2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Superion: Grammar-Aware Greybox Fuzzing

Junjie Wang*, Bihuan Chen†, Lei Wei*, Yang Liu*‡
*School of Computer Science and Engineering, Nanyang Technological University, Singapore
†School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China
‡College of Information Science, Zhejiang Sci-Tech University, China

***Abstract*—In recent years, coverage-based greybox fuzzing has proven itself to be one of the most effective techniques for finding security bugs in practice. Particularly, American Fuzzy Lop (AFL for short) is deemed to be a great success in fuzzing relatively simple test inputs. Unfortunately, when it meets structured test inputs such as XML and JavaScript, those grammar-blind trimming and mutation strategies in AFL hinder the effectiveness and efficiency.**

To this end, we propose a grammar-aware coverage-based greybox fuzzing approach to fuzz programs that process structured inputs. Given the grammar (which is often publicly available) of test inputs, we introduce a grammar-aware trimming strategy to trim test inputs at the tree level using the abstract syntax trees (ASTs) of parsed test inputs. Further, we introduce two grammar-aware mutation strategies (i.e., enhanced dictionary-based mutation and tree-based mutation). Specifically, tree-based mutation works via replacing subtrees using the ASTs of parsed test inputs. Equipped with grammar-awareness, our approach can carry the fuzzing exploration into width and depth.

We implemented our approach as an extension to AFL, named Superion; and evaluated the effectiveness of Superion using large-scale programs (i.e., an XML engine libplist and three JavaScript engines WebKit, Jerryscript and ChakraCore). Our results have demonstrated that Superion can improve the code coverage (i.e., 16.7% and 8.8% in line and function coverage) and bug-finding capability (i.e., 34 new bugs, among which we discovered 22 new vulnerabilities with 19 CVEs assigned and 3.2K USD bug bounty rewards received) over AFL and jsfunfuzz.

***Index Terms*—Greybox Fuzzing, Structured Inputs, ASTs**

I. Introduction

Fuzzing or fuzz testing is an automated software testing technique to feed a large amount of invalid or unexpected test inputs to a target program in the hope of triggering unintended program behaviors, e.g., assertion failures, crashes, or hangs. Since its introduction in the early 1990s [46], fuzzing has become one of the most effective techniques to find vulnerabilities in real-world programs for ensuring software security [45]. It has been applied to testing various applications, ranging from rendering engines and image processors to compilers and interpreters.

A fuzzer can be classified as generation-based (e.g., [32, 62, 64, 69]) or mutation-based (e.g., [9, 43, 55, 59]), depending on whether test inputs are generated by the knowledge of the input format or grammar or by modifying well-formed test inputs. A fuzzer can also be classified as whitebox (e.g., [25, 53]), greybox (e.g., [9, 43]) or blackbox (e.g., [46, 66]), depending on the degree of leveraging a target program's internal structure, which reflects the tradeoffs between effectiveness and efficiency. In this paper, we focus on mutation-based greybox fuzzing.

Coverage-Based Greybox Fuzzing. One of the most successful mutation-based greybox fuzzing techniques is coverage-based greybox fuzzing, which uses the coverage information of each executed test input to determine the test inputs that should be retained for further incremental fuzzing. AFL [71] is a state-of-the-art coverage-based greybox fuzzer, which has discovered thousands of high-profile vulnerabilities. Thus, without the loss of generality, we consider AFL as the typical implementation of coverage-based greybox fuzzing.

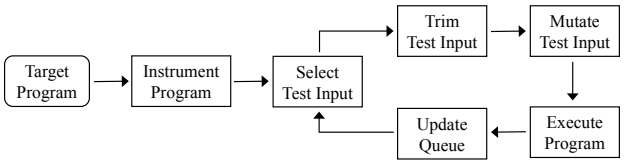


Fig. 1: The General Workflow of AFL

As shown in Fig. 1, AFL takes the target program as an input, and works in two steps: instrumenting the target program and fuzzing the instrumented program. The instrumentation step injects code at branch points to capture branch (edge) coverage together with branch hit counts (which are bucketized to small powers of two). A test input is said to have new coverage if it either hits a new branch, or achieves a new hit count for an already-exercised branch. The fuzzing step can be broken down into five sub-steps. Specifically, a test input is first selected from a queue where the initial test inputs as well as the test inputs that have new coverage are stored. Then the test input is trimmed to the smallest size that does not change the measured behavior of the program, as the size of test inputs has a dramatic impact on the fuzzing efficiency. The trimmed test input is then mutated to generate new test inputs and the program is executed with respect to each mutated test input. Finally, the queue is updated by adding those mutated test inputs to the queue if they achieve new coverage, while the mutated test inputs that achieve no new coverage are discarded. This fuzzing loop continues by selecting a new test input from the queue.

Challenges. The current coverage-based greybox fuzzers can effectively fuzz programs that process compact and unstructured inputs (e.g., images). However, some challenges arise when they are used to target programs that process structured inputs (e.g., XML and JavaScript) that often follow specific grammars. Such programs often process the inputs in stages, i.e., syntax parsing, semantic checking, and application execution [64].

On one hand, the trimming strategies (e.g., removal of chunks of data) in AFL are grammar-blind, and hence can easily violate the grammar or destroy the input structure. As a result, most test inputs in the queue cannot be effectively trimmed to keep them syntax-valid. This is especially the case when the target program

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00081

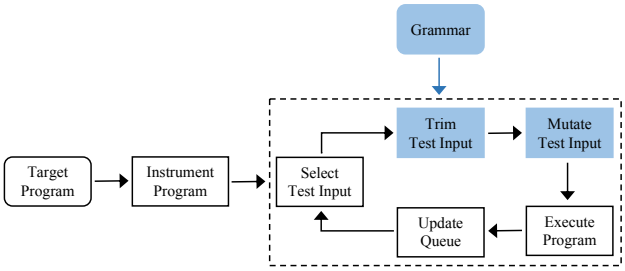


Fig. 2: The General Workflow of Superion with the Highlighted Differences from AFL (see Fig. 1)

can process a part of a test input (triggering coverage) but errors out on the remaining part. This will greatly affect the efficiency of AFL because it needs to spend more time on fuzzing the test inputs whose structures are destroyed, but only finds parsing errors and gets stuck at the syntax parsing stage, which heavily limits the capability of fuzzers in finding deep bugs.

On the other hand, the mutation strategies (e.g., bit flipping) in AFL are grammar-blind, and hence most of the mutated test inputs fail to pass syntax parsing and are rejected at an early stage of processing. As a result, it is difficult for AFL to achieve grammar-aware mutations. Besides, AFL spends a large amount of time struggling with syntax correctness, while only finding parsing errors. Thus, the effectiveness of AFL to find deep bugs is heavily limited for programs that process structured inputs.

The Proposed Approach. To address the challenges, we propose a new grammar-aware coverage-based greybox fuzzing approach for programs that process structured inputs. We also implement the proposed approach as an extension to AFL, named Superion[1]. Our approach takes as inputs a *target program* and a *grammar* of the test inputs that is often publicly available. Based on the grammar, we parse each test input into an abstract syntax tree (AST). Using ASTs, we introduce a grammar-aware trimming strategy that can effectively trim test inputs while keeping the input structure valid. This is realized by iteratively removing each subtree in the AST of a test input and observing coverage differences. Moreover, we propose two grammar-aware mutation strategies that can quickly carry the fuzzing exploration beyond syntax parsing. We first enhance AFL's dictionary-based mutation strategy by inserting/overwriting tokens in a grammar-aware manner, and then propose a tree-based mutation strategy that replaces one subtree in the AST of a test input with the subtree from itself or another test input in the queue.

To evaluate the effectiveness of Superion, we conducted experiments on one XML engine libplist and three JavaScript engines WebKit, Jerryscript and ChakraCore. We compared our approach with AFL with respect to the code coverage and bug-finding capability. The results have demonstrated that Superion can effectively improve the code coverage over AFL by 16.7% in line coverage and 8.8% in function coverage; and Superion can significantly improve the bug-finding capability over AFL by finding 34 new bugs (among which six were found by AFL). Among these bugs, 22 new vulnerabilities were discovered with 19 CVEs assigned; and we received 3.2K USD bug bounty

[1]Superion is an Autobot combiner in the cartoon *The Transformers*.

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
        <key>Some ASCII string</key>
        <string></string>
        <data>
        </data>
</dict>
</plist>
```

Fig. 3: An Example of AFL's Built-In Trimming

rewards. Besides, we compared Superion with jsfunfuzz [57], which is a successful fuzzer specifically designed for JavaScript. However, it failed to find any new bugs. Moreover, we have demonstrated that our grammar-aware trimming strategy can effectively trim test inputs while keeping them syntax-valid; and our grammar-aware mutation strategies can effectively generate new test inputs that can trigger new coverage.

Contributions. The contributions of this work are:

- We proposed a novel grammar-aware coverage-based greybox fuzzing approach for programs that process structured inputs, which complements existing coverage-based greybox fuzzers.
- We implemented our approach and made it open-source[2], and conducted experiments to demonstrate its effectiveness.
- We found 34 new bugs, among which we found 22 new vulnerabilities with 19 CVEs assigned and received 3.2K USD bug bounty rewards.

II. Our Approach

To address the challenges of coverage-based greybox fuzzing (Section I), we propose a novel grammar-aware coverage-based greybox fuzzing approach, which targets programs that process structured inputs. We implement the approach as an extension to AFL [71], named Superion. Fig. 2 introduces the workflow of Superion, and highlights the differences from AFL (see Fig. 1). In particular, a context-free grammar of the test inputs is needed, which is often publicly available (e.g., in ANTLR's community [1]). We introduce a grammar-aware trimming strategy (Section II-A) and two grammar-aware mutation strategies (Section II-B) with the purpose of making AFL grammar-aware.

A. Grammar-Aware Trimming Strategy

The built-in trimming strategy in AFL is grammar-blind, and treats a test input as chunks of data. Basically, it first divides the test input to be trimmed into chunks of len/n bytes where len is the length of the test inputs in bytes, and then tries to remove each chunk sequentially. If the coverage remains the same after the removal of a chunk, this chunk is trimmed. Note that n starts at 16 and increments by a power of two up to 1024. This strategy is very effective for unstructured inputs. However, it cannot effectively prune structured inputs while keeping them syntax-valid, possibly making AFL stuck in the fuzzing exploration of syntax parsing without finding deep bugs.

Example. Fig. 3 gives an example of AFL's built-in trimming on an XML test input with respect to libplist (an XML engine), where "l versio" and "dict> </plis" are trimmed (highlighted by strikethrough). The trimmed test input is syntax-invalid, but still has the same coverage as the original test input because the

[2]https://github.com/zhunki/Superion

Algorithm 1 Grammar-Aware Trimming

Input: the test input to be trimmed in, the grammar G
Output: the trimmed test input ret

```
1  while true do
2      parse in according to G into an AST tree
3      if there are any parsing errors then
4          return built-in-trimming (in)
5      end if
6      for each subtree n in tree do
7          ret = remove n from tree
8          run the target program against ret
9          if coverage remains the same then
10             in = ret
11             break
12         else
13             add n back to tree
14         end if
15         if n is the last subtree in tree then
16             return ret
17         end if
18     end for
19 end while
```

```
...
try{eval("M:if(([15,16,17,18].some(this.unwatch(\"x\"),(([window if([[]])])[this.
      prototype]))))) else{true;return null;}");} catch(ex){}
try{eval("M:while((null >=\"\")&&0){/a/gi}");} catch(ex){}
try{eval("\nbreak M;\n");} catch(ex){}
try{eval("L:if((window[(1.2e3.x::y)]).x) return null; else if((uneval(window))++.
      propertyIsEnumerable(\"x\")){CollectGarbage()}");} catch(ex){}
try{eval("/*for..in*/for(var x in ((({}).hasOwnProperty
    ((,,).hasOwnProperty(\"x\"))))/*for..in*/ M:for(var
    {window, y} =(-1) in this) {1,2,3,4].slice");} catch
    (ex){}
try{eval("if(\"\"){}else if(x4) {null;}");} catch(ex){}
try{eval("{}");} catch(ex){}
try{eval("for(var x = x in x - /x/ ){}");} catch(ex){}
try{eval("if((uneval(x, x))) var x = false; else if((null\n.unwatch(\"x\"))) throw
      window; else {} return 3;");}catch(ex){}
...
```

Fig. 4: An Example of Grammar-Aware Trimming

implementation of libplist does not adhere to XML's grammar specification. Hence, the trimmed test input is used for further fuzzing even though its grammar is destroyed.

To ensure the syntax-validity of trimmed test inputs, we propose a grammar-aware trimming strategy, whose procedure is given in Algorithm 1. It first parses the test input to be trimmed in according to the grammar G into an AST $tree$ (Line 2). If any parsing errors occur (as in's structure may be destroyed by mutations), then it uses AFL's built-in trimming strategy rather than directly discarding it (Line 3–5); otherwise, it attempts to trim a subtree n from $tree$ (Line 6–7). If the coverage is different after n is trimmed, then n cannot be trimmed (Line 12–14), and it tries to trim next subtree; otherwise, n is trimmed, and it re-parses the remaining test input (Line 9–11), and then repeats the procedure until no subtree can be trimmed (Line 15–16). Thus, we resort to AFL's built-in trimming only when our tree-based trimming is not applicable. This is because sometimes invalidity is also useful.

Example. Fig. 4 shows an example of our trimming strategy on a JavaScript test input, where a complete `try-catch` statement (highlighted by strikethrough) is trimmed without raising any coverage difference. It is almost impossible for AFL's built-in trimming strategy to prune such a complete statement.

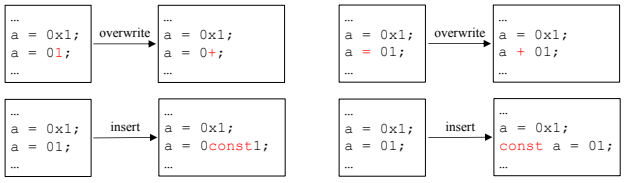
B. Grammar-Aware Mutation Strategies

The default mutation strategies (e.g., bit flipping or token insertion) in AFL are too fine-grained and grammar-blind to keep the input structure following the underlying grammar. Therefore,

Algorithm 2 Dictionary-Based Mutation

Input: the test input in, the dictionary D
Output: the set of mutated test inputs T

```
1  T = ∅
2  l = the length of in
3  for i = 0; i < l; do
4      j = i + 1
5      curr = *(u8*)(in's address + i)  // current byte of in
6      next = *(u8*)(in's address + j)  // next byte of in
7      while j < l && curr and next are alphabet or digit do
8          j = j + 1
9          next = *(u8*)(in's address + j)
10     end while
11     for each token d in D do
12         insert d at i of in / overwrite i to j of in with d
13         T = T ∪ {in}
14     end for
15     i = j
16 end for
```



(a) Original (b) Enhanced

Fig. 5: An Example of Dictionary-Based Mutation

we propose two grammar-aware mutation strategies to improve the mutation effectiveness on triggering new program behaviors.

1) Enhanced Dictionary-Based Mutation: Dictionary-based mutation [70] was introduced to make up for the grammar-blind nature of AFL. The dictionary is actually a list of basic syntax tokens (e.g., reserved keywords) which can be provided by users or automatically identified by AFL. Every token is inserted between every two bytes of the test input to be mutated, or written over every byte sequence of the same length of the token. Such mutations can produce syntax-valid test inputs but are inefficient as most of the generated inputs have destroyed structures.

Therefore, we propose the enhanced dictionary-based mutation as shown in Algorithm 2. This algorithm leverages the key fact that the tokens (e.g., variable names, function names, or reserved keywords) in a structured test input normally only consist of alphabets or digits. Hence, it first locates the token boundaries in a test input by iteratively checking whether the current and next byte are both alphabet or digit (Line 3–10). Then it inserts each token in the dictionary to each located boundary, which avoids the insertion between the consecutive sequence of alphabets and digits and thus greatly decreases the number of token insertions (Line 11–14). Similarly, it writes each token in the dictionary over the content between every two located boundaries, which also greatly decreases the number of token overwrites. Such token insertions and overwrites not only maintains the structure of mutated test inputs but also decreases the number of mutated test inputs, hence greatly improving the effectiveness and efficiency of dictionary-based mutation.

Example. Fig. 5 illustrates the difference between the original and enhanced dictionary-based mutation. In the original one, `01` is not treated as a whole, and thus `1` can be overwritten by `+` and `const` can be inserted between `0` and `1`, which destroys the structure without introducing any new coverage. In the en-

Algorithm 3 Tree-Based Mutation

Input: the test input tar, the grammar G, the test input pro
Output: the set of mutated test inputs T

```
1:  T = ∅
2:  S = ∅  // the set of subtrees in tar and pro
3:  parse tar according to G into an AST tar_tree  // Heuristic 1
4:  if there are any parsing errors then
5:      return
6:  end if
7:  for each subtree n in tar_tree do  // Heuristic 3
8:      S = S ∪ {n}
9:  end for
10: parse pro according to G into an AST pro_tree  // Heuristic 1
11: if there is no parsing error then
12:     for each subtree n in pro_tree do  // Heuristic 3
13:         S = S ∪ {n}
14:     end for
15: end if
16: for each subtree n in tar_tree do  // Heuristic 2
17:     for each subtree s in S do
18:         ret = replace n in tar_tree's copy with s
19:         T = T ∪ {ret}
20:     end for
21: end for
22: return T
```

hanced one, `01` is identified as a whole, and hence the mutated test inputs in Fig. 5a will not be produced. Instead, it can generate the mutated test inputs in Fig. 5b more efficiently, which are taken from our experiments and both lead to new coverage.

2) Tree-Based Mutation: Dictionary-based mutation is aware of the underlying grammar in an implicit way. To be explicitly aware of the grammar and thus producing syntax-valid test inputs, we utilize the grammar knowledge and design a tree-based mutation, which works at the level of ASTs. Different from the tokens used in dictionary-based mutation, AST actually models a test input as objects with named properties and is designed to represent all the information about a test input. Thus, ASTs provide a suitable granularity for a fuzzer to mutate test inputs.

Algorithm 3 shows the procedure of our tree-based mutation. It takes as inputs a test input tar to be mutated, the grammar G, and a test input pro that is randomly chosen from the queue. It first parses tar according to G into an AST tar_tree; and if any parsing errors occur, tar is a syntax-invalid test input and we do not apply tree-based mutation to tar (Line 3–6). If no error occurs, it traverses tar_tree, and stores each subtree in a set S (Line 7–9). Then it parses pro into an AST pro_tree, and stores each subtree of pro_tree in S if there is no parsing error (Line 10–15). Here S serves as the content provider of mutation. Then, for each subtree n in tar_tree, it replaces n with each of the subtree s in S to generate a new mutated test input (Line 16–21). Finally, it returns the set of mutated test inputs. Notice that we do not consider the node type when replacing subtrees because that will harm the general applicability of Superion.

The size of this returned set can be the multiplication of the number of subtrees in tar_tree and the number of subtrees in tar_tree and pro_tree, which could be very large. As an example, our tree-based mutation on tar and pro whose number of subtrees is respectively 100 and 500 will generate $100 \times (100 + 500) = 60,000$ test inputs. This will add burden to the program execution step during fuzzing, making fuzzing less efficient. To relieve the burden, we design three heuristics to reduce the number of mutated test inputs. For clarity, we do not elaborate these heuristics in Algorithm 3, but only show where they are applied.

TABLE I: Target Languages and Their Structure and Samples

Language	# Symbols	Structure Level	# Samples
XML	8	Weak	9,467 (534)
JavaScript	98	Strong	20,845 (2,569)

- **Heuristic 1: Restricting the size of test inputs.** We limit the size of test inputs (i.e., tar and pro in Algorithm 3) as 10,000 bytes long (Line 3 and 10). Hence we do not apply tree-based mutation to tar if tar is more than 10,000 bytes long; and we do not use subtrees of pro as the content provider of mutation if pro is more than 10,000 bytes long. The reasons are that, a larger test input usually needs a larger number of mutations; more memory is required to store the AST of a larger test input; and a larger test input often has a slower execution speed.
- **Heuristic 2: Restricting the number of mutations.** If there are more than 10,000 subtrees in tar and pro, we randomly select 10,000 from all subtrees in S as the content provider of mutation (Line 16). Thus, we keep the number of mutations on each test input in the queue under 10,000 to make sure that each test input in the queue has the chance to get mutated.
- **Heuristic 3: Restricting the size of subtrees.** We limit the size of subtrees (i.e., each subtree in S in Algorithm 3) as 200 bytes long (Line 7 and 12). Thus we do not use the subtrees of tar and pro as the content provider of mutation if the subtree is more than 200 bytes long. Notice that 200 bytes are long enough to include complex statements.

The threshold values in these heuristics were empirically established as good ones.

Example. Fig. 6 shows an example of our tree-based mutation. The left-side is the AST of the test input to be mutated (i.e., tar in Algorithm 3), and the right-side is the AST of the test input that provides the content of mutation (i.e., pro in Algorithm 3). Here the subtree corresponding to the expression $x+2$ in tar is replaced with the subtree corresponding to the expression $Number(x)$ in pro, resulting in a new test input.

III. Evaluation

We implemented Superion in 3,372 lines of C/C++ code by extending AFL [71]. Particularly, given the grammar of test inputs, we adopted ANTLR 4 [50] to generate the lexer and parser, and used ANTLR 4 C++ runtime to parse test inputs and realize our trimming and mutation strategies. Hence, our approach is general and easily adoptable for other structured test inputs.

A. Evaluation Setup

To evaluate the effectiveness and generality of our approach, we selected two target languages and four target programs, and compared our approach with AFL [71] with respect to the bug-finding capability and code coverage.

Target Languages. We chose XML and JavaScript as the target languages with different structure level. Their grammars are all publicly available in ANTLR's community [1]. In particular, XML is a widely-used markup language. As shown in the second column of Table I, the XML grammar only contains eight symbols. Thus, XML can be considered to be weakly-structured. JavaScript is an interpreted language, and its grammar contains 98 symbols. Thus, its structure level can be regarded as strong.

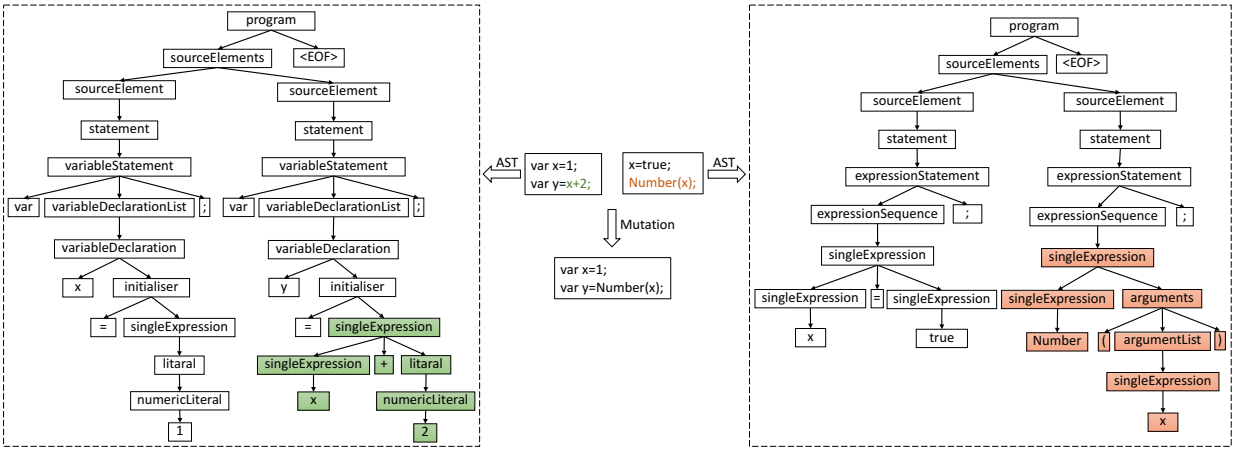


Fig. 6: An Example of Tree-Based Mutation

TABLE II: Target Programs and Their Fuzzing Configuration

Program	Version	# Lines	# Func.	Coverage	Timespan
libplist	1.12	3,317	316	Edge	3 months
WebKit	602.3.12	151,807	60,340	Block	3 months
Jerryscript	1.0	19,963	1,100	Edge	3 months
ChakraCore	1.10.1	236,881	74,132	Block	3 months

As indicated by the last column of Table I, we crawled 9,467 XML samples from the Internet, and 20,845 JavaScript samples from the test inputs of the two open-source JavaScript engines WebKit and Jerryscript. They were used as the initial test inputs (i.e., seeds) for fuzzing. As suggested by AFL, *afl-cmin* should be used to identify the set of functionally distinct seeds that exercise different code paths in the target program when a large number of seeds are available. Therefore, we used *afl-cmin* on the samples, and identified 534 and 2,569 distinct XML and JavaScript samples as the seeds for fuzzing, as shown in the parentheses in the last column of Table I. Notice that, before fuzzing, we pre-processed the JavaScript samples by removing all the comments because comments account for a considerable percentage of waste of mutation.

Target Programs. We selected one open-source XML engine libplist and three open-source JavaScript engines WebKit, Jerryscript and ChakraCore as the programs for fuzzing. The first four columns of Table II list the program details, including the version, the number of lines of code, and the number of functions. Particularly, libplist is a small portable C library to handle Apple Property List format files in binary or XML. It is widely used on iOS and Mac OS. WebKit is a cross-platform web browser engine. It powers Safari, iBooks and App Store, and various Mac OS, iOS and Linux applications. Jerryscript is a lightweight JavaScript engine for Internet of Things, intended to run on a very constrained device. ChakraCore is the core part of the Chakra JavaScript engine that powers Microsoft Edge. We chose these programs because they are security-critical and widely-fuzzed. Thus, finding bugs in them are significant.

As shown in the fifth column of Table II, we used edge coverage for libplist and Jerryscript during fuzzing, but block coverage for others due to non-determinism (i.e., different executions of a test input lead to different coverage). Besides, we excluded the non-deterministic code in WebKit and ChakraCore from instrumentation, following the technique in kAFL [58].

At the time of writing, we have fuzzed these programs for about three months. For libplist and Jerryscript, we have completed more than 100 cycles of fuzzing. For WebKit and ChakraCore, due to their large size, we have not finished one cycle yet. Here a cycle means the fuzzer went over all the interesting test inputs (triggering new coverage) discovered so far, fuzzed them, and looped back to the very beginning.

Research Questions. Using the previous evaluation setup, we aim to answer the following five research questions.

- **RQ1:** How is the bug-finding capability of Superion?
- **RQ2:** How is the code coverage of Superion?
- **RQ3:** How effective is our grammar-aware trimming?
- **RQ4:** How effective is our grammar-aware mutation?
- **RQ5:** What is the performance overhead of Superion?

We conducted all the experiments on machines with 28 Intel Xeon CPU E5-2697v3 cores and 64GB memory, running 64-bit Ubuntu 16.04 as the operating system.

B. Discovered Bugs and Vulnerabilities (RQ1)

Table III lists the unique bugs found by Superion. In libplist, we discovered 11 new bugs, from which we found 10 new vulnerabilities with CVE identifiers assigned. In WebKit, 16 new bugs were found. Seven of them were vulnerabilities with five CVE identifiers assigned, and others are pending for advisories. It is worth mentioning that these bugs obtained high appraisals, e.g.,"*Thank you for the awesome test case*" and "*This bug has existed for a long time. A quick look through blame would say for 4-5 years or so*". In Jerryscript, we found four previously unknown bugs, from which we found four vulnerabilities with three CVE identifiers assigned. In ChakraCore, we discovered three new bugs, and one of them was a vulnerability. Note that we received 3.2K USD bug bounty rewards.

With respect to the type of these bugs (see the third column of Table III), 12 of them are buffer overflow, 2 of them are integer overflow, 4 of them are memory corruption, 2 of them are arbitrary address access, 1 of them is uninitialized memory read and

TABLE III: Unique Bugs Discovered by Superion

Program	Bug	Type	AFL	jsfunfuzz
libplist	CVE-2017-5545	Buffer Overflow	✗	N/A
	CVE-2017-5834	Buffer Overflow	✓	N/A
	CVE-2017-5835	Memory Corruption	✓	N/A
	CVE-2017-6435	Memory Corruption	✗	N/A
	CVE-2017-6436	Memory Corruption	✗	N/A
	CVE-2017-6437	Buffer Overflow	✓	N/A
	CVE-2017-6438	Buffer Overflow	✓	N/A
	CVE-2017-6439	Buffer Overflow	✗	N/A
	CVE-2017-6440	Memory Corruption	✗	N/A
	Bug-90	Assertion Failure	✗	N/A
	CVE-2017-7440	Integer Overflow	✓	N/A
WebKit	CVE-2017-7095	Arbitrary Access	✗	✗
	CVE-2018-4378	Use-After-Free	✗	✗
	CVE-2018-4392	Buffer Overflow	✗	✗
	CVE-2017-7102	Arbitrary Access	✗	✗
	CVE-2017-7107	Integer Overflow	✗	✗
	Bug-191058	Assertion Failure	✗	✗
	Bug-192464	Uninitialized Memory Read	✗	✗
	Bug-185645	Null Pointer Deref	✗	✗
	Bug-188917	Assertion Failure	✗	✗
	Bug-170989	Assertion Failure	✗	✗
	Bug-170990	Assertion Failure	✗	✗
	Bug-172346	Null Pointer Deref	✗	✗
	Bug-172957	Null Pointer Deref	✗	✗
	Bug-172963	Buffer Overflow	✗	✗
	Bug-173305	Assertion Failure	✗	✗
	Bug-173819	Assertion Failure	✗	✗
Jerryscript	CVE-2017-18212	Buffer Overflow	✗	N/A
	CVE-2018-11418	Buffer Overflow	✓	N/A
	CVE-2018-11419	Buffer Overflow	✗	N/A
	Bug-2238	Buffer Overflow	✗	N/A
ChakraCore	CVE-2019-0648	Buffer Overflow	✗	✗
	Bug-5533	Null Pointer Deref	✗	✗
	Bug-5532	Null Pointer Deref	✗	✗

1 of them is use-after-free. These are all vulnerabilities. Besides, 5 of them are null pointer dereference, and 7 of them are assertion failure. These are all denial of service bugs. All these 34 bugs have been confirmed, and 25 of them have been fixed.

Comparison to AFL. Among these 34 bugs, AFL only discovered six of them (as shown in the fourth column of Table III) in three months and did not discover any other new bugs. This indicates that Superion significantly improves the bug finding capability of coverage-based grey-box fuzzers, which owes to the grammar-awareness in Superion. Specifically, for relatively weakly-structured inputs such as XML, AFL discovered 5 bugs, while Superion not only found all these 5 bugs, but also found 6 more bugs than AFL. Differently, for highly-structured inputs such as JavaScript, AFL barely found any bugs. Only one bug about utf-8 encoding problem was found by AFL in Jerryscript. All other bugs in JavaScript engines were found by Superion's tree-based mutation. This shows the significance of injecting grammar-awareness into coverage-based grey-box fuzzers.

Comparison to jsfunfuzz. We also compared Superion with jsfunfuzz [57], which is a successful grammar-aware fuzzer specifically designed for testing JavaScript engines. jsfunfuzz can be used to fuzz WebKit and ChakraCore; but it fails to fuzz Jerryscript because its generated JavaScript inputs have many JavaScript features that are not supported by Jerryscript. After three months of fuzzing, jsfunfuzz only found hundreds of out-of-memory crashes in WebKit and ChakraCore, but failed to find any bugs (as indicated by the last column of Table III). This is because jsfunfuzz uses manually-specified rules to express the grammar rules the generated inputs should satisfy. However, it is daunting, or even impossible to manually express all the required rules. Instead, Superion directly uses the grammar automatically during trimming and mutation.

TABLE IV: Code Coverage of the Target Programs

Program	Line Coverage (%)			Function Coverage (%)		
	Seeds	AFL	Superion	Seeds	AFL	Superion
libplist	33.3	50.8	68.9	27.5	32.6	40.8
WebKit	52.4	56.0	78.0	35.1	37.0	49.5
Jerryscript	81.3	84.0	88.2	76.0	77.1	78.2
ChakraCore	46.7	54.5	76.9	40.7	49.8	63.2

> In summary, Superion can significantly improve the bug-finding capability of coverage-based grey-box fuzzers (e.g., we found 34 new bugs, among which we discovered 22 new vulnerabilities with 19 CVE identifiers assigned).

C. Code Coverage (RQ2)

Apart from the bug-finding capability, we also measured the code coverage of fuzzing. The results are shown in Table IV, including the line and function coverage of the target programs. In particular, we list the coverage achieved by initial seeds, AFL and Superion. The coverage was calculated using *afl-cov* [54]. We were not able to calculate the coverage for jsfunfuzz due to two reasons: jsfunfuzz does not keep the JavaScript samples executed; and jsfunfuzz is very efficient and executes millions of JavaScript samples until it triggers a crash, which makes the coverage computation infeasible.

For line coverage, the initial seeds covered 33.3% lines of libplist, 52.4% lines of WebKit, 81.3% lines of Jerryscript and 46.7% lines of ChakraCore. By fuzzing, AFL respectively increased their line coverage to 50.8%, 56.0%, 84.0% and 54.5%. On average, AFL further covered 7.9% of the code. Superion improved the line coverage to 68.9%, 78.0%, 88.2% and 76.9%, respectively; and it further covered 24.6% of the code on average. Overall, Superion outperformed AFL by 16.7% in line coverage, because the grammar-awareness in Superion carries the fuzzing exploration towards the application execution stage.

On the other hand, for function coverage, the initial seeds covered 44.8% functions on average, and AFL and Superion increased the function coverage to 49.1% and 57.9%, respectively. Generally, Superion outperformed AFL by 8.8% in function coverage due to its grammar-awareness.

> In summary, Superion can significantly improve the code coverage of coverage-based grey-box fuzzers (e.g., 16.7% in line coverage and 8.8% in function coverage).

D. Effectiveness of Grammar-Aware Trimming (RQ3)

Table V compares the trimming ratio (i.e., the ratio of bytes trimmed from test inputs) and the grammar validity ratio (i.e., the ratio of test inputs that are grammar-valid after trimming) using the built-in trimming in AFL and the tree-based trimming in Superion. Numerically, for libplist, the built-in trimming in AFL trimmed out 21.7% of bytes in XML test inputs on average,

TABLE V: Comparison Results of Trimming Strategies

Program	Trimming Ratio (%)		Grammar Validity Ratio (%)	
	Built-In	Tree-Based	Built-In	Tree-Based
libplist	21.7	11.7	74.1	100
WebKit	10.6	7.6	86.4	100
Jerryscript	5.1	4.7	89.3	100
ChakraCore	12.7	11.3	83.7	100

while our tree-based trimming trimmed out 11.7% on average. On the other hand, 74.1% of test inputs after the built-in trimming were grammar-valid, but 100% of test inputs after our tree-based trimming were grammar-valid and can be further used to conduct our grammar-aware mutation.

Similarly, the built-in trimming respectively trimmed out 10.6%, 5.1% and 12.7% of bytes in JavaScript test inputs for WebKit, Jerryscript and ChakraCore, while our tree-based trimming respectively trimmed out 7.6%, 4.7% and 11.3% for WebKit, Jerryscript and ChakraCore. On the other hand, our tree-based trimming increased the grammar validity ratio for WebKit, Jerryscript and ChakraCore from 86.4%, 89.3% and 83.7% to 100%, which can facilitate our grammar-aware mutation by improving the chance of applying grammar-aware mutation (which is more effective in generating test inputs that can trigger new coverage as will be discussed in Section III-E).

> In summary, although with a relatively low trimming ratio, our grammar-aware trimming strategy can significantly improve the grammar validity ratio for the test inputs after trimming, which facilitates our grammar-aware mutation.

E. Effectiveness of Grammar-Aware Mutation (RQ4)

To evaluate the effectiveness of our grammar-aware mutation strategies, we compared them with those built-in mutation strategies of AFL [73], which include bit flips (*flip1*/*flip2*/*flip4* – one/two/four bit(s) flips), byte flips (*flip8*/*flip16*/*flip32* – one/two/four byte(s) flips), arithmetics (*arith8*/*arith16*/*arith32* – subtracting or adding small integers to 8-/16-/32-bit values), value overwrite (*interest8*/*interest16*/*interest32* – setting "interesting" 8-/16-/32-bit values to 8-/16-/32-bit values), *havoc* (random application of bit flips, byte flips, arithmetics, and value overwrite), and *splice* (splicing together two random test inputs from the queue, and then applying havoc). For the ease of presentation, our enhanced dictionary-based mutation strategy is referred to as *ui* (insertion of user-supplied tokens), *uo* (overwrite with user-supplied tokens), *ai* (insertion of automatically extracted tokens), and *ao* (overwrite with automatically extracted tokens); and our tree-based mutation strategy is referred to as *tree*.

Fig. 7 shows the number of interesting test inputs (i.e., triggering new coverage) discovered by different mutation strategies as we fuzzed WebKit. Because of space limit, we omit the similar results for the other three projects. The x-axis denotes the number of test inputs that Superion sequentially took from the queue and processed, and the y-axis denotes the corresponding number of interesting test inputs produced by different mutation strategies. As the process of different test inputs often takes different time, we do not use time to represent the x-axis. Besides, for clarity, Fig. 7 omits the results when all the mutation strategies become ineffective in continuously producing interesting test inputs (i.e., when the curves in Fig. 7 change gently).

The results vary across different seeds. Even with seeds fixed, the results may also vary across different runs due to the random nature of some mutation strategies (i.e., *havoc*, *splice* and *tree*). However, the trend remains the same across runs, and we only discuss the trend which holds across runs. In the beginning, bit and byte flips take a leading position in producing interesting test inputs. The reasons are that i) bit and byte flips often destroy the input structure, and trigger previously unseen error handling paths; and ii) bit and byte flips are the first mutation strategy to be sequentially applied, thus having the opportunity to first trigger the new coverage that could also be triggered by other mutation strategies. Gradually, the number of interesting test inputs generated by our grammar-aware mutation strategies outperform other mutation strategies. Specifically, *tree* and *uo* significantly outperform other mutation strategies. These results indicate that grammar-aware mutation strategies are effective in producing interesting test inputs.

Besides, we also explore the efficiency of different mutation strategies in producing interesting test inputs. The results are shown in Fig. 8, where the x-axis is the same to Fig. 7 and the y-axis denotes the ratio of interesting test inputs to the total number of generated test inputs. Surprisingly, all the mutation strategies are very inefficient in producing interesting test inputs, i.e., only two of the 1000 mutated test inputs can trigger new coverage. Thus, a huge amount of fuzzing efforts are wasted in mutating and executed test inputs. Therefore, adaptive mutation rather than exhaustive mutation should be designed to smartly apply mutation strategies.

Moreover, to evaluate our enhancement to dictionary-based mutation, we compared the dictionary overwrite and insertion in AFL with those in Superion. The results are reported in Fig. 9, where the x-axis is the same to Fig. 7, and the y-axis in Fig. 9a and Fig. 9b represent the number of times each mutation is applied and the number of interesting test inputs generated. We can see that our enhanced dictionary-based mutation greatly decreases the number of mutation applications by half, while still generating significantly more interesting test inputs.

> In summary, our grammar-aware mutation strategies are effective in generating test inputs that trigger new coverage, compared to the built-in mutation strategies in AFL. The efficiency of all mutation strategies needs to be improved.

F. Performance Overhead (RQ5)

The fuzzing process of a test input includes three major steps: parsing, mutation and execution. Among them, the parsing step is one-off for each test input, followed by a large number of mutations and executions. In Fig. 10a and 10b, we show the parsing time of JavaScript/XML test inputs in milliseconds (the y-axis) with respect to the size of test input files in bytes (the x-axis). Without loss of generality, we only report the results for the test inputs kept in the queue. In detail, the parsing time includes the time to read, parse and traverse a test input file. Approximately, the parsing time is linearly correlated to the size

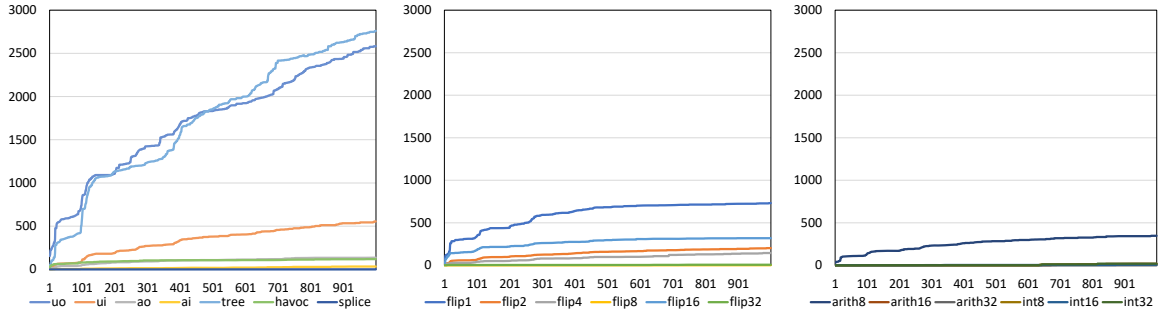


Fig. 7: The Effectiveness of Different Mutation Strategies in Producing Test Inputs that Trigger New Coverage

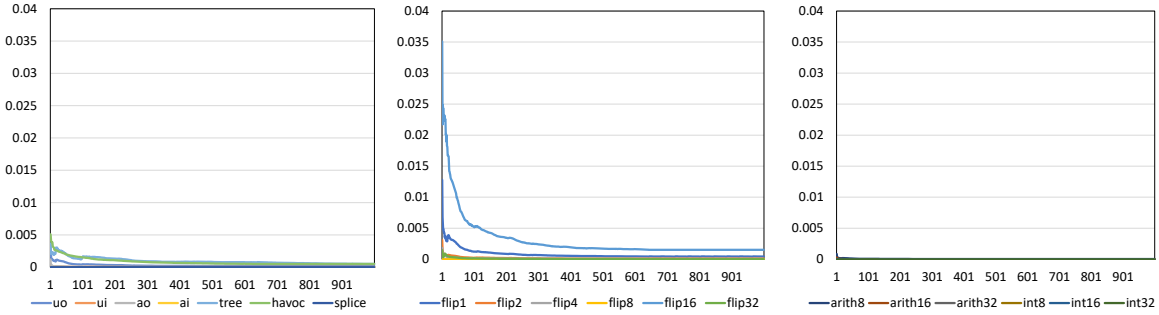
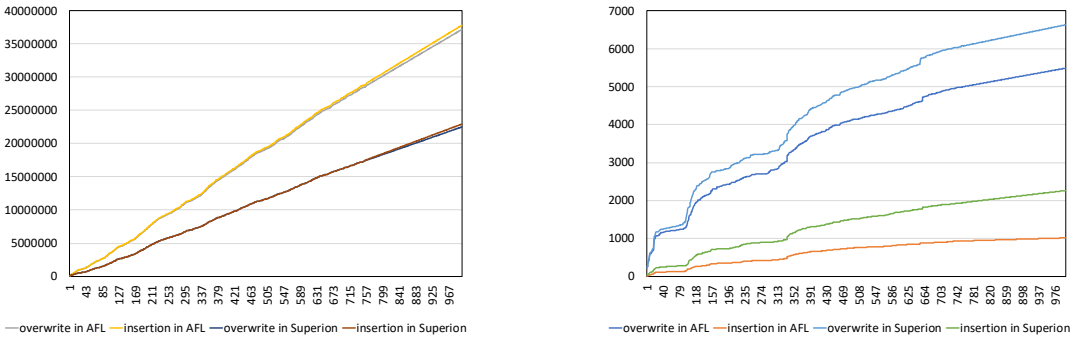


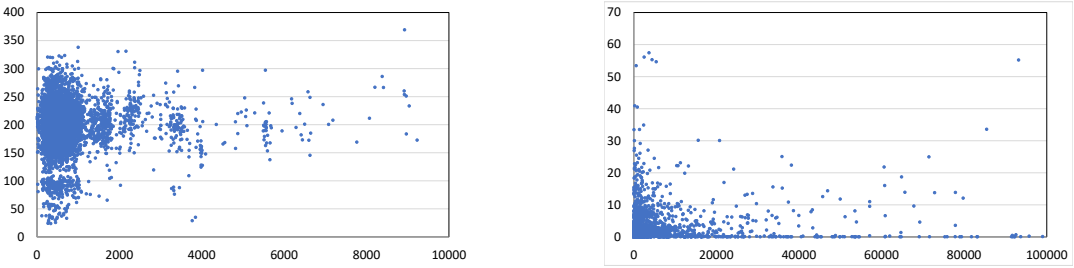
Fig. 8: The Efficiency of Different Mutation Strategies in Producing Test Inputs that Trigger New Coverage



(a) The Number of Mutation Applications

(b) The Effectiveness in Producing Interesting Test Inputs

Fig. 9: Comparison Results of Dictionary-Based Mutations



(a) JavaScript Test Inputs

(b) XML Test Inputs

Fig. 10: The Time to Read, Parse and Traverse Test Inputs with Respect to Different Size

of test input files. JavaScript test inputs' size is mostly under 10 KB and their parsing time is 199.3 milliseconds on average; and the parsing time of XML test inputs is 2.0 milliseconds on average. Notice that the parser generated using ANLTR is not optimized for the performance. We may reduce the execution time further by improving the parser's implementation.

Apart from the parsing time, the major performance overhead Superion imposes on mutation and execution is caused by our tree-based mutation. Table VI reports the overhead of applying tree-based mutation (in the second column) as well as the corresponding overhead of executing the mutated test input (in the third column). For small projects like libplist, it is very fast to

TABLE VI: Performance Overhead on Target Programs

Program	Tree-Based Mutation (ms)	Execution (ms)
libplist	0.63	0.39
WebKit	5.65	12.50
Jerryscript	5.65	3.57
ChakraCore	5.65	20.00

perform tree-based mutation and execution, i.e., the mutation took 0.63 ms and the execution took 0.39 ms on average. For large projects such as WebKit, Jerryscript and ChakraCore, the execution took much more time; e.g., executing a JavaScript input on ChakraCore took 20.00 ms, while the mutation took 5.65 ms on average. Considering the improvements to bug-finding capability and code coverage, the performance overhead introduced by Superion is acceptable.

In summary, Superion introduces additional overhead due to our grammar-aware tree-based mutation strategy. However, such overhead is still acceptable considering the improved bug-finding capability and code coverage.

G. Case Study

The JavaScript code fragment in Fig. 11 gives a representative test input that was generated by Superion and triggered an integer overflow vulnerability in WebKit, assigned CVE-2017-7107. In particular, this vulnerability is triggered because the method `setInput` in class `RegExpCachedResult` forgets to reify the `leftContext` and `rightContext`. As a result, when later WebKit attempts to reify them, it will end up using indices into an old input string to create a substring of a new input string. For the test input in Fig. 11, WebKit tried to get a substring through `jsSubstring`, whose length is 1 (i.e., length of "a") - 2 (i.e., `m_result.end` of "ss") = -1, as shown in Fig. 12, which is a very large number when treated as positive. Thus, an integer overflow vulnerability is caused.

The test input in Fig. 11 was actually simplified from a large test input for the ease of presentation. It was generated by applying our tree-based mutation on the two test inputs in Fig. 13 and Fig. 14. This proof-of-concept was not generated through one mutation, but was generated after several times of mutations. The intermediate test inputs that triggered new coverage were kept and added to the queue for further mutations. Eventually, it evolved into the proof-of-concept. This vulnerability was not triggered by AFL. This indicates that AFL's built-in mutation is not effective in fuzzing programs that process structured inputs, where our tree-based mutation becomes effective.

H. Discussion

Threats. First, we did not evaluate Superion on standardized data sets, e.g., LAVA [21] and CGC [2]. Many of the programs in these data sets process unstructured inputs, or are difficult to come up with a grammar. Hence, we did not use them. Instead, we used four real-life, large-scale, well-fuzzed programs. Second, we did not empirically compare Superion with the two mostly closely related grammar-aware mutation-based fuzzers, LangFuzz [32] and IFuzzer [62]. LangFuzz is not publicly available. It heavily relies on the seed, which is a collection of proof-of-concepts (POCs) that are difficult to obtain. Superion does

```
var str="ss";
var re=str.replace(/\b\w+\b/g);
RegExp.input="a";
RegExp.rightContext;
```

Fig. 11: A Proof-of-Concept of CVE-2017-7107

```
JSString* RegExpCachedResult::rightContext(ExecState* exec, JSObject* owner)
{
  // Make sure we're reified.
  lastResult(exec, owner);
  if (!m_reifiedRightContext) {
   unsigned length = m_reifiedInput->length();
   m_reifiedRightContext.set(exec->vm(), owner, m_result.end != length ?
      jsSubstring(exec, m_reifiedInput.get(), m_result.end, length - m_result.end)
      : jsEmptyString(exec));
  }
  return m_reifiedRightContext.get();
}
```

Fig. 12: The Vulnerable Code Fragment for CVE-2017-7107

```
...
var str = "ss"
var re=str.replace(/\b\w+\b/g);
...
```

Fig. 13: Source Test Input to Trigger CVE-2017-7107

```
...
write('RegExp.input: ' + RegExp.input);
...
write('RegExp.rightContext: ' + RegExp.rightContext);
...
```

Fig. 14: Source Test Input to Trigger CVE-2017-7107

not require such prior knowledge, and thus we did not compare Superion with LangFuzz. IFuzzer is open-source, but it lacks sufficient documentation to set up. Moreover, most bugs found by IFuzzer were not vulnerabilities, and only one CVE was exposed in their evaluation. Instead, we compared Superion with jsfunfuzz, a successful grammar-aware generation-based fuzzer for JavaScript engines. Third, we did not have a statistical significance analysis argued by Klees et al. [38]. This is due to the large time scale and resources involved in fuzzing real-life and large-scale programs for finding serious vulnerabilities.

Limitation. Superion needs a user-provided grammar, which limits the applicability to only publicly documented formats that have specified grammars. Therefore, Superion may have trouble finding proprietary grammars or undocumented extensions to standard grammars. However, several automatic grammar inference techniques [7, 28, 33, 34, 63] have been proposed, we plan to integrate such techniques to have wider applicability.

IV. Related Work

Instead of listing all related work, we focus our discussion on the most relevant fuzzing work in five aspects: guided mutation, grammar-based mutation, block-based generation, grammar-based generation, and fuzzing boosting.

Guided Mutation. Mutation-based fuzzing was proposed to generate test inputs by randomly mutating well-formed test inputs [46]. Then, a large body of work has been developed to use heuristics to guide mutation. AFL [71], Steelix [43], FairFuzz [42] and CollAFL [23] use coverage to achieve the guidance, and SlowFuzz [52] and PerfFuzz [41] further use resource usage to realize the guidance. BuzzFuzz [24], Vuzzer [55] and Angora [15] leverage taint analysis to identify those interesting bytes for mutation. SAGE [26, 27], Babić et al. [6], Pham et al. [53] and

Badger [48] leverage symbolic execution to facilitate fuzzing. Dowser [30], TaintScope [65] and BORG [47] integrate taint analysis with symbolic execution to guide fuzzing. Driller [59] combines fuzzing and concolic execution to discover deep bugs. Kargén and Shahmehri [37] perform mutations on the machine code of the generating programs instead of directly on a test input in order to leverage the information about the input format encoded in the generating programs. In summary, these fuzzing techniques target programs that process compact or unstructured inputs, which become less effective for programs that process structured inputs. Complementary to them, Superion can effectively fuzz programs that process structured inputs.

It is worth mentioning that application-specific fuzzers have been attracting great interests, e.g., compiler fuzzing [16, 18, 39, 40, 44, 60], kernel fuzzing [17, 31, 58], IoT (Internet of Things) fuzzing [14], OS fuzzing [49] and smart contract fuzzing [36]. It is interesting to investigate how to extend our general-purpose fuzzer (e.g., by designing new mutation operators or feedback mechanisms) to be effective in fuzzing specific applications.

Grammar-Based Mutation. Several techniques have been proposed to perform mutations based on grammar. MongoDB's fuzzer [29] wreaks controlled havoc on the AST of a JavaScript test input. While our tree-based mutation is similar, Superion conducts the mutations in an incremental way by keeping those interesting intermediate test inputs for further fuzzing. Similarly, µ4SQLi [5] applies a set of mutation operators on valid SQLs to generate syntactically correct and executable SQLs that can reveal SQL vulnerabilities. However, both MongoDB and µ4SQLi are specifically designed for JavaScript or SQL, and hence they may not work for other structured inputs. Superion is general for other structured inputs as long as their grammar is available.

LangFuzz [32] uses a grammar to separate previously failing test input to code fragments and save them into a fragment pool. Then, some code fragments of a test input are mutated by replacing them with the same type of code fragments in the pool. Similarly, IFuzzer [62] uses the grammar to extract code fragments from test inputs and recomposes them in an evolutionary way. Different from these two blackbox fuzzers, Superion brings grammar-awareness into coverage-based greybox fuzzers.

Block-Based Generation. As some bytes in a test input are used collectively as a single value in the program, they should be considered together as a block during fuzzing. Following this observation, TestMiner [19] first mines literals from a corpus of test inputs and then queries the mined data for values suitable for a given method under test. These predicted values are then used as test inputs during test generation. It is not clear whether it works well for highly-structured inputs such as JavaScript as they experimented with simple formats such as IBAN, SQL, E-mail and Network address. Spike [4] and Peach [3] use input models, specifying the format of data chunks and integrity constraints, to regard test inputs as blocks of data, and leverage mutations to generate new test inputs. While being effective in fuzzing programs that process weakly-structured inputs (e.g., images and protocols), these approaches become less effective for highly-structured inputs (e.g., JavaScript). Complementary to them, Superion is designed for such highly-structured inputs.

Grammar-Based Generation. Another line of work is to use the grammar to directly generate test inputs. mangleme [72] is an automated broken HTML generator and browser fuzzer. jsfunfuzz [57] uses specific knowledge about past and common vulnerabilities and hard-coded rules to generate new test inputs. Dewey et al. [20] propose to use constraint logic programming for program generation. Valotta [61] uses his domain knowledge to manually build a fuzzer to test browsers. While being effective in finding vulnerabilities, they all rely on some hard-coded or manually-specified rules to express semantic rules, which hinder their applications to a wider audience.

Godefroid et al. [25] apply symbolic execution to generate grammar-based constraints, and use grammar-based constraint solver to generate test inputs. CSmith [69] iteratively and randomly selects one production rule in the grammar to generate C programs. Domato [22] generates test inputs from scratch given the grammars that specify HTML/CSS structures and JavaScript objects, properties and functions. Domato also fuzzed WebKit for three months; but none of our bugs were found by Domato. This is a strong evidence that Superion has the characteristics that grammar-aware fuzzers without coverage feedback do not have. Skyfire [64] and TreeFuzz [51] learn a probabilistic model from the grammar and a corpus of test inputs to generate test inputs. They are generation-based, while Superion is grammar-aware mutation-based, which incrementally utilizes the interesting behaviors embedded in previous interesting test inputs.

Fuzzing Boosting. Another thread of work focuses on improving the efficiency of fuzzing, e.g., seed selection [56], seed scheduling [9, 66], parameter tuning [10, 35], directed fuzzing [8, 12, 13] to reproduce crashes or assess potential bugs found by vulnerable code matching [11, 68], and operating primitives [67]. These boosting techniques are orthogonal to Superion.

V. Conclusions

In this paper, we propose a grammar-aware coverage-based greybox fuzzing approach, Superion, for programs that process structured inputs. Specifically, we propose a grammar-aware trimming strategy and two grammar-aware mutation strategies to effectively trim and mutate test inputs while keeping the input structure valid, quickly carrying the fuzzing exploration into width and depth. Our experimental study on several XML and JavaScript engines has demonstrated that Superion improved code coverage and bug-finding capability over AFL. Moreover, Superion found 34 new bugs, among which 22 new vulnerabilities were discovered and 19 CVEs were assigned.

Acknowledgment

We would like to thank Michał Zalewski for the American Fuzzy Lop fuzzer. This research was supported (in part) by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30, Award No. NRF2016NCR-NCR002-026) and administered by the National Cybersecurity R&D Directorate. Bihuan Chen is the corresponding author of this paper.

REFERENCES

[1] Antlr's grammar list for different languages. [Online]. Available: https://github.com/antlr/grammars-v4

[2] Cyber grand challenge (cgc). [Online]. Available: http://archive.darpa.mil/cybergrandchallenge/

[3] Peach fuzzer platform. [Online]. Available: http://www.peachfuzzer.com/products/peach-platform/

[4] Spike fuzzer platform. [Online]. Available: http://www.immunitysec.com/

[5] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: an input mutation approach," in *ISSTA*, 2014, pp. 259–269.

[6] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *ISSTA*, 2011, pp. 12–22.

[7] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *PLDI*, 2017, pp. 95–110.

[8] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *CCS*, 2017.

[9] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *CCS*, 2016, pp. 1032–1043.

[10] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *SP*, 2015, pp. 725–741.

[11] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *FSE*, 2016, pp. 678–689.

[12] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu, "Fot: A versatile, configurable, extensible fuzzing framework," in *ESEC/FSE*, 2018, pp. 867–870.

[13] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzing," in *CCS*, 2018, pp. 2095–2108.

[14] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *NDSS*, 2018.

[15] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *SP*, 2018.

[16] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *PLDI*, 2013, pp. 197–208.

[17] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *CCS*, 2017, pp. 2123–2138.

[18] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *ISSTA*, 2018, pp. 95–105.

[19] L. Della Toffola, C.-A. Staicu, and M. Pradel, "Saying hi!is not enough: Mining inputs for effective test generation," in *ASE*, 2017.

[20] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *ASE*, 2014, pp. 725–730.

[21] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *S&P*, 2016, pp. 110–121.

[22] I. Fratric. (2017) The great dom fuzz-off of 2017. [Online]. Available: https://googleprojectzero.blogspot.sg/2017/09/the-great-dom-fuzz-off-of-2017.html

[23] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *SP*, 2018.

[24] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *ICSE*, 2009, pp. 474–484.

[25] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *PLDI*, 2008, pp. 206–215.

[26] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *NDSS*, 2008.

[27] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[28] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *ASE*, 2017, pp. 50–59.

[29] R. Guo, "Mongodb's javascript fuzzer," *Commun. ACM*, vol. 60, no. 5, pp. 43–47, 2017.

[30] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *USENIX Security*, 2013, pp. 49–64.

[31] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *CCS*, 2017, pp. 2345–2358.

[32] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *USENIX Security*, 2012, pp. 445–458.

[33] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *ASE*, 2016, pp. 720–725.

[34] M. Höschele and A. Zeller, "Mining input grammars with autogram," in *ICSE*, 2017, pp. 31–34.

[35] A. Householder and J. Foote, "Probability-based parameter selection for black-box fuzz testing," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2012-TN-019, 2012.

[36] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *ASE*, 2018.

[37] U. Kargén and N. Shahmehri, "Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing," in *FSE*, 2015, pp. 782–792.

[38] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *CCS*, 2018, pp. 2123–2138.

[39] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *PLDI*, 2014, pp. 216–226.

[40] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *OOPSLA*, 2015, pp. 386–399.

[41] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," in *ISSTA*, 2018, pp. 254–265.

[42] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *ASE*, 2018.

[43] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *ESEC/FSE*, 2017, pp. 627–637.

[44] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *PLDI*, 2015, pp. 65–76.

[45] G. Meng, Y. Liu, J. Zhang, A. Pokluda, and R. Boutaba, "Collaborative security: A survey and taxonomy," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 1:1–1:42, 2015.

[46] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[47] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, "The borg: Nanoprobing binaries for buffer overreads," in *CODASPY*, 2015, pp. 87–97.

[48] Y. Noller, R. Kersten, and C. S. Păsăreanu, "Badger: Complexity analysis with fuzzing and symbolic execution," in *ISSTA*, 2018, pp. 322–332.

[49] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing OS fuzzer seed selection with trace distillation," in *USENIX Security*, 2018.

[50] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[51] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," TU Darmstadt, Tech. Rep. TUD-CS-2016-14664, 2016.

[52] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *CCS*, 2017, pp. 2155–2168.

[53] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *ASE*, 2016, pp. 543–553.

[54] M. Rash. afl-cov - afl fuzzing code coverage. [Online]. Available: https://github.com/mrash/afl-cov

[55] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *NDSS*, 2017.

[56] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *USENIX Security*, 2014, pp. 861–875.

[57] J. Ruderman. (2007) Introducing jsfunfuzz. [Online]. Available: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz

[58] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for os kernels," in *USENIX Security*, 2017, pp. 167–182.

[59] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016.

[60] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *OOPSLA*, 2016, pp. 849–863.

[61] R. Valotta, "Taking browsers fuzzing to the next (dom) level," in *DeepSec*, 2012.

[62] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *ESORICS*, 2016, pp. 581–601.

[63] J. Viide, A. Helin, M. Laakso, P. Pietikäinen, M. Seppänen, K. Halunen, R. Puuperä, and J. Röning, "Experiences with model inference assisted fuzzing," in *WOOT*, 2008, pp. 2:1–2:6.

[64] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed

generation for fuzzing," in *SP*, 2017, pp. 579–594.
[65] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *SP*, 2010, pp. 497–512.
[66] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *CCS*, 2013, pp. 511–522.
[67] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *CCS*, 2017, pp. 2313–2328.
[68] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *CCS*, 2017, pp. 363–376.
[69] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *PLDI*, 2011, pp. 283–294.
[70] M. Zalewski. afl-fuzz: making up grammar with a dictionary in hand. [Online]. Available: https://lcamtuf.blogspot.sg/2015/01/afl-fuzz-making-up-grammar-with.html
[71] M. Zalewski. American fuzzy lop. [Online]. Available: http://lcamtuf.coredump.cx/afl/
[72] M. Zalewski. mangleme. [Online]. Available: http://freecode.com/projects/mangleme/
[73] M. Zalewski. Mutation strategies in american fuzzy lop. [Online]. Available: http://lcamtuf.coredump.cx/afl/status_screen.txt