the input code for replacement. It performs replacement by selecting a random member of the fragment pool which corresponds to the same non-terminal, or by generating a new fragment using an expansion algorithm. Our expansion algorithm assumes that all the production rules have equal and fixed probabilities for selection. We use the following expansion algorithm:

1. Select the non-terminal n from the parse tree to expand.
2. From the production rules of the grammar, select a production for n and replace it with n.
3. Repeat the following steps up to *num* iterations.
   (a) Identify a random set N of non-terminals in the resulting incomplete parse tree.
   (b) Extract a set of production rules P_n, for the selected non-terminal n, from the production rules P (i.e., $P_n \subseteq P$) listed in the grammar specification.
   (c) Select a production $P_{selected}$ randomly for each identified non-terminal $\in N$.
   (d) Replace the non-terminals occurrence with $P_{selected}$.
4. After expansion, replace all remaining occurrences of non-terminals with corresponding code fragments, selected randomly from the fragment pool. Note that steps 3 & 4 also solve the problem of non-termination in the presence of mutually recursive production rules.

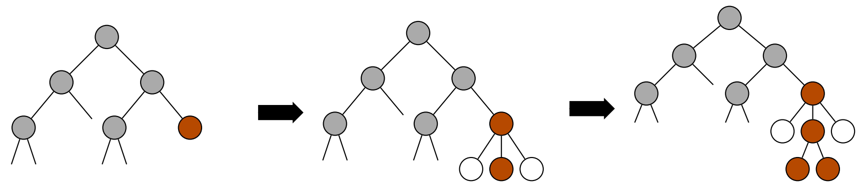

Fig. 3: Example of stepwise expansion on the parse tree: all the dark nodes represent non-terminals and white nodes represent terminals. A particular node is selected and expanded as shown.

Figure 3 illustrates an example of the expansion algorithm. Dark nodes in the parse tree represent the non-terminals and white nodes represent the terminals. A dark node from the parse tree is selected during the mutation process and is expanded to the certain depth (*num*) as discussed above. This algorithm does not yield a valid expansion with more iterations. After expansion, we may still have unexpanded non-terminals. IFuzzer handles this by choosing *code fragments* from the fragment pool and replaces remaining non-terminals by such code fragments, which are represented by the same non-terminals. In this way, the tree converges with terminals and results in a valid parse tree.

Crossover. During crossover, for a given pair of individuals (parents), IFuzzer selects a common non-terminal n from parse trees of the individuals and extracts random fragments, originating from n, from both the individuals. These selected fragments from one individual are exchanged with fragments of another individual, thereby generating two new parse trees. Using these trees, IFuzzzer generates two new offsprings.

Replacement. During the process of offspring generation, it is important to retain the features of the best individuals (parents) participating in evolution. Therefore, IFuzzer adopts the common technique of *fitness elitism* to retain the best individuals among the parents in the next generation. IFuzzer generates the remaining population in the next generation by crossover and mutation. Elitism prevents losing the best configurations in the process.

Reusing Literals. The code generation operations may result in semantically invalid fragments or a loss of context. For instance, after a modification a statement in the program may use an identifier `a` which is not declared in this program. Introducing language semantics will tie IFuzzer to a language specification and we therefore perform generic semantic improvements at the syntactic level. Specifically, IFuzzer reduces the errors due to undeclared identifiers by renaming the identifiers around the modification points to the ones declared elsewhere in the program. Since it knows the grammar rules that contain them, IFuzzer can easily extract such identifiers from the parse tree automatically. In our example of the undeclared variable a, it will mapped it to another identifier `b` declared elsewhere and replace all occurrences of `a` with `b`.

3.2 Bloat Control

Bloat control pertains to different levels [20] and IFuzzer uses it during the fitness evaluation and breeding stages:

Stage 1: Fitness Evaluation. Applying bloat control at the level of fitness evaluation is a common technique. In IFuzzer, we use *parsimony pressure* [21,22] to alter the selection probability of individuals by penalizing larger ones.

Calculating Parsimony Coefficient: The parsimony co-efficient $c(t)$ at each generation t is given by the following correlation coefficient [23].

$$c(t) = \frac{Covariance(f,l)}{Variance(l)} = \frac{\sum_{i=0}^{n}(f_i - \bar{f})(l_i - \bar{l})}{n-1} \times \frac{n-1}{\sum_{i=0}^{n}(l_i - \bar{l})^2} \tag{1}$$

where $\bar{l}$ and $\bar{f}$ are the mean fitness and length of all individuals in the population, and f_i and l_i are the original fitness and length of an individual i. $Covariance(f,l)$ calculates the co-variance between an individual's fitness and length, while $Variance(l)$ gives the variance in the length of the individuals. In Section 3.3, we will see that IFuzzer uses the parsimony coefficient to add penalty to the fitness value.

Stage 2: We also apply bloat control at the breeding level by means of fair size generation techniques [16].*Fair Size Generation* limits the growth of the offspring's program size. In our approach, we restrict the percentage of increase in program size to a biased value:

$$length_{generated\_code}/length_{original\_code} < bias_{threshold}$$

where $length_x$ gives information about the number of non-terminals in the parse tree x and $bias_{threshold}$ is the threshold value for fair size generation. This restricts the size of code and if the generated program fails to meet this constraint, IFuzzer discards as invalid. In that case, it re-generates the program using the same GA operator with which it started. After a certain number of failed attempts, it discards the individual completely and excludes it from further consideration for offspring generation.

Finally, we use *Delta debugging* algorithm [24, 25] to determine the code fragments that are relevant for failure production and to filter out irrelevant code fragments from the test cases, further reduces the size of test case. This essentially results in part of the test case that is relevant to the failure [26]. The same algorithm reduces the number of lines of code executed and results in suitably possible valid small test case.

3.3 Fitness Evaluation

The evolutionary process is an objective driven process and the fitness function that defines the objective of the process plays a vital role in the code generation process. After crossover and mutation phases, the generated code fragments are evaluated for fitness.

As IFuzzer aims to generate uncommon code structures to trigger exceptional behavior, we consider both *structural* aspects and interpreter *feedback* of the generated program as inputs to the objective function. The interpreter feedback includes warnings, execution timeouts, errors, crashes, etc.—in other words, the goal itself. Moreover, during the fitness evaluation, we calculate structural metrics such as the cyclomatic complexity for the program. The cyclomatic complexity [27] gives information about the structural complexity of the code. For instance, nested (or complex) structure has a tendency to create uncommon behavior [28], so such structures have higher scores than less complex programs.

At its core, IFuzzer calculates the base fitness value $f_b(x)$ of an individual x as the sum of its structural score ($score_{structure}$) and its feedback score ($score_{feedback}$).

$$f_b(x) = score_{structure} + score_{feedback}$$

Finally, as discussed in section 3.2, IFuzzer's bloat control re-calculates the fitness with a penalty determined by the product of its parsimony co-efficient c and the length of the individual l:

$$f_{final}(x) = f_b(x) - c * (l(x))$$

where $f_{final}(x)$ is the updated fitness value of an individual x.

Parameters IFuzzer contains many adjustable GP and fitness parameters, including the mutation rate, crossover rates, population size, and the number of generations. In order to arrive at a set of optimal values, we ran application (to be tested) with various combinations of these parameters and observed for properties like input diversity, structural properties etc. We adhere to the policy that higher the values of such properties, better is the combination of parameters. In the experiments, we use the best combination based on observations, made during a fixed profiling period. We, however, note that it should be possible to fine tune all these parameters further for optimal results.

4 Experimentation

In this section, we evaluate the effectiveness of IFuzzer by performing experimentation on real-world applications. IFuzzer is a cross platform tool, which runs on UNIX and Windows operating systems. All the experiments were performed on a standalone machine with a configuration of Quad-Core 1.6Ghz Intel i5-4200 CPU and 8GB RAM. The outcome of our experiments aims to answer the following questions:

1. Does IFuzzer perform better than the known state-of-art tools? What is the effectiveness of IFuzzer?

2. What are the benefits of using GP? What drives GP to reach its objective?
3. Does our defined objective function encourage the generation of uncommon code?
4. How important is it that IFuzzer generates uncommon code? How is this related to having high coverage of the interpreter?

In order to answer the questions mentioned above, we performed two experiments. In the first experiment, we evaluate IFuzzer and compare it against the state-of-the-art LangFuzz using the same test software [3]. In the second experiment, we run IFuzzer against the latest build of SpiderMonkey. We have also run IFuzzer with different configurations in order to evaluate the effect of separate code generation strategies. Results of these experiments are in the Appendix.

We also ran IFuzzer on Chrome JavaScript engine V8 and reported few bugs. However, our reported-bugs were not accepted as *security bugs* by the Chrome V8 team and therefore, we do not report them in detail in this paper. In order to establish the usability of IFuzzer to other interpreters, we could configure IFuzzer for Java by using Java Grammar Specifications, available at [29]. However, we have not tested this environment to its full extent. The main intention of performing this action is to show the flexibility of IFuzzer to other grammars.

4.1 Testing Environment

In our experiments, we used the Mozilla development test suite as the initial input set. The test suite consists of $\tilde{3}000$ programs chosen from a target version. We used the same test suite for fragment pool generation and program generation. Fragment Pool generation is a one-time process, which reads all programs at the start of the fuzzing process and extracts fragments for different non-terminals. We assume that the test suite involves inputs (i.e. code fragments) that have been used in testing in the past and resulted in triggering bugs. We choose *SpiderMonkey* as the target interpreter for JavaScript. We write input grammar specification from the ECMAScipt standard specification and grammar rules from the ECMAScript 262 specification [30].

4.2 IFuzzer vs LangFuzz

Our first experiment evaluated IFuzzer by running it against interpreters with the aim of finding exploitable bugs and compare our results to those of LangFuzz. We compare in terms of time taken in finding bugs and the extent of the overlap in bugs found by both the fuzzers. Since we do not have access to the LangFuzz code, we base our comparison on the results reported in [3]. For a meaningful comparison with LangFuzz, we chose SpiderMonkey 1.8.5 as the interpreter as this was the version of SpiderMonkey that was current when LangFuzz was introduced in [3].

During the experiment on SpiderMonkey 1.8.5 version, IFuzzer found 40 bugs in a time span of one month, while Langfuzz found 51 bugs in 3 months. More importantly, when comparing the bugs found by the two fuzzers, the overlap is "only" 24 bugs. In other, a large fraction of the bugs found by IFuzzer is unique.

With roughly 36% overlap in the bugs, IFuzzer clearly finds different bugs–bugs that were missed by today's state-of-the-art interpreter fuzzer—in comparable time frames.

We speculate that IFuzzer will find even more bugs if we further fine-tune its parameters and run it for a longer period. We also notice that there are many build configurations possible for SpiderMonkey, and Langfuzz tries to run on all such possible build configurations. In contrast, due to resource constraints, we configured IFuzzer to run only on two such different configurations (with and without enabling debugging). Trying more configurations may well uncover more bugs [31]

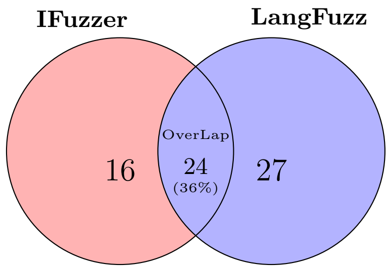

Fig. 4: Number of defects found by IFuzzer (40) and LangFuzz (51) in SpiderMonkey version 1.8.5

In order to determine the severity of the bugs, we investigated them manually with gdb-exploitable [32]—a widely used tool for classifying a given application crash file (core dump) as *exploitable* or non-exploitable. Out of IFuzzer's 40 bugs, gdb-exploitable classified no fewer than 12 as *exploitable*.

Example of a defect triggered by IFuzzer: Listing 1.2 shows an example of a generated program triggering an assertion violation in SpiderMonkey 1.8.5. The JavaScript engine crashes at line 6, as it fails to build an argument array for the given argument value `abcd*&^%$$`. Instead, one would expect an exception or error stating that the argument as invalid.

```
1 if (typeof options == "function")
2 { var opts = options();
3 if (!/\bstrict\b/.test(opts))
4         options("strict");
5 if (!/\bwerror\b/.test(opts))
6         options('abcd*&^%$$');
7 }
```

Listing 1.2: A test Case generated by IFuzzer, which crashes the SpiderMonkey JavaScript engine with an internal assertion upon executing line 6

```
function test(code) {
f = eval("(function(){"
        + code + "})")
        f()
}
test("x=7");
test("\"use strict\";
for(d in[x=arguments]){}");
test("for(v in((Object.seal)(x)));
x.length=Function")
```

Listing 1.3: Test Case generated by IFuzzer, that crashes the SpiderMonkey JavaScript engine 1.8.5.

Another example (shown in Listing 1.3) exposes security issues in SpiderMonkey

1.8.5, which is related to strict mode changes to the JavaScript semantics [33]. Line 8 enables the strict mode which makes changes to the way, SpiderMonkey executes the code. On execution, the JavaScript engine crashes due to an access violation and results in a stack overflow.

Bug ID	**Description**
1131986	Segmentation fault
1133247	OOM error is not reported in the browser console
1141969*	Crash[@js::SCInput::SCInput] or assertion failure: (nbytes & 7) == 0 at Structured-Clone.cpp:463
1192381*	Crash due to Assertion failure: input()-> isRecoveredOnBailout() == mustBeRecovered_ (assertRecoveredOnBailout failed during compilation), at js/src/jit/Recover.cpp:1465
1192379	input()->isRecoveredOnBailout()
1192401	
(CVE-2015-4507)	Crash due to Assertion failure: getSlotRef(EVAL).isUndefined(), at js/src/vm/GlobalObject.h:147
1193307	evaluate() method results in ”Error: compartment cannot save singleton anymore”
1205603	crash due to uncaught exception: out of memory
1205605	InternalError: too much recursion
1234323	AddressSanitizer failed to allocate 0x001000000000 bytes. AddressSanitizer's allocator is terminating the process instead of returning 0
1248188*	Crash due to Assertion Failure : Could not allocate ObjectGroup in EnsureTrackPropertyTyp
1234979	Segmentation fault at js/src/jsobj.h:122
1235122	AddressSanitizer failed to allocate 0x400002000 (17179877376) bytes of LargeMmapAllocator
1235160	crash due to Assertion failure: index < length_, at js/src/jit/FixedList.h:84
1247231*	Segmentation fault at js/src/vm/NativeObject.h:86
1248321*	Crash due to Assertion failure: JSScript::argumentsOptimizationFailed, at js/src/jscntxt.cpp:12
1258189	Crash due to Assertion failure: isLive(), at js/src/build1/dist/include/js/HashTable.h:774

Table 1: Bugs found in the latest version of Mozilla's SpiderMonkey.

4.3 Spidermonkey Version 38

We also ran an instance of IFuzzer to target SpiderMonkey 38 (latest version at the time of experimentation). Table 1 shows the results of running IFuzzer on latest build. IFuzzer detected 17 bugs and out of these, 4 were confirmed to be exploitable. Five of the crashes (marked with $*$) are due to assertion failures (which may be fixed in subsequent versions), unhandled out of memory crashes, or spurious crashes that we could not reproduce. The remaining ones are significant bugs in the interpreter itself.

For instance, the following code looks to be an infinite loop, except that one of the interconnected components may fail to handle the memory management and, hence the JavaScript engine keeps consuming the heap memory, creating a denial of service by crashing the machine in few seconds. The code fragment responsible for the crash is shown in Listing 1.4.

```
try {
  a = new Array() ;
  while(1)
        a = new Array(a) ;
}
catch (e) { }
```

Listing 1.4: Test Case generated by IFuzzer, crashing the latest version XXX of SpiderMonkey JavaScript engine. JavaScript engine fails to handle the situation, leading to a memory leak

Also, in this case, our contribution and efforts were rewarded by the Mozilla's bounty program for one of the bugs detected by IFuzzer. The bug received an advisory from mozilla [34] and CVE Number CVE-2015-4507 and concerns a crash due to a `getSlotRef` assertion failure and application exit in the `SavedStacks` class in the JavaScript implementation. Mozilla classified its security severeness as “moderate”.

The results discussed so far establishes that the evolutionary approach followed by *IFuzzer* tool is capable of generating programs with a given objective and trigger significant bugs in real-world applications.

Other interpreters. When evaluating our work on the Chrome JavaScript engine V8, IFuzzer worked out of the box and reported few bugs that resulted in crash (see table 2). As far as we can tell, these bugs do not appear to be security bugs and require further scrutiny.

```
function f() {
    var s = "switch (x) {";
    for (var i = 8000; i < 16400; g++) {
        s += "case " + i + ": return " + i + "; break;";
    }
    s += "case 8005: return -1; break;";
    s += "}";
    var g = Func.tion("x", s);
    assertEq(g(8005), 8005);
}
f();
```

Listing 1.5: Test Case generated by IFuzzer, crashing the latest version XXX of Chrome V8 JavaScript engine. JavaScript engine fails to handle the situation crashes with an illegal instruction error.

1 - 2 crashes with Fatal error in `CALL_AND_RETRY_LAST` # Allocation failed - process out of memory
2 - few crashes due to NULL pointer exception

Table 2: IFuzzer crashes found on Chrome V8 [4.7.0]

In order to establish the usability of IFuzzer to other languages , we could further configure IFuzzer for Java by using Java Grammar Specifications, available at [29]. However, we have not fully tested this environment to its full extent.

5 Remarks on IFuzzer’s Design Decisions

Recall that IFuzzer uses an evolutionary approach for code generation by guiding the evolution process to generate uncommon code fragments. As stated earlier, there are several parameters available to fine-tune IFuzzer for better performance. For example, the choice of using a subset (of cardinality equal to the size of population) of the initial test suite, rather than the whole suite, as the first generation is to make an effective use of resources available. The remaining inputs from test suite can be used in later generation when IFuzzer gets stuck at some *local minima*, which is a known obstacle in evolutionary algorithms.

The generation in which a bug is identified depends on different factors, including the size of the input test sample, the size of the fragment considered for genetic operation and the size of new fragment induced etc. As discussed, the higher the complexity of inputs, the higher the probability of finding a bug. Bloat control and the time taken by the parser to process the generated programs (one of the fitness parameters) will restrict larger programs from making it into the next generations. IFuzzer does not completely discard larger programs, but deprioritizes them.

We also observed that almost all the bugs in SpiderMonkey 1.8.5 are triggered in the range of 3-120 generations with an average range of 35-40 generations. With the increase in complexity and number of language features added to the interpreter, the latest version requires more uncommonness to trigger the bugs, which implies more time to evolve inputs. As an example, all the bugs in the latest version are found on average after 90-95 generations.

While there are some similarities between LangFuzz and IFuzzer, the differences are significant. It is difficult to make a fair comparison on all aspects. IFuzzer's GP-based approach is a guided evolutionary approach with the help of a fitness function, whereas LangFuzz follows a pure mutation-based approach by changing the input and testing. IFuzzer's main strength is its feedback loop and the *evolution* of inputs as dictated by its new fitness function makes the design of IFuzzer very different from that of LangFuzz.

Both IFuzzer and LangFuzz are generative fuzzers that use grammars in order to be language independent but differ in their code generation processes. LangFuzz uses code mutation whereas IFuzzer uses GP for code generation. The use of GP provides IFuzzer the flexibility of tuning various parameters for efficient code generation.

Intuitively, the fitness function (objective function) is constructed to use the structural information about the program along with interpreter feedback information to calculate the fitness. Structural metrics, along with the interpreter feedback information, are also considered in the fitness calculation. Structural information is used to measure the singularity and complexity of the code generated. The chances of introducing errors are higher with larger and more complex code. Hence, the inputs that triggered bugs are not entirely new inputs but have evolved through generations starting from the initial test cases. We observed this evolutionary manifestation repeatedly during our experimentation.

In a nutshell, we observed that the *uncommonness* characteristic of the input code (like the structural complexity or the presence of type casting and type conversions) relates well with the possibility of finding exceptional behavior of the interpreter. Throughout this work, the driving intuition has been that most tests during development of the interpreter focused on the common cases. Therefore, testing the interpreter on uncommon ("weird") test cases should be promising as generating such test cases manually may not be straightforward and thereby some failure cases are missed.

6 Related Work

Fuzz testing was transformed from a small research project for testing UNIX system utilities [35] to an important and widely-adopted technique.

Researchers started fuzzers as brute forcing tools [36] for discovering flaws, after which they would analyze for the possibility of security exploitation. Later, the community realized that such simple approaches have many limitations in discovering complex flaws. Smart Fuzzer, overcame some of these limitations and proved more effective [37].

In 2001, Kaksonen [38] used an approach known as *mini-simulation*, a simplified description of protocols and syntax, to generate inputs that nearly match with the protocol used. This approach is generally known as a grammar-based approach. It provides the fuzzer with sufficient information to understand protocol specifications. Kaksonen's *mini-simulation* ensures that the protocol checksums are always valid and systematically checks which rules are broken. In contrast, IFuzzer uses the grammar *specification* to generate valid inputs.

Yang et al [39] presented their work on CSmith, a random "C program" generator. It uses grammar for producing programs with different features, thereby performing testing and analyzing C compilers. This process is a language independent fuzzer and uses semantic information during the generation process.

In the area of security, Zalewski presented ref_fuzz [40] and crossfuzz [41] aiming at the DOM component in browsers. JsFunFuzz [42] is a language-dependent generative tool written by Ruddersman in 2007, which targets JavaScript interpreters in web browsers, and has led to the discovery of more than 1800 bugs in SpiderMonkey. It is considered as one of the most relevant work in the field of web interpreters. LangFuzz, a language independent tool presented by Holler *et. al.* [3] uses language grammar and code mutation approaches for test generation. In contrast, IFuzzer uses grammar specification and code generation. Proprietary fuzzers include Google's ClusterFuzz [43] which tests a variety of functionalities in Chrome. It is tuned to generate almost 5 million tests in a day and has detected several unique vulnerabilities in chrome components.

However, all these approaches may deviate the process of code generation from generating the required test data, thereby degenerating into random search, and providing low code coverage. Feedback fuzzers, on the other hand, adjust and generate dynamic inputs based on information from the target system.

An example of feedback-based fuzzing is an *evolutionary fuzzer.* Evolutionary fuzzing uses evolutionary algorithms to create the required search space of data and operates based on an objective function that controls the test input generation. One of the first published evolutionary fuzzersis by DeMott et al. in 2007 [10] . This is a grey-box technique that generates new inputs with better code coverage to find bugs by measuring code block coverage.

Search-based test generation using metaheuristic search techniques and evolutionay computation has been explored earlier for generating test data [44, 45]. In the context of generating inputs using GP for code generation (as also adopted by IFuzzer), recent work by Kifetew *et.al.* [46] combines stochastic grammar with GP to evolve test suites for system-level *branch coverage* in the system under test.

Our approach differs from existing work in many aspects. First, our approach uses GP with a uniquely designed guiding objective function, directed towards generating uncommon code combinations—making it more suitable for fuzzing. In order to be syntactically correct but still *uncommon*, we apply several heuristics when applying mutation and crossover operations. Our approach is implemented as a language independent black box fuzzer. To the best of our knowledge, IFuzzer is the first prototype to use GP for interpreter fuzzing with very encouraging results on real-world application.

7 Conclusion and Future Work

In this paper, we elaborate on the difficulties of efficiently fuzzing an interpreter as well as our ideas to mitigate them. The main challenge comes from the fact that we need to generate *code* that is able to fuzz the interpreter to reveal bugs buried deep inside the interpreter implementation. Several of these bugs are found to be security bugs, which are exploitable, which makes an interpreter a very attractive target for future attacks.

In this work, we proposed an effective, fully automated, and systematic approach for interpreter fuzzing and evaluated a prototype, IFuzzer, on real-world applications. IFuzzer uses an evolutionary code generation strategy that applies to any computer language of which we have the appropriate grammar specifications and a set of test cases for the code generation process. IFuzzer introduces a novel objective function that helps the fuzzer to reach its goal of generating valid but uncommon code fragments in an efficient way. In our evaluation, we show that IFuzzer is fast at discovering bugs when compared with a state-of-the-art fuzzer of its class. IFuzzer found several security bugs in the SpiderMonkey JavaScript interpreter that is used in Mozilla browser. The approach used in this paper is generic enough for automated code generation for the purpose of testing any targeted language interpreters and compilers, for which a grammar specification is available and serves as a *framework* for generating fuzzers for any interpreted language and corresponding interpreters.

IFuzzer is still *evolving* and we envision avenues for further improvements. We plan to investigate more code (property) parameters to be considered for the fitness evaluation. In our experiments, we observed that the parameters for the genetic operations (mutation and crossover) should be tuned further to improve the evolutionary process. Another improvement can be to keep track of more information during program execution, which helps to guide the fuzzer in a more fine-grained manner. For example through dynamic program analysis we can gather information about the program paths traversed, which gives coverage information as well as correlation between program paths and the bugs they lead to. This information could be used to refine the fitness function, thus improving the quality of code generation.

Acknowledgments. This work was partially supported by Netherlands Organisation for Scientific Research through the NWO 639.023.309 VICI "Dowsing" project.

We would like to thank Mozilla Security Team and conference reviewers for their useful suggestions to improve the paper.

References

1. V. Anupam and A. J. Mayer, "Security of web browser scripting languages: Vulnerabilities, attacks, and remedies," in *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*, 1998.
2. O. Hallaraker and G. Vigna, "Detecting malicious javascript code in mozilla," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '05, pp. 85–94, 2005.
3. C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21th USENIX Security Symposium*, pp. 445–458, August 2012.
4. L. Guang-Hong, W. Gang, Z. Tao, S. Jian-Mei, and T. Zhuo-Chun, "Vulnerability analysis for x86 executables using genetic algorithm and fuzzing," in *Convergence and Hybrid Information Technology, 2008. ICCIT '08. Third International Conference on*, pp. 491–497, Nov 2008.
5. S. Rawat and L. Mounier, "An evolutionary computing approach for hunting buffer overflow vulnerabilities: A case of aiming in dim light," in *Proceedings of the 2010 European Conference on Computer Network Defense*, EC2ND '10, pp. 37–45, 2010.
6. S. Sparks, S. Embleton, R. Cunningham, and C. Zou, "Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pp. 477–486, 2007.
7. C. Del Grosso, G. Antoniol, E. Merlo, and P. Galinier, "Detecting buffer overflow via automatic test input data generation," *Comput. Oper. Res.*, vol. 35, pp. 3125–3143, Oct. 2008.
8. E. Alba and J. F. Chicano, "Software testing with evolutionary strategies," in *Proceedings of the Second International Conference on Rapid Integration of Software Engineering Techniques*, pp. 50–65, 2006.
9. M. Zalewski, "American fuzzy lop." At: http://lcamtuf.coredump.cx/afl/.
10. DeMott, Jared, Enbody, Richard, Punch, and W. F., "Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing,"
11. W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (Washington, DC, USA), pp. 364–374, IEEE Computer Society, 2009.
12. D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 802–811, IEEE Press, 2013.
13. G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.
14. R. I. McKay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'Neill, "Grammar-based genetic programming: a survey," *Genetic Programming and Evolvable Machines*, vol. 11, pp. 365–396, May 2010.
15. R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software Testing Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

16. R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*.
17. T. Soule, J. A. Foster, and Dickinson, "Code growth in genetic programming," in *Genetic Programming 1996:Proceedings of the First Annual Conference*, pp. 215–223, May 1996.
18. W. B. Langdon and R. Poli, "Fitness causes bloat: Mutation," in *Proceedings of Genetic Programming, First European Workshop, EuroGP*, pp. 37–48, May 1998.
19. T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd ed., 2013.
20. S. Luke and L. Panait, "A comparison of bloat control methods for genetic programming," *Evolutionary Computation*, vol. 14, pp. 309–344, September 2006.
21. T. Soule and J. A. Foster, "Effects of code growth and parsimony pressure on populations in geneticprogramming," *Evolutionary Computation*, vol. 6, pp. 293–309, December 1998.
22. B.-T. Zhang and H. Mhlenbein, "Balancing accuracy and parsimony in genetic programming.," *Evolutionary Computation*, vol. 3, no. 1, pp. 17–38, 1995.
23. R. Poli and N. F. McPhee, "Covariant parsimony pressure in genetic programming," 2008.
24. S. McPeak and D. S. Wilkerson, "The delta tool." `http://delta.tigris.org`.
25. "Javascript delta tool." `https://github.com/wala/jsdelta`.
26. A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002.
27. T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, pp. 308–320, Dec 1976.
28. D. Mitchell, R. J., *Managing complexity in software engineering*. No. 17 in IEE Computing series, P. Peregrinus Ltd. on behalf of the Institution of Electrical Engineers, 1990.
29. "The java language specification: Java se 8 edition."
30. ECMA International, *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 ed., June 2011.
31. `https://bugzilla.mozilla.org/show_bug.cgi?id=676763`.
32. "Gdb 'exploitable' plugin." `http://www.cert.org/vulnerability-analysis/tools/triage.cfm`.
33. `https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Strict_mode`.
34. `https://www.mozilla.org/en-US/security/advisories/mfsa2015-102/`.
35. B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, pp. 32–44, December 1990.
36. T. Clarke, "Fuzzing for software vulnerability discovery," 2009.
37. C. Miller, "How smart is intelligent fuzzing-or-how stupid is dumb fuzzing," August 2007.
38. R. Kaksonen, M. Laakso, and A. Takanen, "System security assessment through specification mutations and fault injection," in *Proceedings of the International Conference on Communications and Multimedia Security Issues*, May 2001.
39. X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation,*, pp. 283–294, June 2011.
40. Zalewski, "Announcing ref_fuzz a 2 year old fuzzer." `http://lcamtuf.blogspot.in/2010/06/announcing-reffuzz-2yo-fuzzer.html`.
41. Zalewski, "Announcing cross_fuzz a potential 0-day in circulation and more." `http://lcamtuf.blogspot.in/2011/01/announcing-crossfuzz-potential-0-day-in.html`.

42. J. Rudersman, “Introducing jsfunfuzz.” `http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz`.
43. A. Arya and C. Neckar, “Fuzzing for security.” `http://blog.chromium.org/2012/04/fuzzing-for-security.html`.
44. W. Afzal, R. Torkar, and R. Feldt, “A systematic review of search-based testing for non-functional system properties,” *Information and Software Technology*, vol. 51, no. 6, pp. 957 – 976, 2009.
45. P. McMinn, “Search-based software test data generation: A survey,” *Software Testing Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.
46. F. M. Kifetew, R. Tiella, and P. Tonella, “Combining stochastic grammars and genetic programming for coverage testing at the system level,” pp. 138–152, 2014.

Appendix

Comparing Code Generation Approaches

The aim of this experiment is to compare GP based code generative approach against code mutation, employed by [3] and pure generative approach. This experiment should clarify how these approaches accounts for good results. To measure the impact of these approaches, we need three independent runs of IFuzzer.

Genetic Programming Approach. First run is with a default configuration that follows the GP approach by performing genetic operations on the individuals, making a semantic adjustment, and using extracted code fragments for replacements.

Code Mutation Approach. In the Second run, IFuzzer is set to perform code mutation and to use parsed fragments in replacement process, which is similar to LangFuzz approach. This process is performed by disabling crossover and replacement functionality of the IFuzzer. Objective function has no role in this process, and we do not calculate the fitness of the individuals.

Generative Approach. The third run perform code generation using the generative approach, the configuration should produce a random code generation without using mutation or genetic operators. This falls back to pure generative approach and does not use extracted fragments for replacement. In this approach, we start with a *start terminal* in the language grammar and generate the code by randomly selecting the production rules for a non-terminal that appears in this process. This process will be terminated after reaching terminals and in case of recursive grammar rules sometimes it may result in an infinite loop.

Code Mutation and GP approaches can bring diversity among the generated code, thereby resulting in the higher chance to introduce errors. The generative approach, by definition, should have been easier to construct valid programs, but this leads to incomparable results, as there is no consistent environment.

In order to compare these approaches, we initially ran all three independent configurations on SpiderMonkey 1.8.5 for 2-3 days. All these processes are driven by randomization and therefore it is difficult to compare the results. The main intuition of our experiment was to observe the divergence of the code generation and the performance. It was observed that generative approach required more semantic knowledge without which it generated very large code fragments and its performance is based on the structure of grammar. We continued for multiple instances with the first and second configurations for five more days and observed that IFuzzer is fast enough to find bugs with the first configuration. Even with a greater overlap ratio, the number of bugs found was slightly higher with a GP approach when compared with the pure code mutation approach.

Figure 5 shows the results of comparison experiments between IFuzzer“s GP and code mutation approaches. By considering the fact that both runs are independent and results are very hard to compare as the entire process runs on randomization, it appears that GP directs the program to generate required output and improves the performance of the program.

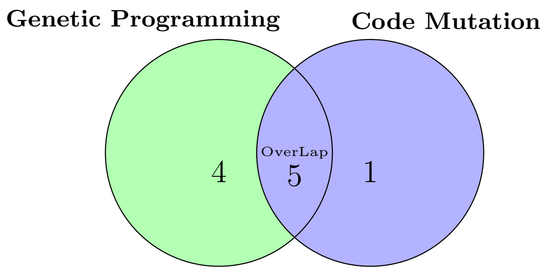

Fig. 5: Defects found with Code Mutation and Genetic Programming approaches

To measure the impact of code mutation and GP approach, we recorded the code evolution process. In code mutation and GP approaches code generation is performed with or without expanding. In either approach, extracted fragments are used for replacements. Both the approaches brought divergence, but without evolutionary computing divergence was achieved at a slower rate.

The IFuzzer’s GP based approach is a guided evolutionary approach with the help of fitness function, whereas LangFuzz follows a pure mutation-based approach by changing the input and testing. There is no evolutionary process involved in LangFuzz by using a fitness function. The inputs that triggered bugs are not entirely new inputs but have evolved through generations starting from the initial test cases. We repeated this experiment and observed such findings.