IFuzzer: An evolutionary fuzzer using genetic programming

Spandan Veggalam
IIIT Hyderabad,India
veggalam.s@reseach.iiit.ac.in

Sanjay Rawat
IIIT Hyderabad,India
sanjay.rawat@iiit.ac.in

ABSTRACT

Recently, there has been a surge in finding vulnerabilities in browsers. This is mainly attributed to its ability to host variety of services by executing corresponding applications. Although, most of the research has gone into addressing web-related vulnerabilities like, XSS, SQLi etc., we focus on other type of vulnerabilities namely low-level or implementation level bugs. We specifically target embedded interpreters as these software process code that may come from a malicious site. We present an automated evolutionary fuzzing technique to find bugs in JavaScript interpreters. Fuzzing is an automated black box testing used for finding security vulnerabilities in the software by providing random data as input. However, in the case of interpreter, fuzzing is challenging because inputs are pieces of code that should be *digestible* i.e. syntactically/semantically valid to pass the interpreter elementary checks. On the other hand, the fuzzed input should also be *indigestible* so as to trigger the uncommon behavior of the interpreter. In our approach, we use evolutionary computing techniques, spcifically *grammatical evolution* techniques to guide the fuzzer in generating indigestible input code fragments that may trigger exceptional behavior of interpreter such as crashes, memory leaks and failing assertions. We implement a prototype named IFuzzer to evaluate our technique on real-world examples. IFuzzer uses language grammar to generate digestible inputs. We applied IFuzzer on JavaScript interpreter which found 40 vulnerabilities in the time span of one week, out of which 5 are new bugs.

Categories and Subject Descriptors

[**Security and Privacy**]: Systems Security—*Vulnerability detection*

General Terms

SECURITY, VERIFICATION

Keywords

Security, Fuzzing, JavaScript, Grammatical Evolution, Evolutionary Computing, Artificial Intelligence, Interpreters, software vulnerability

XXXX '15 XXXX, XXXXXXXXXXXX, USA

1. INTRODUCTION

Software security is an essential dimension for a reliable software. Reliability is even more demanding issue when we consider interaction of software, specially with untrusted Internet. The Web has proved to interlink different information systems and has become main medium for sharing information. In the recent past, there have been numerous studies, techniques, and tools proposed mainly to address the security issues related to information processing. Web browsers are one of the application software that serve as an interface for different networks and systems. These browsers are becoming more and more sophisticated, because they render different services for which they include several interconnected components. Interpreters for various langauges like JavaScript, PHP, Flash, PDF, XSLT and many more are one of such component. Because of their widespread usage, they became primary applications for security attack, data and privacy breaches. The embedded interpreters can be exploited for launching browser based security attacks [4]. JavaScript embedded interpreter in modern browsers (e.g., SpiderMonkey in Firefox) is one such widely used interpreter responsible for several vulnerabilities [14]. In general, we look at bugs (and security flaws) in browsers [31], and in interpreters particularly, in the following perspective:

1. The way of handling information which includes extracting, parsing, storing, manipulating and communicating may also raise some vulnerabilities
   e.g., Cross Site Scripting
2. Flaws in browser software components like script engines, plug-ins, etc., can be used to exploit vulnerabilities
3. Browser components may also result in low-level vulnerabilities which can be exploited
   e.g., memory corruptions, assertion failures, uncommon behaviors etc

However, in the context of web related vulnerabilities particularly software's like browsers and their components, not much has been investigated in the direction of the above mentioned point 2. As a result, security bugs were getting accumulated and over a short period of time, there have been many bugs uncovered in browser-run software's: e.g., DOM Components, PDF interpreters, JavaScript Engines [15].

Fuzz testing is a useful approach for finding vulnerabilities in software. One of its variants, evolutionary fuzzing, turned out to be an useful smart fuzzing method. This variant make use of evolutionary computing approaches to automatically generate inputs that exhibit vulnerabilities. In the past, this form of fuzzing has been shown to be very effective in testing software [3, 8, 10, 13, 28, 34]. However, applying fuzzing to test interpreter is bit more challenging. Following are few issues that we have observed to motivate our present work in this direction:

1. Traditionally, fuzzing is about mutating the *data* that is manipulated by software (i.e., code). In the case of interpreter, fuzzing is about mutating *code*.

2. interpreter fuzzers must generate syntactically valid input, otherwise inputs will not pass the elementary interpreter checks. Therefore input must be generated making use of knowledge about target language. Assuming JavaScript interpreter to be target, fuzzed input must follow the syntax specifications of JavaScript language. Otherwise JavaScript interpreter discards the inputs during its first step i.e., parsing. Therefore JavaScript language grammar is used to generate syntactically valid code fragments.

3. interpreter may use a somewhat different (or evolved) version of grammer than the one known publicly. This makes it difficult to fuzz the interpreter to its full extent.

With built-in language grammar, valid data can be modeled. Both language dependent and independent fuzzers can be built with some additional knowledge of a specific language. *jsfunfuzz* [29] is popular fuzzer for Mozilla's JavaScript engine. This is an example for language dependent fuzzer. Another fuzzer *Langfuzz* [15] is an example of language independent fuzzer that generates syntactically valid code fragments. In this work we focus on answering the following question - can we build language independent fuzz like langfuzz which is more established well-formed approach, using evolutionary computing techniques. In this paper we introduce a new framework called *IFuzzer*, that generates code fragments using Grammatical Evolution method - a particular form of Genetic Programming [19], thereby allowing us to test interpreters by following black-box fuzzing technique. IFuzzer takes language's context free grammar as input for tests generation. Uses grammar to learn code fragments from the input code base. For isnstance IFuzzer takes javascript grammar input and output code fragments Given a test suite, IFuzzer performs standard genetic algorithm operations - crossover and mutations (discussed in subsection 3.3) on the input code fragments and uses learnt code fragments for replacements.
Grammatical Evolution brings transparency on making decision, inspired by biological evolution. It follows Darwin's theory of evolution and selects programs with high fitness. High fitness is a value computed by an objective function which favors the program that are uncommon enough. Grammatical Evolution approach is appropriate at generating code, and likely to produce diverse code fragments. This approach

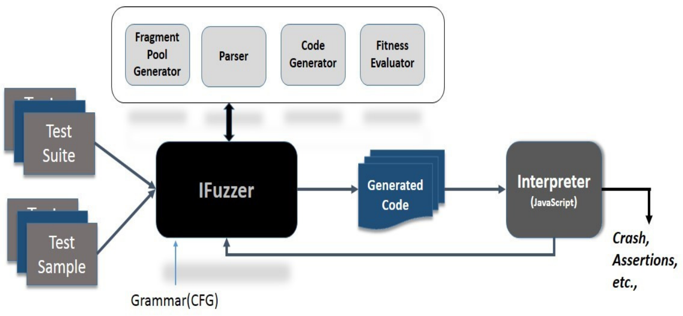

Figure 1: Overview of IFuzzer Approach

gives more flexibility from its rich modularity provided in form of grammar. It is possible to use different search strategies, such as evolutionary, heuristic, stochastic or deterministic. As the structures are described in terms of grammar, changes to strucutres can be easily by using grammar rules.

Figure 1 describes the overview of IFuzzer which takes test suite, language grammar and sample code as input. Parser uses language grammar to parse the program and generates XML object as abstract syntax representation. Fragment pool extractor generates a pool of code fragments is formed from the input sample codes, by extracting code fragments for different non-terminals in grammar. Input test suite itself can also be used as sample codes. Code generator generates new code fragments by performing genetic operations on the test suite. All the genereted code fragments are passed to interpreter. Based on the feedback from the interpreter, all the fragements are evaluated by the fitness evaluator.

Contributions. point out the main contributions in this sections. For example:

1. A complete automated technique for code fragments generation

2. A guided approach defined by the objective function.

3. Development of a tool IFuzzer for testing real-world interpreters

4. Empirical proof by showing experimental results

Remaining paper explains the code generation process and fuzzing process. section 2 presents the motivation for choosing Grammatical Evolution for code generation. In section 3 we describe code generation process. Implementation of IFuzzer is discussed in section 4. section 5 discusses the experimentation setup and evaluation step of IFuzzer and section 7 concludes the work with comments on possible future work.

2. BACKGROUND

Terminology

In the following section we introduce terminology that used in our work. Majority of them stem from the discipline of grammatical evolution [24].

1. **Interpreter.** In this paper the term *"interpreter"* is software system that executes input program. This translates code from one form to other form during execution.

2. **Genotype.** In this paper the term *"Genotype"* refers to an object that carries information of the individuals, taking part in evolution process.

3. **Phenotype.** In this paper the term *"Phenotype"* refers to the structure of the program that is directly executed and on which fitness is calculated based on the behavior of this individual.

4. **Grammar.** The grammar defines the context free grammar of the target language. Grammar is defined in backus normal form or extended backus normal form.

5. **Genome.** In grammatical evolution genome is the linear representation of an individual. All the genetic operators are applied on this linear representation and plays vital role in evolution process.

6. **Codon.** *Co-don* is building block of Genome. Each codon is of equal size.

7. **Search Space.** Search space is the set of all feasible solutions. Each point is the represents a solution defined by fitness values or some other values related to an individual.

8. **Bug/Defect.** In this paper the term *bug* or *defect* refers to an abnormal behavior of the target software system (e.g. Crash due to memory overflows, assertion failures, etc.)

3. IFUZZER APPROACH

In this section we discuss how to use Grammatical Evolution(GE) Techniques [30], [23], [24], [27] for program generation. In general testing inputs are generated using Generative approaches or Mutation approaches or both. *IFuzzer* makes use of both mutation and generative approaches. IFuzzer follows GE approach in code generation. GE is a Genetic Programming(GP) paradigm and analogous to biological evolutionary process. GE applies genetic operators crossover, mutation and replacements on the individuals in program generation process. In this process of evolution all Genotypes are mapped to Phenotypes [12].

3.1 Fragment Pool

Fragment pool is a collection of code fragments for each non-terminal in grammar specification. It is constucted by processing and extracting fragments from all files in the input test-suite. Each code fragment can be represented by a non-terminal. Using the parser, input files are parsed and code fragments are extracted for different non-terminals. With sufficient number of input files, code fragments can be generated to all non-terminals in the language grammar. These code fragments are used in mutation and code generation phase, same logic is followed in crossover for identifying code fragments for selected common non-Terminal between the participating individuals.

In mutation code fragments in the input is replaced with corresponding non-terminal code fragment selected from the fragment pool. In crossover code fragments of same type from two different programs are exchanged with each other. All these operations discussed in subsection 3.3, may result in semantically invalid fragments or loose context on programs. For this we perform semantic improvement and new code fragments are generated in context of participating individuals.

3.2 Initial Population

A set of initial population of individuals is generated by randomly selecting certain number of programs equal to population size from the input test samples. This forms first generation of evolution. After each generation, individuals forms the parental set of individuals (genotypes), which undergo genetic operations and there by evolves into off-springs.

Genetic algoritms uses many parameters such as, crossover, mutation rate and population size. Their values are set based on hueristics. Small changes to the values may adversely affect the evolution process and deviates it from the objective. There are few other parameters that control the process.

3.3 Genetic Operators

In this section we will look into how the genetic operators are implemented in IFuzzer. Figure 3.3 illustrates the crossover and mutation operations over tree structures and binary strings. After each operation, the off-spring evolved is evaluated for the fitness by the objective function defined. Fitness evaluation is further discussed in subsection 4.3.

3.3.1 Crossover

In Grammatical Evolution Crossover [25] is analogous to biological operation, performed between two individuals in population and is considered to be an exploratory operator. Types of crossover operations are differentiated based on number of crossover points. In our approach we perform homologous two point crossover on two individuals by swapping code fragments that belongs to same non-terminal. Following steps are followed:

1. Randomly two individuals are picked from the population.

2. About 70% of time, individuals are selected from top half of the population sorted based on fitness values. In remaining 30% of time entire population is considered for selection.

3. Parse the two individuals that undergo crossover by the language parser.

4. Identify list of common non-terminals between the individuals.

5. A non-terminal is randomly selected from the list of common non-terminals.

6. Code fragment of common non-terminal is randomly selected from both the individuals is exchanged with

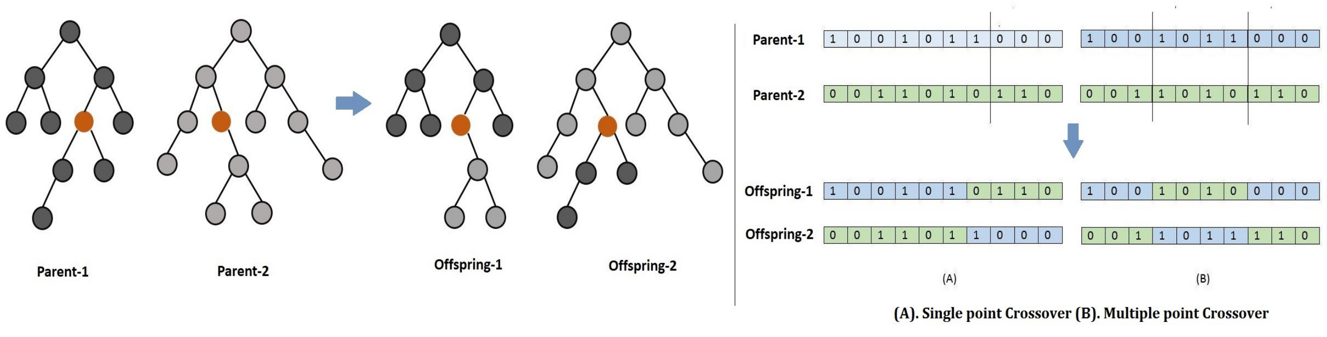
Parent-1
Parent-2
Offspring-1
Offspring-2
Parent-1 1 0 0 1 0 1 1 0 0 0 | 1 0 0 1 0 1 1 0 0 0
Parent-2 0 0 1 1 0 1 0 1 1 0 | 0 0 1 1 0 1 0 1 1 0
Offspring-1 1 0 0 1 0 1 0 1 1 0 | 1 0 0 1 0 1 0 0 0 0
Offspring-2 0 0 1 1 0 1 1 0 0 0 | 0 0 1 1 0 1 1 1 1 0
(A)
(B)
(A). Single point Crossover (B). Multiple point Crossover

Figure 2: Crossover illustration on tree and binary representations at boundary of similarity

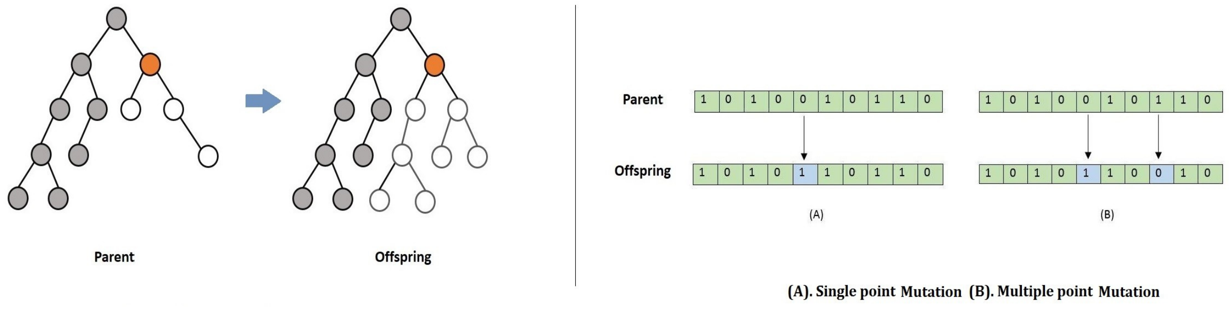
Parent
Offspring
Parent 1 0 1 0 0 1 0 1 1 0 | 1 0 1 0 0 1 0 1 1 0
Offspring 1 0 1 0 1 1 0 1 1 0 | 1 0 1 0 1 1 0 0 1 0
(A)
(B)
(A). Single point Mutation (B). Multiple point Mutation

Figure 3: Mutation illustration on tree and binary representations at boundary of similarity

each other. More than one code fragment in an individual may be an example of selected common non-terminal.

7. Typically, at most 3 non-terminals are selected for crossover.

3.3.2 *Mutation*

Mutation considered as divergence operator [6], brings diversity among the population and improves the search space. It brings structural changes to individuals and has benificial impact on search space traversals. Here, Mutation is performed by replacing code fragment with different code fragments of similar type. For this we pick some of code fragments randomly from the fragment pool and replace them with the code fragments of same type. Replacement is performing by selecting code fragment from the pool or before replacing them directly, we perform some improvements to the structure as discussed in subsection 3.5. Selected code fragment is replaced with one formed by expanding the non-terminal using the corresponding production rules from language grammar. After expansion all the non-terminals are converged to terminals. At most 3 non-terminals are selected for mutation.

3.3.3 *Shrink Mutation and Hoist Mutation*

Apart from mutation we perform Shrink mutation in which we replace a randomly chosen code fragment with a terminal or the code fragment is removed completely. As with hoist mutation code fragment is selected and is considered as complete program for next generation. In both shrink and hoist mutation, it is motivated by the desire to reduce program size.

3.3.4 *Replacement*

During the process of off-spring generation, it is important to retain the features of best chromosomes (parents) participating in evolution. Therefore Fitness elitism method is followed to retain the best chromosomes among the parents, these chromosomes remain untouched and are included as the off-springs. Remaining off-spring population is formed with the ones generated from crossover and mutation. This prevents the loosing the best ones in the process.

3.4 Bloat Control

Bloating [27] is a phenomenon that adversely effects in evolutionary computing. Bloating is distinguished into two types: functional and structural bloating.

In practical after some point of time the average size of programs start growing rapidly due to uncontrolled growth [33]. This results in inefficient code and growth doesn't relate to improvements in fitness. On the other hand, large programs require more computation to process. This is because code modification in a region may result in inviable code that perform no function or it may result in large code which do not change fitness at all. This is structural bloating.

In functional bloating [17] range of fitness values become narrow and thereby reduces the search space. Though it is common to have different individuals with same fitness but because of bloating after some time everything looks linear and it becomes hard to distinguish individuals.

Bloat control [18] can be applied at different stages and applying at the level of the fitness evaluation is very common technique. We use parsimony pressure [32], ad-hoc method for combating the bloat. With this method the selection probability of the individual by penalizing the larger ones i.e. fitness is re-calculated with some penalty depending on their size. Bloat control is also applied at breeding level uses shrink mutation and fair size crossover [27] techniques. We use following bloat control techniques.

Fitness re-calculation using Parsimony Co-efficient
Fitness value of the individual is re-calculated with some penalty using following equation:

$$f_{new}(x) = f(x) - penalty$$

$f_{new}(x)$ is new fitnes value, $f(x)$ is the actual fitness and *penalty* is the product of parsimony co-efficient and length of individual.

$$penalty = c * (l(x))$$

where *l(x)* is length of abstract syntax structure i.e. XML Object. Here, length of abstract structure is considered as length of individual. c is the parsimony co-efficient or parsimony penalty computed dynamically.
Following equation is used to calculate parsimony co-efficient *c(t)* at each generation t:

$$c(t) = \frac{Covariance(f, l)}{Variance(l)}$$

$$c(t) = \frac{\sum_{i=0}^{n}(f_i - \bar{f})(l_i - \bar{l})}{n-1} X \frac{n-1}{\sum_{i=0}^{n}(l_i - \bar{l})^2}$$

where $\bar{l}$ and $\bar{f}$ are mean fitness and length of all individuals in population respectively, f_i and l_i are original fitness and length of individual i.
$Covariance(f, l)$ calculates the co-variance between individual's fitness and length, $Variance(l)$ gives the variance in the length of the individuals respectively.

Fair Size Generation
Fair size generation limits the growth of the offspring's with bias value. In our approach we restrict the percentage of increase in program non-terminals to certain biased value. In other way we reject the individual. This is repeated for certain number of iterations before discarding.

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

where length gives information about of number of non-terminals in parse tree and bias is the threshold value for fair size generation. In this way we restrict the growth of program, if the program generated fails to meet this constraint program it is discarded as invalid and the program is re-generated

3.5 Code Evolution

With Crossover and Mutations, we generate diverse code fragments and order of genetic operations is purely random either of mutation or crossover can take place first.

Using language parser parses the input program and returns the abstract syntax tree represented by an XML structure. Tags in XML object are formed with language non-terminals and text embedded between tags is the code fragment identified from input program according to language syntax. All the non-terminals nested and ordered according to input program structure. Extracting the text in the order of XML structure re-generates input program itself.

During mutation method, several random code fragments part of input program are selected for mutation. Removing the selected non-terminal form an incomplete program and voidness of this code fragment is filled by code fragment that can be identified by non-terminal or non-terminal is expanded to complete program. We use following expansion algorithm:

1. Following steps are executed in loop up to a depth d
   (a) Set of non-Terminals in the incomplete program are identified.
   (b) Identify the set of productions $P_n \subset$ P under the non-Terminal n.
   (c) Modulo operation on the number of productions with decimal integer of codon, a production p is selected.
   $$p_{rule-selected} = codonmodnumber - of - rules$$
   (d) Replace the non-terminals occurrence with p.
   (e) Gene string is appended to itself in case of in sufficient codon values.
2. After expanding to a certain depth d, all the remaining non-terminal occurrences are to be replaced with terminals to yield complete program. For this we randomly select same type of code fragments from fragment pool.

   Codons are the chunks of 8-bits from the gene of the individual.

In Crossover, several random fragments (Typically 1..3) are selected from the both the individuals by selecting common non-terminals from the corresponding XML objects. These selected fragments from one individual are exchanged with fragments of other individual. All these operations are performed on XML objects, after which new programs are generated from these modified XML objects.

Genetic operators mutation or crossover or both performed on the population in each generation and thereby forms offspring population i.e. next generation. Code generation process is continuous process and is terminated after a certain number of generations or based on stopping criteria.

With primary target to trigger exceptional behavior of the Interpreters and Compilers, we start with existing test cases written for the target language by developers as a standard practice. All the processes discussed above are applied to the test cases one after one, from the learning phase to mutation phase, thereby creating executables from the original test cases.

3.6 Semantic Adjustment

In our Language Independent approach, we try to generate semantically valid programs by maintaining semantical context of the input programs. Introducing language semantics ties IFuzzer to a language specifications. Code fragments generated with Crossover, Mutation and Initial Population generation methods might be semantically invalid or out of semantic context of original inputs. We perform generic class of semantic transformations as part of small semantic adjustments at syntactic level.

Continuation of semantics is one such generic approach can be used to any programming language. Re-using literals occurred in body of input programs is an example of this approach. With this we can reduce undeclared identifier exceptions. In most of the languages all the memory locations and functions are named with identifiers and are declared somewhere in the body of the program. These identifiers must be declared before they are used, but in some languages like JavaScript they can be declared anywhere in the program and are evaluated during run time. Re-using the identifiers from the body of the original input within the new fragments reduce the chances to have undeclared identifiers. This can be done at the syntactic level. For this IFuzzer need to know the identifiers used in the input program, from the language grammar using identifier non-terminal we identify the identifiers from the program and replace identifiers in new fragments. It is still possible that identifiers are undeclared at the time of executing (because the reused identifier may be declared downward the new fragment) but the chance of undeclared identifier is reduced.

IFuzzer is aware of the language global objects and built in functions which can be used with out declaring them. List of these objects and functions are passed as an argument with which they are identified in the new fragments. These objects are left unmodified by IFuzzer, are usually defined in the implementation of language.

SemiColon Insertion

Semantic Adjustment discussed above with small additions can be a language independent adjustment. Another adjustment we make is Automatic Semicolon Insertion is language specific approach performed based on ECMAScript-262 specifications [11]. According to grammar some JavaScript statements must be terminated with semicolons and are therefore affected by automatic semicolon insertion.

4. IMPLEMENTATION

A fuzzer is implemented as proof-of-concept based on all the methods discussed so far and works as described in overview diagram Figure 1. IFuzzer starts with Fragment Pool generation process, where it takes each input file from the test suite and extracts code fragments for different non-terminals in language grammar. After extracting fragments from all input file, fuzzer starts it code generation and fuzzing process.

4.1 Parser

In most of the discussed methods input code is parsed and corresponding XML structure is generated. For this, IFuzzer uses ANTLR parser generator framework [26] to generated parsers for different languages using grammar specified in BNF. Why Antlr? Because of its easy-of-use over performance and different parse generation options which automatically build trees that can be folded into any application. Parser is used to extract code fragments from input test suite or to generate abstract syntax for which in either cases parser generates XML object as abstract syntax of program. Code fragments are extracted for each non-terminal from these XML objects. XML objects are involved in fragment pool generation and code generation methods.

4.2 Code Generation

IFuzzer uses ANTLR parser for target language which are generated using ANTLR language grammar. As discussed in subsection 3.5 Crossover, Mutation and Initial Population generation methods make use of XML objects returned by the parser for performing their operations. Small simplifications are made to grammar syntax internally without making any semantic changes and also changes are made to introduced Identifier as non-terminal (In most of ANTLR grammars identifier is included as terminal fragment). This makes step-wise expansion algorithm discussed in subsection 3.5 easier and newly introduced identifier non-terminal simplifies semantic adjustment and identifiers listing process.
Grammar simplification includes following modifications:

1. Rules containing sub-alternatives are written as separate rules.

2. All the rule quantifiers and optionals are removed by introducing additional rules.

Additional rule created are added to corresponding non-terminals. After these simplifications all the rules are left with non-terminals and terminals.

During Initial population creation itself, identifiers are extracted for particular individual and cached for further processing. In Code generation process, new fragments are fitted into existing semantic-al context as explained in subsection 3.6. For this purpose IFuzzer uses the cached identifiers for replacing the identifiers in new fragments. For suppose if an identifier "a" in new fragment is replaced with "b", all the occurrences of "a" are replaced with "b". Local Identifiers are mapped to Global and built-in identifiers but vice versa will not happen.
Code generation process is continued for a number of generations or until a individual with target fitness value is generated.

4.3 Fitness Evaluation

Evolutionary process is an objective driven process therefore fitness function that defines the objective of the process plays an vital role in code generation process. Individual's fitness is calculated at different stages. After crossover and mutation phases the generated code fragments are evaluated. Fitness value is calculated based on different factors. Various program parameters which includes structure of program, warnings raised by the interpreter, crashes, execution time outs, singularity are considered as fitness parameters. For instance nested (or complex) structure have tendency to create uncommon behaviour, so it gains more score compared to less complex structured programs. Each parameter

J_k in the fitness function is associated with a weight w_j determined based on the observations. For each individual I_i, we define the fitness as follows:

$$score_i = \sum_{j=0}^{k} w_j * f_{ij}$$

$$F_i = score_i + singularity_i$$

where f_{ij} is the occurrence frequency or characteristic value of parameter, singularity of the individual is calculated at the point of mutation based on the selection and expansion of non terminals. W is the set of weights for each parameter. Selection of values in W is of predominant important for fitness function and are decided based on heuristics.

As discussed in subsection 3.4 fitness is re-calculated with some penalty which is product of parsimony co-efficient c and the length of individual l. Fitness is re-calculated at each generation using following equation:

$$f_{new}(x) = f(x) - c * (l(x))$$

where $f_{new}(x)$ and $f(x)$ are new and old fitness values of an individual x.

4.4 Executing tests

IFuzzer executes the test inputs generated after each GA methods discussed. Valid test cases are considered for fitness evaluation. IFuzzer requires to generate more number of population generations thereby and after each iteration code generation moves towards the more desired solution. Here the objective is to identify uncommon behavior of interpreter with valid inputs. All the tests are performed in the persistent shell of the interpreter and uncommon behaviors are traced.

In case of failure due to invalid code fragment or if program doesn't meet Fair Size bias constraint discussed in subsection 3.4, Mutation and Crossover operations try to re-generate programs till it generates valid programs and gives up the re-generation process after certain number of trails.

IFuzzer contains many adjustable grammatical evolution parameters, e.g. mutation rate, crossover rates, population size, number of generations and many others. All the default values are derived from experiments. We tried to use the best combination based on observations, but these doesn't promise the certainty to deliver the best performance. It is not feasible to compare all parameter combinations because the evaluation of parameter set is time consuming. After a change in a paramter set it takes hours (or days) to evaluate, so repeating the process several times and observing the process to compare the results is not practical to do easily. In Table section 7 we listed the important parameters and their default values.

Delta Debugging algorithm [20] is used to filter out the irrelevant code from the test case resulting in small test cases, thereby part of test case that is relevant to failure [38] is filtered out. This algorithm reduces the number of lines of code executed and results in suitably small code fragment that causes failure.

5. EXPERIMENTATION

In our experiment, we used Mozilla development test suite as input which consists of 3000-3200 programs. Same test suite is used for fragment pool generation and program generation. Fragment Pool generation is one time process, all the programs are read at the start of fuzzer and fragments are extracted for different non-terminals. This test suite is assumed to have triggered exceptional behavior at some point time earlier in development cycles. Grammar rules are written from ECMAScript 262 specification [11] and for latest version Javascript engines we include more rules that are part of the Mozilla's ECMAScript 6,7 (Harmony) development [2].

5.1 LangFuzz vs IFuzzer

Langfuzz written by holler at el [15], is mutative fuzzer for testing interpreters. This is playing active part in Mozilla's and other companies QA activities and this product is integrated as part of mozilla development process.

There are significant differences between Langfuzz and IFuzzer. Langfuzz is a mutation fuzzer and as it is provided with grammar specifications makes it adaptable to test any interpreter. To be language independent it compromises the semantic level adaption during code generation process but in order to generate valid programs it makes certain semantic level adjustments. This gives the flexibility to remain as language independent tool and need not bother about the new features induced in the language. Langfuzz easily incorporates these features from the updated grammar specifications. This means that

1. Langfuzz is solely dependent on the language grammar, with some additional semantic knowledge.

2. Changes to language features doesnt require any additional product development, other than grammar updates.

In contrast, IFuzzer is a guided fuzzer, with an objective to trigger unexceptional behavior of the interpreters. IFuzzer remains as language independent tool and incorporates the evolution strategies in code generation process.

Both IFuzzer and LangFuzz are generative fuzzers that use grammar to be language independent but differ in code generation process. LangFuzz uses mutative approaches where as IFuzzer uses grammatical evolution technique for code generation. This makes fair comparison difficult.

5.2 Environment Setup

We choose *Spidermonkey* as a target interpreter for JavaScript. The main reason for choosing Spidermonkey is, Mozilla development process and related artifacts are publicly available. For all the tests, we used Mozilla's development repository and the test cases as first set of input. Input grammar specification is written from ECMAScipt standard specification. We decided to choose Spidermonkey 1.8.5 as comparison base. The main reason for choosing this version is to have comparison with LangFuzz. This version of Spidermonkey is in development by the time Holler presented his work on LangFuzz [15]. All the bugs found by LangFuzz during its initial development where listed in holler presentation with bug id's filed in Mozilla's bugzilla tool. In each instance we ran our application for 4-6 hours before making any changes to parameters which guides the fuzzer to reach goal. All the parameters of code generation are selected based on heuristics. When ever a bug is found we analyze it for cause and check whether it is identified by the LangFuzz during the period it targeted that specified version.

Listing 1: Test Case generated by IFuzzer, crashing the Spidermonkey JavaScript engine with an internal assertion when executing line 4.

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

5.3 Result of Comparison

During the experiment, IFuzzer found a total of 40 bugs in the time span of one week. In contrast, to the 51 bugs listed in LangFuzz presentation which are found during its initial run of 3 months. There are 11 overlapped bugs found by both LangFuzz and IFuzzer, this addresses the research question "How many overlapped bugs are found by both fuzzers?".

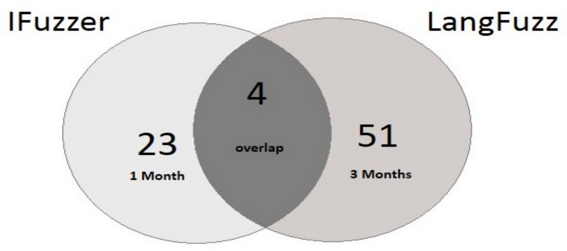

Figure 4: Number of bugs found by both fuzzers and their overlap

The number of bugs found by IFuzzer shows the effectiveness of the evolutionary approach and it is as good as other generative approaches. IFuzzer tool shows that evolutionary approach which is similar to that of biological evolution is capable of generating programs with a given objective.

Listing 2: Test Case generated by IFuzzer, crashing the Spidermonkey JavaScript engine. JavaScript engine fails to handle the situation, leading to memory leak

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

With curiosity, when we ran on latest version of Spidermonkey IFuzzer found 4 bugs. One of the bug is accepted as severe bug (Bug Id: 1133247) [1] where one of the component fails to handle the situation and JavaScript engine consumes available heap memory and crashes in few seconds. Code fragment responsible for crash is shown in **??**

6. RELATED WORK

Fuzz testing was transformed from small recess technique like Barton Miller's [21] project for testing unix system utilities to one that is widely adopted techniques.

Researchers started fuzzers as brute forcing tools [7] for discovering flaws, after which they would analyze for possibility of security exploitation. After many years it was realized that such simple approaches have many limitations in discovering complex flaws. Smart Fuzzer, aka Intelligent Fuzzer overcomes these limitations and are found to be more effective [22].

In 2001, Kaksonen [16] in his approach *mini-simulation*, used simplified description of protocols and syntax to automatically generate inputs that nearly matches with the protocol usage. This approach is generally called as grammar based approach, it provides the fuzzer with sufficient information to understand protocol specifications. Kaksonen's *mini-simulation* is used to ensure the protocol checksums are always valid and also systematically checks which rules are broken. In contrast, IFuzzer uses the grammar specification to generate valid inputs.

Yang et al [35] presented their work on CSmith, a random c program generator tool. It uses grammar for producing programs with different features, there by performing testing and analysing C compilers. This process uses semantic information during their generation process, which is reasonable for language independent fuzzer. For constructing language independent fuzzers like IFuzzer, so far it is very difficult.

Zero-day vulnerability forced the researchers to pay more attention on software security. Similar vulnerabilities of web browsers and its components are increasing rapidly. This made security testing on web browsers promising. In 2011, Zalewski presented ref_fuzz [36] and crossfuzz [37] tools aiming DOM component in browsers. He revealed interesting problem with the modern web browsers. Zelaski also published fuzzers for different browser components and found several bugs. JsFunFuzz [29] is a language dependent generative tool written by Ruddersman in 2007, targets JavaScript interpreters in web browsers and has lead to discovery of more than 1800 bugs in Spidermonkey, firefox's JavaScript engine. It is considered as one of most relevant work in the field of web interpreters. Langfuzz, a language independent tool presented by Holler at el [15] uses language grammar and code mutation approaches for test generation. In contrast, IFuzzer uses grammar specification and code generation techniques.

There are few proprietary fuzzers targeting different software components. Google's ClusterFuzz [5] one of such fuzzer tests different 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 these approaches may deviate the process of code generation from generating required test data, thereby degenerating into random search and providing low code coverage.

Feedback fuzzers adjusts and generates dynamic inputs based on information from the target system, *Evolutionary Fuzzer* is one example of this kind. Evolutionary fuzzing uses evolutionary algorithms to create required search space of data and operates based on a objective function that takes the control of test input generation. One of the first published approaches to evolutionary fuzzing [9] is from Michigan State University by DeMott et al. in 2007. This a Grey-box technique that measures code block coverage and generates new inputs generation-ally with better code coverage to find bugs. IFuzzer adapted the idea of evolutionary computing algorithms for code generations.

IFuzzer is a language independent black box fuzzer, which when given JavaScript grammar generates the Javascript programs and tests the target javascript engines. It used grammatical evolution, a evolutionary computing technique for test generation. With the feedback from the target and other code parameters, it evaluates each program and there by guides the generation process.

7. CONCLUSION AND FUTURE WORK

In an approach to fuzz testing, IFuzzer uses an evolutionary code generation strategies that can be applied to any language provided with appropriate grammar specification and set of test cases for code generation process. The fuzzer uses objective function defined to help the fuzzer in reaching the goal in an efficient way. This makes the IFuzzer an effective tool. In our evaluation, it is shown that IFuzzer is fast in reaching objective and also it is as effective as other generative fuzzers. We recommend our approach for automated code generation for testing of any target language that can be expressed in grammar specification, this includes the software products like interpreters and compilers.

We plan to investigate for more code parameters that can be considered for fitness evaluation. In our experiments we observed that genetic operations parameters must be tuned for further improvement in evolution process. This can be done on based of the observations and vary for different environments. Another improvement can be to keep track of more information on the program execution which helps to guide the fuzzer in better way. For example, finding the tainted paths which gives coverage of functions executed, lines executed, paths that lead to a bug will be useful in code generation. Finally we intend to improve the code generation part and thereby extend its usage to more applications.

8. REFERENCES

[1] `https://bugzilla.mozilla.org/show_bug.cgi?id=1133247`.

[2] *ECMAScript 2015 - ECMAScript Language Specification*. Draft edition, April 2015.

[3] 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*, RISE'05, pages 50–65, 2006.

[4] 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.

[5] A. Arya and C. Neckar. Fuzzing for security. `http://blog.chromium.org/2012/04/fuzzing-for-security.html`, April 2012.

[6] J. Byrne, M. O'Neill, J. McDermott, and A. Brabazon. An Analysis of the Behaviour of Mutation in Grammatical Evolution. In *Proceedings of Genetic Programming, 13th European Conference, EuroGP*, pages 14–25, April 2010.

[7] T. Clarke. Fuzzing for software vulnerability discovery. 2009.

[8] C. Del Grosso, G. Antoniol, E. Merlo, and P. Galinier. Detecting buffer overflow via automatic test input data generation. *Comput. Oper. Res.*, 35(10):3125–3143, Oct. 2008.

[9] DeMott, Jared, Enbody, Richard, Punch, and W. F. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing.

[10] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz. Kameleonfuzz: Evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 37–48, New York, NY, USA, 2014. ACM.

[11] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. Ecma International, 5.1 edition, June 2011.

[12] D. Fagan, M. O'Neill, E. G. López, A. Brabazon, and S. McGarraghy. An analysis of genotype-phenotype maps in grammatical evolution. In *Proceedings of Genetic Programming, 13th European Conference, EuroGP 2010*, pages 62–73, 2010.

[13] 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*, volume 2, pages 491–497, Nov 2008.

[14] 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, pages 85–94, Washington, DC, USA, 2005.

[15] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21th USENIX Security Symposium*, pages 445–458, August 2012.

[16] 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.

[17] W. B. Langdon and R. Poli. Fitness causes bloat: Mutation. In *Proceedings ofGenetic Programming, First European Workshop, EuroGP'98*, pages 37–48, 1998.

[18] S. Luke and L. Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, September 2006.

[19] 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*, 11(3-4):365–396, May 2010.

[20] S. McPeak and D. S. Wilkerson. The delta tool. `http://delta.tigris.org`.

[21] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[22] C. Miller. How smart is intelligent fuzzing-or-how stupid is dumb fuzzing. August 2007.

[23] M. O'Neill and C. Ryan. Grammar based function definition in grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference GECCO*, pages 485–490, July 2000.

[24] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Trans. Evolutionary Computation*, 5(4):349–358, August 2001.

[25] M. O'Neill and C. Ryan. Crossover in grammatical evolution. *Genetic Programming and Evolvable Machines*, 4(1):67–93, March 2003.

[26] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[27] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. lulu.com, 2008.

[28] 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, pages 37–45, Washington, DC, USA, 2010. IEEE Computer Society.

[29] J. Rudersman. Introducing jsfunfuzz. `http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz`, 2007.

[30] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Genetic Programming, First European Workshop, EuroGP*, pages 83–96, April 1998.

[31] R. Sekar. An efficient blackbox technique for defeating web application attacks. In *Proceedings of the Network and Distributed System Security Symposium*, February 2009.

[32] T. Soule and J. A. Foster. Effects of code growth and parsimony pressure on populations in geneticprogramming. *Evolutionary Computation*, 6(4):293–309, December 1998.

[33] T. Soule, J. A. Foster, and Dickinson. Code growth in genetic programming. In *Genetic Programming 1996:Proceedings of the First Annual Conference*, pages 215–223, May 1996.

[34] 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*, pages 477–486. IEEE, 2007.

[35] 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,*, pages 283–294, June 2011.

[36] Zalewski. Announcing ref_fuzz a 2 year old fuzzer. `http://lcamtuf.blogspot.in/2010/06/announcing-reffuzz-2yo-fuzzer.html`, 2010.

[37] 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`, 2011.

[38] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

APPENDIX

Parameters

Parameter	**Value**
Population Size	300
Maximum number of generations	100
Fitness Selections	Elitism
Crossover rate	0.85
Mutation rate	0.2
Generative mutation rate (Mutation with expansion)	0.3
Mutliple point mutation and crossover rate	0.2
Maximum number of fragment replacements allowed in mutation	3
Children per crossover	2
Crossover bias threshold (length)	1.15
Maximum expansion depth	2
Fitness type	max
Replacement selections	Tournament

Table 1: IFuzzer common parameters and their default values.