Token-Level Fuzzing

Christopher Salls, *UC Santa Barbara;* Chani Jindal, *Microsoft;* Jake Corina, *Seaside Security;* Christopher Kruegel and Giovanni Vigna, *UC Santa Barbara*

https://www.usenix.org/conference/usenixsecurity21/presentation/salls

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

Token-Level Fuzzing

Christopher Salls
UC, Santa Barbara

Chani Jindal
Microsoft

Jake Corina
Seaside Security

Christopher Kruegel
UC, Santa Barbara

Giovanni Vigna
UC, Santa Barbara

Abstract

Fuzzing has become a commonly used approach to identifying bugs in complex, real-world programs. However, *interpreters* are notoriously difficult to fuzz effectively, as they expect highly structured inputs, which are rarely produced by most fuzzing mutations. For this class of programs, grammar-based fuzzing has been shown to be effective. Tools based on this approach can find bugs in the code that is executed after parsing the interpreter inputs, by following language-specific rules when generating and mutating test cases.

Unfortunately, grammar-based fuzzing is often unable to discover subtle bugs associated with the parsing and handling of the language syntax. Additionally, if the grammar provided to the fuzzer is incomplete, or does not match the implementation completely, the fuzzer will fail to exercise important parts of the available functionality.

In this paper, we propose a new fuzzing technique, called Token-Level Fuzzing. Instead of applying mutations either at the byte level or at the grammar level, Token-Level Fuzzing applies mutations at the *token level*. Evolutionary fuzzers can leverage this technique to both generate inputs that are parsed successfully *and* generate inputs that do not conform strictly to the grammar. As a result, the proposed approach can find bugs that neither byte-level fuzzing nor grammar-based fuzzing can find. We evaluated Token-Level Fuzzing by modifying AFL and fuzzing four popular JavaScript engines, finding 29 previously unknown bugs, several of which could not be found with state-of-the-art byte-level and grammar-based fuzzers.

1 Introduction

As the amount of software in the world grows, so does the need for effective bug-finding techniques. Unfortunately, it is very common for companies to employ far more developers than security engineers. BSIMM, a study of software security initiatives started by Synopsys, found that there was an average ratio of a single security engineer for every sixty software developers [29]. Consequently, security engineers are often responsible for very large amounts of code; far more than is feasible to check manually. As a result, it is imperative that effective automated techniques are used to identify security bugs.

In the past few years, fuzz testing has become widely popular. Fuzzers such as American Fuzzy Lop (AFL) [45], Syzkaller [17], and Libfuzzer [40] are responsible for the detection of hundreds of high-severity security issues. The success of these fuzzers, as well as others, has caused fuzzing to become a preeminent automated analysis for detecting memory corruption vulnerabilities. Fuzzing is employed by companies and organizations both for finding old bugs and as an additional test in continuous integration systems [15].

The popularity of fuzzing has inspired a vast amount of research to develop new techniques, tailored to a variety of targets. A particularly interesting target is represented by *interpreters*. Interpreters are in widespread use; they are found in many components of browsers, document viewers, programming languages, and more. As such, interpreters are often a high-value target for attackers, and a high-impact topic for security researchers.

Analyzing interpreters is challenging: Modern-day interpreters can be very complex (for example, V8, Google's JavaScript engine, contains over 700K lines of code), and, in addition, they expect highly structured inputs, composed of individual *tokens*. If the input does not match the format that the interpreter is expecting, it may throw an error very early in the processing (input parsing) step. As a result, many of the most common fuzzers fail to perform well when applied to interpreters, such as JavaScript engines, because their mutations typically result in simple syntax errors.

Because of the aforementioned issue, some fuzzers, which are targeted at the analysis of interpreters, use grammar-based approaches to generate and mutate inputs [13, 21, 26, 43]. Their goal is to generate inputs that exercise deeper code paths. While these approaches are effective, they also suffer from important limitations. First, they need to be given or be able to learn a grammar, which makes it difficult to

re-target them for different languages. Another limitation is that grammar-based fuzzers frequently conform too tightly to the supplied grammar and fail to generate unusual situations for the parser, potentially missing subtle bugs related to the syntactic analysis.

In this paper, we introduce a novel technique, called *Token-Level Fuzzing*. Token-Level Fuzzing can be thought of as a level in between the byte-level approaches and the grammar-based approaches typically employed by fuzzers. The basic idea behind Token-Level Fuzzing is to have the mutations operate with whole tokens, either replacing or inserting entire words. For example, instead of replacing a few random bytes, which has a small chance of producing an interesting input, a token-level approach would replace a few tokens in the input with different tokens, without taking into account grammar rules. This approach allows the fuzzer to have a much higher chance of producing useful mutations, while avoiding the strictness and complexity of grammar-based approaches.

We created a modified version of AFL, called *Token-Level AFL*, which implements Token-Level Fuzzing. Even though Token-Level AFL is specifically implemented for fuzzing JavaScript interpreters, the Token-Level Fuzzing technique itself is general. We tested Token-Level AFL against the most up-to-date versions of four major JavaScript interpreters, namely, V8, SpiderMonkey, JavaScriptCore, and ChakraCore, and we discovered 29 previously unknown bugs, some of which are severe and can lead to remote code execution.

In summary, we make the following contributions:

- We introduce a new technique, called Token-Level Fuzzing, for fuzzing language-based programs, such as interpreters;
- We implemented this technique to fuzz JavaScript engines. The implementation is done on top of AFL to take advantage of its efficient coverage-guided fuzzing;
- We evaluated Token-Level AFL on the latest versions of four major JavaScript engines, finding 29 previously unknown bugs;
- We compared our tool to other state-of-the-art JavaScript fuzzers, demonstrating that our tool is more effective at finding bugs.

2 Background and Related Work

Fuzzing is one of the most effective and scalable vulnerability discovery solutions. Fuzzers generate a vast number of test cases to exercise target applications and monitor their run-time execution to discover security bugs. Most fuzzing research can be characterized across three axes: Input generation, program access, and coverage goals.

Input Generation. There are two main classes of approaches to generate inputs: mutational fuzzing and generational fuzzing. Mutational fuzzing [4, 20, 34] modifies seeds of typically well-formed inputs to generate new inputs. Generational fuzzing, on the other hand, tends to be more structure-aware and leverages descriptions of the input format to generate inputs following that structure [19, 22, 23].

Program Access. Fuzzing approaches might differ in the level of insight they have into the execution of a target program. White-box fuzzing performs program analyses and collects constraints from conditional branches during execution. Solutions obtained from solving these constraints are then mapped to new inputs [14, 38]. Black-box fuzzing approaches, instead, do not have any access to the internals of the program being tested [9, 41]. Finally, in the middle, are grey-box fuzzing approaches, which use lightweight techniques to gather information about program execution, such as branch coverage [47].

Coverage Goals. Fuzzing approaches might have different goals when exercising a target program. For example, directed fuzzing has the objective of targeting a set of deep paths [2]. Coverage-based fuzzing, instead, uses different types of tracking such as block coverage, edge coverage, etc., to track the inputs that maximize code coverage, so that they are used as a basis for further mutations [5, 17, 45].

In the following, we provide more details about a subclass of mutational fuzzing approaches, called *evolutionary* approaches, and how they are applied to JavaScript fuzzing, as JavaScript interpreters are the target of our prototype.

2.1 Evolutionary Fuzzing

American Fuzzy Lop (AFL) is a grey-box fuzzer that leverages compile-time instrumentation [45] to collect meta-information about a target program's execution. AFL has been demonstrated to be extremely effective in finding vulnerabilities and other interesting bugs in many applications [46]. The main insight behind AFL's success is that inputs that exercise new paths in a program are best suited for fostering the bug-discovery process. Therefore, whenever AFL identifies an input that discovers a new path in the program, it uses that input as a basis for additional mutations, to see if these "evolved" inputs can cause the program to execute additional portions of code (measured in basic blocks).

This evolutionary approach has been extremely effective, and, therefore, there has been much research work on improving evolutionary fuzzing. For example, Vuzzer [34] focuses on extracting two main features, namely data-flow features (using taint analysis) and control-flow features, to create a smart feedback loop. These features, which are extracted using static analysis, help infer important properties of the inputs and prioritize/de-prioritize certain paths. AFLFAST, on the other hand, uses a Markov-chain-based search strategy to choose low-frequency paths, enabling the tool to explore more paths in the same fuzzing time [3]. Another evolutionary approach is Angora [5], which uses byte-level taint tracking and

gradient-based search algorithms in addition to input length exploration and context-sensitive branch count.

2.2 JavaScript Fuzzing

JavaScript engines are one of the most complicated components of modern-day browsers, making them a very popular target for both attackers and researchers. As a consequence, there have been continuous efforts towards improving fuzzing approaches to find JavaScript engine vulnerabilities. Most of these approaches fall into two categories: Grammar-based approaches and coverage-guided approaches.

2.2.1 Grammar-Based Approaches

Some of the most popular JavaScript fuzzers have been centered around generating syntactically correct test cases based on either a predefined grammar or a trained probabilistic language model. JSFunFuzz is one such JavaScript grammar-based fuzzers. JSFunFuzz relies purely on a generative approach to create new test cases [36], and has been used to exercise a wide range of JavaScript language features. Another example of a generative approach is Domato [11], which uses HTML, CSS, and JavaScript grammars to generate samples that target DOM-specific logic issues.

A different approach is followed by CodeAlchemist [23], which uses semantics-aware assembly to produce JavaScript code snippets. This approach breaks JavaScript seeds into code fragments, and each fragment is tagged with constraints and analyzed for used variables. The code fragments are then combined to produce syntactically and semantically correct test cases.

LangFuzz [24] also employed the concept of code fragments, combined with both generative and mutation-based fuzzing, to maintain the syntax and semantics of code samples. One key feature of LangFuzz is that it is language-independent, which means that it bases its testing strategy solely on grammar and existing programs and not language-specific information.

2.2.2 Coverage-Guided Approaches

Coverage-guided fuzzing has also been successful in finding JavaScript engine vulnerabilities. In these approaches, one of the most common targets for mutation is the Abstract Syntax Tree (AST) of JavaScript programs [21,43]. For example, Fuzzilli [20] developed an intermediate language, called FuzzIL, which supports better control-flow-based and data-flow-based decisions in the mutation process.

Nautilus [1] is another tool that performs mutations on the ASTs, with its unique point being that it also performs byte mutations on the raw code strings. Montage [27] also leverages the idea that fragments of the ASTs from the test cases can be combined in unique ways to find bugs, and they do so using machine learning. Superion [43] modifies AFL to perform fuzzing on the ASTs, using custom grammar-based mutation strategies to achieve both grammar-aware trimming and tree-based mutations. We compare our approach against Superion in Section 5.3, showing that our approach finds more bugs and has better code coverage.

Another work on JavaScript fuzzing that operates on ASTs was presented by Park, et al. [31]. The authors introduce the concept of aspect-preserving mutations. Their fuzzer, called *Die*, centers around the idea that there are key properties, or *aspects*, in the seeds present in test cases or other bug reports. The goal of this technique is to keep these beneficial properties from the original seed and retain them across mutations. For example, control-flow structures, like loops, can trigger JIT compilation, which, in turn, might reveal a buggy optimization logic; therefore, control-flow structures are an aspect that the fuzzer should specifically try to preserve during the mutation process. This fuzzer also performs mutations in a grammar-aware manner, with mutations performed on the ASTs.

3 Motivation

As discussed in the previous section, fuzzing research has come quite a long way from just generating purely random input. AFL, in particular, is a venerable fuzzer that has been instrumental in finding many bugs in over one hundred highly used targets. However, when AFL is applied to interpreters, such as JavaScript engines, some significant downsides begin to emerge. As most of the mutations that AFL performs are at a byte- or bit-level, we see it repeatedly generating inputs that simply fail to parse.

If we consider a simple bit-flip mutation on a small piece of JavaScript, the results will frequently look like the following mutations, which will immediately fail to parse:

```
while (bar.x) ⟶ whkle (bar.x)
              ⟶ whilep(bar.x)
              ⟶ while xbar.x)
              ⟶ while (bar.|)
```

It should be straightforward to see that mutations such as these are not particularly helpful; they will only cause simple syntax errors. As such, this approach to mutating the inputs would very likely not lead to more code coverage, and would simply waste execution time.

The ineffectiveness of byte-level fuzzing suggests that taking into account the rules governing input format might allow for a more comprehensive exploration of a JavaScript interpreter's code base. Grammar-based fuzzers are incredibly powerful in their ability to very quickly generate syntactically correct pieces of input for a given program. An obvious downside with this approach, however, is the work required to first define a grammar, or otherwise rely on an existing grammar definition before fuzzing can be performed [23,24,32,33,43].

```
function main() {
  const v1 = [13.37,13.37,13.37];
  const v6 = [1337,1337,v1];
  function v9(v10,...v1) {
    const v13 = [1337,1337,1337];
    return v13;
  }
  const v18 = v9(v6);
}
main();
```

Listing 1: Example of code generated by Fuzzilli. Fuzzilli follows a static single-assignment format for the generated code. As such, variables will always be assigned exactly once and some syntactic/semantic patterns cannot be emitted.

An additional downside to grammar-based fuzzing is the adherence to the grammar that is given to the fuzzer. This not only limits the fuzzer to creating code that matches the grammar, but it *also* limits the fuzzer to finding bugs that can be expressed as such. This will prevent most grammar-based fuzzers from finding bugs that *require* syntactically or semantically incorrect inputs to be triggered. Even bugs with unusual semantics can be unreachable by grammar-based fuzzers. This is because a grammar-based fuzzer, though powerful in its generational capabilities and language awareness, will only generate inputs that adhere to the grammar that has been supplied.

To explain this limitation further, we will show an example from Fuzzilli and talk about how its grammar limits the bugs it can find. Listing 1 shows an example input generated by Fuzzilli, which was taken when fuzzing a JavaScript engine. Note how each line assigns at most a single new variable, and variables are never overwritten. This is because Fuzzilli uses a static single-assignment intermediate representation [20], and the inputs it generates will conform tightly to it. While this feature is instrumental in achieving the the real-world results that Fuzzilli has published, it also limits the sorts of bugs that it is able to find. Any bug that requires as input a different or more complicated structure, such as redefining variables, will not be found. Furthermore, Fuzzilli will never create nested expressions and cannot output many JavaScript syntax errors.

Unsurprisingly, there are bugs that do require incorrect semantics or even incorrect syntax, as well as bugs that require unusual constructs. We will briefly look at an example of such a bug that was found in V8. Chromium issue 800032 [18] describes a high-impact bug found in V8 that could lead to remote code execution (RCE). Note that although the bug has high impact with potential for RCE, no CVE was assigned to it, as it was discovered internally by Google Project Zero member Jung Hoon Lee. The bug report includes the proof-of-concept in Listing 2, which triggers the issue.

```
class Sub extends RegExp {
  constructor(a) {
    // expected_nof_properties() skipped
    // due to error
    const a = 1; // semantic error
  }
}

let o = Reflect.construct(RegExp,[],Sub);
// OOB write
o.lastIndex = 0x1234;
```

Listing 2: Proof-of-concept code for Chromium Issue 800032. This code triggers an error, which causes a miscalculation in the number of properties leading to an exploitable out-of-bounds write.

```
function f() {
  ({
    a: {
      b = 0x1111, // invalid assignment
      c = 0x2222,
    }.c = 0x3333
  } = {});
}

f();
```

Listing 3: Proof of concept code for CVE-2017-8729, which was caused by a parser error in Edge. Line 4 (`b = 0x1111`) contains a syntax error by trying to assign to a member with = while creating an object.

The proof-of-concept code creates a subclass of a Regular Expression object. In the constructor of the subclass, there is an error. Specifically, the line `const a = 1` will attempt to redefine `a` as constant, which is invalid. Because of this error, the size of an object gets incorrectly computed, which can then lead to out-of-bounds reads and writes on the object. A fuzzing approach that follows a strict grammar definition would not be able to find issues such as this one.

Another example of a bug that could be difficult to find with a grammar-based fuzzer that strictly follows its grammar is shown in Listing 3. This example is CVE-2017-8729 of Edge [16], where the parser would incorrectly parse the code, and, in doing so, lead to a type confusion when assigning to the object member later. As this bug requires incorrect syntax to trigger, this example represents another case in which grammar-based fuzzers may suffer due to their adherence to the grammar.

We have just shown how grammar-based fuzzers may be unable to find certain bugs in interpreters, and previously, we showed how byte-level fuzzers, such as AFL, struggle to make

any progress in fuzzing language-based inputs. It is apparent there is a need for a new approach that can make progress and explore interpreters effectively, but without the limitations of a grammar. In order to find a way to utilize the powerful evolutionary capabilities of tools like AFL on interpreters inputs, we introduce a new technique, called *Token-Level Fuzzing*. Token-Level Fuzzing works at a higher level than bytes, but is not strictly bound to the language grammar, allowing it to find bugs that neither byte-level fuzzing nor grammar-based fuzzing would find.

4 Overview of Token-Level Fuzzing

The idea behind Token-Level Fuzzing is fairly simple: Valid tokens should be replaced with valid tokens. For example, when fuzzing the code shown in Section 3, instead of mutating individual characters in the word `while`, we would replace the entire word with a different word. If we replace `while` with `if` or `Number`, we would obtain a much better mutation. Below are examples of possible better mutations if we use Token-Level Fuzzing:

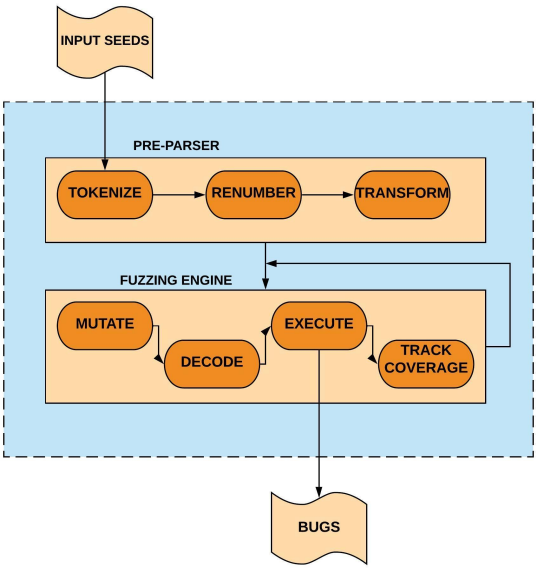
```
while (bar.x) ⟶ if (bar.x)
              ⟶ Number (bar.x)
              ⟶ while (bar+x)
              ⟶ while (while.x)
```

Notice that Token-Level Fuzzing can still produce invalid syntax, as is the case with the last line above: `while (while .x)`. Even mutations like that can be beneficial if they trigger a new error handler or if they can iteratively be mutated until a different valid JavaScript statement is reached.

A natural question to ask is how this technique compares to the "dictionary" that tools such as AFL [30] and LibFuzzer [28] allow users to provide. The first major difference is that AFL will still perform the byte-level mutations as well as the dictionary-based mutations. Second, the dictionary mutations are not aligned to tokens, so the fuzzer might insert the word `while` in the middle of a token instead of replacing the entire token. Finally, it may take multiple token additions/replacements to reach a new and interesting input. Some fuzzers, such as AFL, will only insert one dictionary word in a mutation, which limits its exploration.

Another question is how Token-Level Fuzzing compares to grammar-based fuzzing. Grammar-based fuzzing mutates inputs or generates inputs according to a grammar, whereas Token-Level Fuzzing does not follow any grammar. Token-Level Fuzzing can generate many patterns that can be difficult or impossible to produce for a particular grammar-based fuzzer, in particular those with complex or incorrect syntax. On the other hand, grammar-based fuzzers focus on exercising the interpreter with correct syntax, possibly allowing faster exploration of that part of the program. As a result, we expect that the two approaches complement each other well and are likely to find different bugs.

Figure 1: The architecture of Token-Level AFL. The tool has two primary components: The pre-parser and the fuzzing engine. The pre-parser is responsible for transforming input seeds into a list of 16-bit numbers. Then the fuzzing engine works on these lists, only decoding them back to JavaScript when they are passed to the target program.

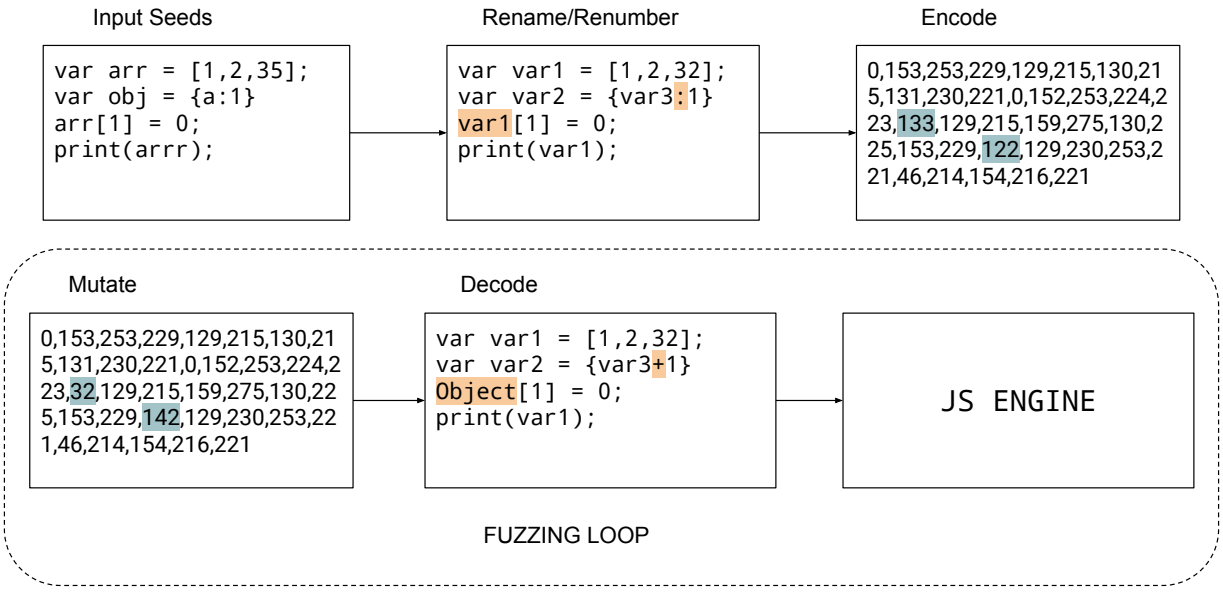


4.1 Method

To create a fuzzer that works at the *token level*, we start by constructing a map, which assigns to each possible token in the language a unique numerical value. Then, we *encode* input files into a list of numbers, which are the encoded version of the seeds. Fuzzing is then performed on this list of numbers, where changing any number to a different number is equivalent to replacing the encoded token with a different token. Whenever we want to run against the target (i.e., the JavaScript interpreter) we need to transform the mutated list of numbers back into the original language. This is done with a *decode* function that replaces each number with the corresponding token and concatenates them with spaces as needed. Thus, fuzzing can be done on the list of numbers without any knowledge of what they mean.

Of course, we need to consider that the list of valid tokens is infinite for many languages, as it includes, for example, all possible numbers and all possible variable names that are legal in the language. If the token-map contains too many numbers, then it would unnecessarily slow down the fuzzer, because most tokens would be numbers and only very few would be other functionality. To address this issue, we pick a small set of valid numbers consisting of all the powers of two (up to 2^{32}), as well as the numbers that are a power of two plus/minus one. Similar values have been chosen for other fuzzers, such as DIFUZE and AFL, to reduce the number of

Figure 2: An example of what happens to a single seed in Token-Level Fuzzing. The seed first goes through the renaming and encoding stages which produce a list of numbers. Then when running in the fuzzing loop, it is mutated and decoded prior to execution, where coverage feedback will determine if the input is added to the queue or mutated further. We highlighted how changing a couple numbers in the encoded form results in completely different tokens in the decoded result.



possible inputs [8,45]. Similarly, we found by looking through regression cases that only a small number of variables were needed to trigger most bugs, so we limited the number of variable names to fifteen.

4.2 Implementation

Our implementation of Token-Level Fuzzing is done on top of AFL, to take advantage of its coverage-guided engine. The resulting tool is called *Token-Level AFL*. Token-Level AFL is the combination of two components: a preprocessor, written in Python, that analyzes the tokens of the input files and encodes them for fuzzing, and a modified version of AFL that performs fuzzing on the encoded inputs. Figure 1 shows the overall architecture of the tool.

The preprocessor runs the following steps:

Rename: For each input seed, variable names are randomly replaced with one of the fifteen predefined variable names: var1, var2, ..., var15. Variable names are not repeated unless all fifteen variable names have been used. Within a single seed, all instances of the same variable are mapped to the same predefined name.

Renumber: As described earlier, we limit the set of valid numbers to a predefined set. All numbers are replaced with the closest number from that set.

Token Analysis: We use a JavaScript lexer to find all the tokens used in all the seeds. We assign to each token a numerical value, which will be its encoded value.

Encoding: We transform each input into a list of numbers by replacing each token with its encoded value. This list is then flattened by encoding each value as a 16-bit integer.

After the preprocessing step, which generates the token mapping and the encoded seeds, fuzzing occurs on the encoded inputs. This step is done with minor modifications to AFL:

Mutations: Mutations are slightly modified to work on an array of 16-bit numbers rather than an array of bytes. 16-bit numbers were necessary because there were more than 256 tokens. Note that this change is very small; it is effectively just changing the type of the array from `byte*` to `short*`.

Decoding: The input is decoded immediately before executing the input in the target JavaScript interpreter. This small shim simply concatenates the tokens together, adding spaces as needed[1].

Figure 2 shows an example of the various steps that Token-Level AFL applies to an input file. Although we expect that

[1] No spaces are added for certain tokens, such as quotation marks.

the input seeds are broad enough to include all valid tokens, if the seeds do not include all of the valid tokens, it is easy to add the remaining tokens by hand.

4.3 Further Mutation Modifications

Some of the mutations that AFL performs are not very useful or applicable to Token-Level Fuzzing. Specifically, these are the "arithmetics" and "interesting number" strategies. According to these strategies, AFL will try inserting interesting numbers such as "1024", "2147483647", "-100663046", etc, into the stream of bytes. Because these will get translated into a series of tokens, this just will add a constant random list of tokens into the fuzzed input. Therefore, we removed these strategies from AFL, as they do not apply well to our scenario.

Of course, the next question is whether there are strategies that we can add to improve Token-Level Fuzzing. One simple strategy that we identified is to randomly insert and overwrite multiple tokens in a row. The intuition behind this is that changing one token at a time may not be enough, and it may be necessary to change more than one token to create a new interesting input. We tested with different numbers and found that inserting and overwriting up to three tokens at a time gave good results. Therefore, we limit these to three tokens in our implementation.

We also wanted to improve AFL's ability to chain together different actions. The reason for this is that there are many bugs in interpreters that require multiple actions chained together in the correct order. Therefore, we added a mutation strategy that would copy a statement from one input to another.

We summarize the mutation strategies we added here:

Random Insert: Randomly insert new tokens somewhere into the file being mutated.

Random Overwrite: Randomly overwrite tokens in a row in the file with the same number of new tokens.

Random Replace: Randomly replace tokens in the file with new tokens. Note that this strategy can insert more or less tokens than were removed.

Statement Splice: Copy a statement from one test case to another test case. This mutation strategy assumes that statements start and end at semicolons; the strategy then replaces all the tokens between two semicolons.

5 Evaluation

To evaluate our implementation of Token-Level Fuzzing, we run the fuzzer on the JavaScript interpreters from the four major browsers, namely, V8, SpiderMonkey, JavaScriptCore, and ChakraCore[2]. Our goal is to understand the bug-finding capabilities of Token-Level Fuzzing as well as how our implementation compares to other state-of-the-art JavaScript interpreter fuzzers. In order to reason about these goals, we answer the following research questions:

RQ1: DoesToken-Level Fuzzing generate more syntactically correct inputs than byte-level fuzzing?

RQ2: How does Token-Level Fuzzing compare to other state-of-the-art fuzzers?

RQ3: Is Token-Level Fuzzing able to find real-world vulnerabilities in the latest JavaScript interpreters?

RQ4: Do bugs found by Token-Level Fuzzing involve incorrect syntax/semantics?

5.1 Experiment Setup

We started by downloading the latest available versions of the four major JavaScript interpreters as of October 1, 2019. These were the development versions cloned from the official git repositories. We compiled all interpreters with debug checks. Debug checks are additional checks that the programmers include to try to catch unexpected conditions [39]; therefore, we enabled them to catch more potential security bugs. We did not enable Address Sanitizer or other sanitizers, as these tended to be too slow in our tests.

Seed Collection. Having good seeds is essential for our fuzzer, for multiple reasons. First, the list of potential tokens that will be used by our fuzzer are taken from the set of input files. Thus, it is essential that the seeds cover as many of the tokens used by the language as possible. Second, our implementation of Token-Level Fuzzing is based on AFL and evolutionary fuzzing, so having a quality set of diverse seeds helps the fuzzer greatly, because it will explore starting from these initial seeds. To collect seeds, we manually selected regression tests from the repositories of the various JavaScript interpreters. We manually picked seeds covering a wide range of functionality, but limited the number of seeds to one hundred.

Comparison with other tools. We compared Token-Level AFL against the following state-of-the-art tools: AFL [45], Fuzzilli [20], CodeAlchemist [23], and Superion [43]. To this end, we ran each tool for three days on 30 cores, on each of the four major JavaScript interpreters, resulting in a total of 2,160 core-hours for each fuzzing run. We then repeated each fuzzing run (that is, each fuzzer-JavaScript interpreter combination) five times, to limit randomness in our experiments. Note that Fuzzilli does not provide a mechanism for using seeds, so it was run without seeds. On the other

[2]ChakraCore is no longer used in Edge as of January 2020 [10].

hand, the authors of CodeAlchemist used far more seeds in their paper [23], and, therefore, to fairly evaluate this tool, we created a much larger seed collection, which included all JavaScript files from the regression tests, resulting in 32,682 seeds. This larger set of seeds was only used when testing CodeAlchemist.

Note that when comparing against these tools there may be a bias in terms of number of bugs found. This is because other published tools may have already reported the bugs that they were able to find, and these bugs might have been already fixed in the JavaScript engines that we analyzed. However, running experiments on the latest JavaScript interpreters will let us know if Token-Level AFL finds different bugs than the other tools.

5.2 Syntactically Valid Inputs

The most basic assumption of Token-Level Fuzzing is that it generates more syntactically correct inputs than byte-level fuzzing, and that these inputs will, in turn, trigger deeper functionality. To assess the validity of this assumption, we first compare the results of AFL and Token-Level AFL. Both fuzzers were given the same seeds, and AFL was given all of the tokens in the input files as a dictionary. With a dictionary, AFL will try inserting the keywords in the mutation steps. This allows AFL to make some progress on languages such as JavaScript, and showcases the best configuration for AFL [30]. In our experiments, even with a full dictionary and the same input seeds, AFL was only able to find 2 bugs across all the JavaScript interpreters, whereas Token-Level AFL found 19. Furthermore, as shown in Table 4, both bugs reported by AFL were also found by our tool.

Next, we added tracking to the V8 JavaScript engine to determine how many of the inputs, generated by the two fuzzers, were parsed successfully or led to parser errors. These numbers are shown in Table 1. Only 10.7% of all the inputs AFL produced could be parsed successfully. This shows that, as we suspected in Section 3, most inputs generated by AFL fail to parse, and do not trigger any reasonable functionality in the JavaScript interpreters. The improvement provided by Token-Level Fuzzing is immediately evident; 29.98% of all inputs generated by Token-Level AFL were successfully parsed. The higher fraction of successfully parsed inputs allows the fuzzer to generate more inputs that trigger useful functionality. This, in turn, allows the fuzzer to find deeper bugs and explore more of the JavaScript interpreter's functionality.

Answer for RQ1: The results show that Token-Level Fuzzing generates syntactically correct inputs about three times more often than byte-level fuzzing, enabling more efficient fuzzing of interpreters.

Table 1: This table shows what fraction of inputs generated by AFL and Token-Level AFL are able to be parsed successfully when fuzzing V8. The higher parse rate of Token-Level AFL shows that by mutating tokens instead of bytes, our technique is able to generate more correct inputs.

Fuzzer	Successful Parse Rates
AFL	10.70%
Token-Level AFL	29.98%

5.3 Comparison with other State-of-the-Art Fuzzers

In this section, we will explore how Token-Level AFL performs when compared against other state-of-the-art JavaScript interpreter fuzzers. For this comparison, we selected AFL, Fuzzilli [20], CodeAlchemist [23], and Superion [43]. These comparison tools were chosen because of their impressive results and their varying techniques.

As mentioned previously, we evaluated all of these fuzzers on the latest available JavaScript interpreters, which were retrieved from the official repositories on Oct 1, 2019. Each fuzzer was run on 30 cores for three days on each of the four JavaScript interpreters. Moreover, each test run was repeated five times to reduce randomness.

As is usual for fuzzing research, we use the number of bugs found as the main performance metric. For the purpose of this analysis, we consider any debug check, release check, or memory corruption to be a bug. Although debug checks may not always indicate that a security issue was found, they do indicate that an assumption was violated, and they show that a fuzzer is finding bugs that have not been previously found. To identify unique bugs, we filter the tool's reports based on any asserts hit, as well as using manual analysis to ensure that only unique issues are counted.

Additionally, we investigate block coverage during this evaluation. Although block coverage may not be as meaningful a measurement as the number of bugs found, it still shows useful information [25, 37]. To be able to trigger a bug, a fuzzer must be able to reach the code where the bug is located. So, coverage is a necessary, but not sufficient, condition for finding bugs and can be used as a performance metric. We collected block coverage information throughout each of the fuzzing runs.

Results: As shown in Figure 3, Token-Level AFL found the most crashes during the three-day fuzzing periods. More precisely, Token-Level AFL found 19 total bugs across the five runs, while the second best performer, CodeAlchemist, found 8 bugs. Additionally, only 4 of the 19 bugs found by Token-Level AFL were found by any other tool (see Table 4.) Each of the other 15 bugs were unique to Token-Level AFL. Also, although CodeAlchemist found seven bugs in ChakraCore, only two of those bugs overlapped with the four found by Token-Level AFL. This indicates that our method finds bugs

Figure 3: This graph shows the total number of unique bugs found by each of the tested fuzzers when run on the four major JavaScript interpreters for a time period of 72 hours. This graph shows the aggregate number of bugs across all five runs, and only unique bugs are counted.

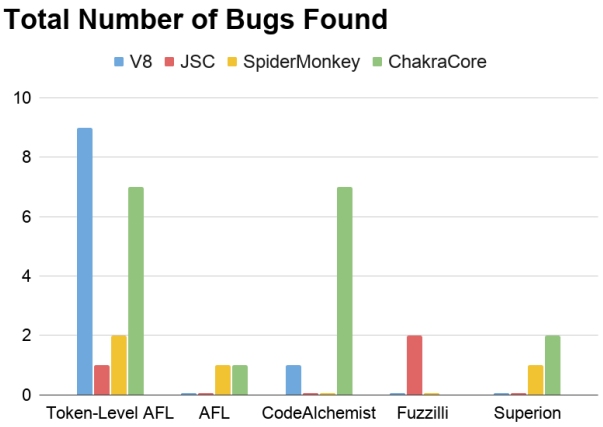


Table 2: Average number of bugs found by each of the tested fuzzers on the four major JavaScript interpreters in a single run. The 95% confidence interval is ±1.39 for each entry. (Fuzzilli does not have code for running on ChakraCore, so that table entry is omitted).

	V8	JSC	Spider-Monkey	Chakra
Token-Level AFL	5.2	0.6	0.8	3.2
AFL	0	0	1.0	0.2
CodeAlchemist	0.6	0	0	4.0
Fuzzilli	0	1.0	0	N/A
Superion	0	0	1	0.4

that other fuzzers are not able to find. Furthermore, in Table 2 we show the average number of bugs each tool found in any single run. This data shows that Token-Level AFL also finds more bugs in each run than other tools.

When investigating coverage, we found that, on average, Token-Level AFL covered more blocks in the JavaScript interpreters than three of the other tools; only CodeAlchemist triggered more basic blocks. The average number of basic blocks found in each configuration is shown in Table 3, and a graph of block coverage over the three days of fuzzing is shown in Figure 4. When investigating these numbers in more detail, we discovered that the seeds may play a large role in CodeAlchemist's superior code coverage. In particular, the 32,682 seeds given to CodeAlchemist alone triggered about 160,000 blocks in V8, whereas the 100 seeds given to Token-Level AFL only covered about 94,000 blocks. However, even with higher coverage, CodeAlchemist triggered fewer bugs, showing that code coverage does not yield bugs on its own; assumptions must be violated as well. Also, Token-Level AFL

Figure 4: This graph shows the block coverage over time for each of the fuzzers when running on V8. Token-Level AFL was able to continually find and trigger new blocks throughout the three-day experiment.

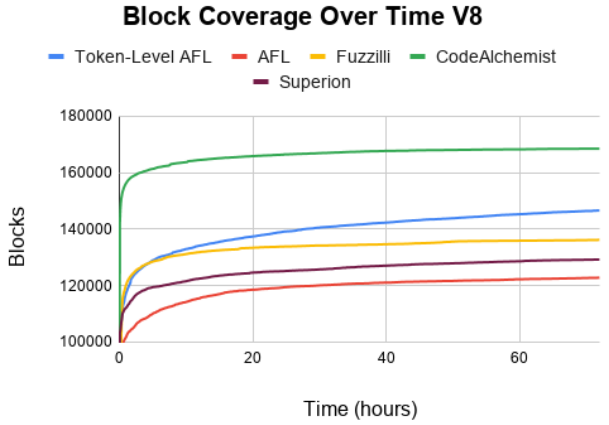


Table 3: Average number of basic blocks triggered by each of the tools on each of the target interpreters. Token-Level AFL performed similarly to Fuzzilli in terms of number of blocks covered. CodeAlchemist, which used many more seeds, had the best block coverage.

	V8	JSC	Spider-Monkey	Chakra-Core
Token-Level AFL	146,625	246,720	172,050	178,126
AFL	122,833	219,774	157,031	139,070
CodeAlchemist	168,512	256,650	212,267	214,499
Fuzzilli	136,218	244,391	184,363	N/A
Superion	129,753	223,656	165,674	169,440

was able to find many blocks that were not triggered by the initial seeds. Finally, the graph shows that Token-Level AFL was still finding new basic blocks at the end of the fuzzing period, whereas the other tools had plateaued.

It is worth mentioning that the lack of bugs found by other tools does not necessarily indicate a lack of performance. Instead, it is quite likely that, because these fuzzers are open-source, they are currently being run and bugs that they find are reported and fixed frequently. However, our results do show that Token-Level AFL is finding new, different bugs that these other tools are not able to find as easily.

Breakdown: Table 4 shows the overlap of bugs found during the comparison experiments. In V8 and Spidermonkey, Token-Level AFL was the only fuzzer to find unique bugs that no other fuzzer found. However, in both JSC and Chakra there were some bugs that we missed and that were found only by a different fuzzer.

We performed deeper analysis to understand if there are similarities among the bugs that only Token-Level AFL finds,

Table 4: This table shows a breakdown of which fuzzers found each of the 27 unique bugs during the comparison experiment on the October 2019 versions. This shows cumulative results across all 5 runs.

		Token-Level AFL	AFL	Code-Alchemist	Fuzzilli	Superion
V8	Bug 1	X				
	Bug 2	X				
	Bug 3	X		X		
	Bug 4	X				
	Bug 5	X				
	Bug 6	X				
	Bug 7	X				
	Bug 8	X				
	Bug 9	X				
Spider-monkey	Bug 1	X	X			X
	Bug 2	X				
JSC	Bug 1				X	
	Bug 2	X				
	Bug 3				X	
Chakra	Bug 1	X	X	X		X
	Bug 2	X		X		
	Bug 3					X
	Bug 4			X		
	Bug 5			X		
	Bug 6			X		
	Bug 7			X		
	Bug 8			X		
	Bug 9	X				
	Bug 10	X				
	Bug 11	X				
	Bug 12	X				
	Bug 13	X				

as well as those bugs that our system missed. For many of Token-Level AFL's unique crashes, we found that the inputs used to trigger these bugs include code patterns with uncommon or completely invalid syntax. This underlines the value of a fuzzer that can generate inputs that do not strictly follow a grammar. We provide a more detailed case study for some of these crashes in Section 5.5.

For the bugs that Token-Level AFL did not find, we did not find any obvious shared characteristics. In fact, it appeared to us that it would be possible to trigger these crashes with different seeds or better luck. The bugs were triggered by specific sequences of (valid) operations, and Token-Level AFL had not (yet) produced the required order.

Answer for RQ2: Token-Level AFL is able to find bugs that other state-of-the-art fuzzers are unable to find. Furthermore, in our tests, Token-Level AFL found more bugs in the major JavaScript interpreters than any of the other state-of-the-art fuzzers.

5.4 Real-World Bugs

In the previous section, we have shown that Token-Level AFL is effective in finding bugs in JavaScript interpreters that other fuzzers are unable to find. These bugs were in the JavaScript interpreters that were available as of October 1, 2019. In a separate experiment, we wanted to further explore Token-Level AFL's ability to find bugs when run over a longer period of time (rather than the three days used for the comparative evaluation). To do this, we let our fuzzer run for 60 days. We started with the interpreters as of September 20, 2019. Over the duration of the following two months, we periodically restarted the fuzzer and updated the JavaScript interpreters as new versions became available.

Table 5 shows a summary of all the bugs that Token-Level AFL found across the analyzed JavaScript interpreters. The table shows in which interpreter each bug was found and a description of the bug. The status column shows if the bug has been reported by us and fixed. "Confirmed" indicates that we have confirmed the bug in the latest version. "Fixed internally" means that the interpreter developers identified and fixed the bug without our report (i.e., after we found the bug, but before we had a chance to report it); sometimes these were short-lived bugs. Note that the 19 bugs found during our comparative evaluation by Token-Level AFL were also found during this experiment, and they are included in Table 5. Thus, Token-Level AFL identified 10 additional bugs when given more time.

Our fuzzer found the 29 bugs across many areas of the JavaScript interpreters: from the parser, to the handler of regular expressions, to the JIT compiler. We believe that this shows not only that Token-Level AFL is capable of finding unknown bugs in JavaScript interpreters, but also that it is widely applicable and can find bugs in many components of the interpreter.

Also, these bugs include some that could lead to remote code execution. We were able to write an RCE exploit for Chrome using bugs that we found with this tool. Furthermore, we have been awarded over ten thousand dollars in bounties, showing the impact of our research.

Answer for RQ3: Token-Level AFL is able to find real-world bugs in all of the major JavaScript interpreters. This shows that Token-Level AFL has impact and can be used for finding previously unknown bugs as well as for catching bugs as they are introduced.

5.5 Case Study

In this section, we investigate some of the bugs to determine if Token-Level AFL is finding bugs that involve invalid syntax, which strict grammar based tools may be unable to find.

In Listing 4, we show (a minimized) example of the JavaScript code that triggers a bug that Token-Level AFL found. This is a bug in V8 that leads to memory corruption. It requires a syntax error to trigger and was introduced when new parser code was added that allowed for certain incorrect syntax patterns, such as the one that is shown in the listing.

Table 5: This table shows the bugs which Token-Level AFL found in the analyzed JavaScript interpreters over a 60-day period. Some of these bugs resulted in memory corruption, which could lead to exploitation and remote code execution. In the "Status" column we note if we have confirmed that the bug still exists in the most up-to-date code, reported it, or if it was fixed internally. We are currently in the process of responsibly disclosing all confirmed bugs to the respective software vendors.

Bug Number	JS Interpreter	Description	Status	Bug ID
1	V8	Memory corruption while parsing	Reported/Fixed	CR 1015567
2	V8	Debug Check due to incorrect parsing of arrow functions.	Reported/Fixed	V8 9758
3	V8	Null dereference	Fixed Internally	
4	V8	Debug Check in regular expression runtime	Reported/Fixed	CR 1018592
5	V8	Out of bounds indexing in an array due to incorrect parsing	Reported/Fixed	CR 1021457
6	V8	Parser debug check due to incorrectly allocated variable	Fixed Internally	
7	V8	Debug Check in garbage collection	Reported	CR 1044261
8	V8	Debug check when converting integer to index	Fixed Internally	
9	V8	Triggers unreachable code due to frozen elements	Reported/Fixed	CR 1045572
10	V8	Unexpected error handler triggered in JIT	Fixed Internally	
11	V8	Check failed due to incorrect object size	Reported/Fixed	CR 1076106
12	V8	JIT bug leading to memory corruption	Reported/Fixed	CVE-2020-6468
13	V8	Triggers unreachable code due to frozen elements	Reported/Fixed	V8 10484
14	V8	Jit bug in bytecode analysis	Fixed internally	
15	V8	Parser error leading to debug check	Confirmed in latest	
16	V8	JIT assertion related to a syntax error	Confirmed in latest	
17	JSC	JIT bug resulting in an unexpected switch case	Reported	Webkit 221069
18	JSC	JIT bug in FTL resulting in an unexpected null pointer	Fixed internally	
19	JSC	JIT bug in DFG failing a validation check	Confirmed in latest	
20	JSC	JIT bug in FTL to DFG Lowering	Fixed Internally	
21	SpiderMonkey	Length related assertion	Reported	1669616
22	SpiderMonkey	Parser assertion	Confirmed in latest	
23	SpiderMonkey	Parser bug leading to leaked magic value	Fixed	
24	ChakraCore	Type mismatch in parsing	Reported	MS 041681
25	ChakraCore	Unexpected case in the JIT	Reported	MS 041671
26	ChakraCore	Array length changed where it should not have changed	Reported	MS 041673
27	ChakraCore	Out of Bounds in Array runction	Reported	MS 041679
28	ChakraCore	Assertion setting a field on an object	Reported	MS 041678
29	ChakraCore	Assertion in set accessor	Reported	MS 041676

```
class var6 extends Object {
  constructor ( a,b,c) {
    super (1.1 ) 1 ;
  }
};

new var6();
```

Listing 4: Code which triggers a bug found by Token-Level AFL in V8. This bug contains a syntax error due to the number 1 after the call to `super(1.1)`. In this case, the parser would incorrectly calculate the index into an array, resulting in exploitable memory corruption.

```
function f () {
  var14=[1,2,3,4,5,6,7,8];
  var15=var14;
  var14.length = 0x100 ;
  var14.__defineGetter__(/./, function(){
    var14.unshift ( 0x20 ) ;
    var14.shift();
    var var3=new Uint32Array(var14);
    Object.entries(var14).toString();
  } ) ;
  print(Object.entries(var14).toString());
}
f();
```

Listing 5: This minimized test case triggers a debug check found in V8. This bug is caused by repeated shifting and unshifting of an array, which can trigger a debug check in the garbage collector.

Our tool was able to find this syntax, partially because of its evolutionary behavior. The bug was fixed due to our report and a bounty was awarded.

Listing 5 shows another example of code that triggers a bug. This bug results in a Debug Check in V8's garbage collector. The code shown is a minimized version of the real test case, after removing redundant statements. This bug is more complex than the previous example, and requires many valid JavaScript statements. We attribute the ability of Token-Level AFL to produce complex valid test cases to its coverage-guided capabilities, which will tend to discard test cases that do not hit new functionality, allowing it to explore deep code paths.

Bugs found by our technique included both examples where incorrect syntax or semantics is used to trigger a bug and examples where no such error exists in the test case. Also, many of the bugs that we found were in the parser, as opposed to the other tools we tested, which tend to miss those bugs. Our results show that Token-Level AFL is applicable to finding bugs both in the parser and elsewhere in the JavaScript interpreter.

> **Answer for RQ4**: Bugs found by Token-Level AFL include examples where both entirely valid syntax is used and examples where invalid syntax is needed.

6 Discussion

Token-Level Fuzzing is a promising new technique that enables deep fuzzing of JavaScript interpreters, without some of the limitations that come with grammar-based fuzzers. By performing coverage-guided mutations on tokens, rather than individual bytes, it can easily mutate the highly structured inputs involved in the language. Additionally, because Token-Level Fuzzing is able to find bugs with unusual constructs (syntax and semantics), we believe it will complement the current grammar-based approaches nicely. In this section, we will discuss the generalizability of our technique, as well as directions for future work.

6.1 Generalizability

Although we implemented and tested Token-Level AFL only on JavaScript interpreters, the technique is likely applicable to other programs that process inputs formatted in well-defined languages, such as compilers and configuration parsers. The tool would need a different pre-processor, specific to the target, that can separate the text into tokens and identify variables. Similarly, a new decoder would need to be written for that target to transform the encoded input back into the original language. These are not technical challenges, and we believe this technique should be effective on other token-based programs, especially given the results it has shown on JavaScript interpreters. Furthermore, this is likely easier than adapting a grammar-based fuzzer to a new target.

6.2 Seed Selection

Token-Level AFL relies heavily on the input seeds, and it is intuitive that this selection can matter greatly. If a seed is close to triggering a bug, then the number of mutations needed to exercise the bug may be small. In fact, we noticed substantial similarities between some of the bugs that we found and the input test cases that we provided. Additionally, having seeds that trigger a wide variety of functionality helps the fuzzer to explore the various areas of the interpreter's code.

One result shown in Section 5 is that Token-Level AFL's block coverage could likely be improved by having a better, larger set of seeds. For our experiments, we used a fairly ad hoc approach for our seed collection, and applying a better and more systematic methods might yield even better results. For example, Skyfire [42] could be used to generate promising JavaScript seeds. There are also various papers suggesting

better seed selection strategies, which we could employ to improve our results [7, 35].

6.3 Future Work

Because our technique transforms the JavaScript tokens into the familiar binary-based format, we could leverage recent advancements that have been made in the fuzzing field. For example, because there are so many edges in the JavaScript interpreters, we find that there are many collisions in the edge tracking of AFL. We could use the path sensitivity of CollAFL [12] to help remedy this. Applying ensemble based fuzzing [6], by using Token-Level AFL alongside a grammar-based approach, could allow both techniques to build on top of their results. Another direction would be to try to use better prioritization on the inputs, as suggested by Wang, et al. [44], especially since we typically have tens of thousands of inputs in the fuzzer queue after a few days of fuzzing.

7 Conclusion

In this paper, we have presented Token-Level Fuzzing, a new technique for fuzzing language-based programs, such as interpreters. Token-Level Fuzzing allows one to fuzz these complex programs without the need of a grammar, allowing it to exercise both the parsing layers, as well as the actual interpretation. This relatively simple idea (one can fuzz at an intermediate level between grammar-based and byte-based fuzzers) provides security researchers with a powerful new technique that can be built upon for further research.

In our evaluation, Token-Level AFL found 29 new bugs across the most up-to-date JavaScript interpreters, several of which were high-severity issues. Given the difficulty of fuzzing such programs, we believe that these results showcase the potential of our technique.

References

[1] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars." in *NDSS*, 2019.

[2] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[3] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[4] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 725–741.

[5] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[6] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1967–1983.

[7] L. Cheng, Y. Zhang, Y. Zhang, C. Wu, Z. Li, Y. Fu, and H. Li, "Optimizing seed inputs in fuzzing with machine learning," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 244–245.

[8] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[9] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Vulnerability Scanner," in *Proceedings of the USENIX Security Symposium*, Aug. 2012.

[10] Engadget, "Microsoft's chromium edge browser arrives january 15th," 2019, https://www.engadget.com/2019-11-04-chromium-edge-browser-release-date.html.

[11] I. Fratric, "The great dom fuzz-off of 2017," 2017.

[12] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[13] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 206–215.

[14] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20–27, 2012.

[15] Google, https://google.github.io/oss-fuzz/getting-started/continuous-integration.

[16] ——, https://bugs.chromium.org/p/project-zero/issues/detail?id=1308.

[17] ——, "syzkaller - linux syscall fuzzer," 2017, https://github.com/google/syzkaller.

[18] "Issue 800032: Security: V8: Bugs in genesis::initializeglobal," Google, 2018, https://bugs.chromium.org/p/chromium/issues/detail?id=800032.

[19] G. Grieco, M. Ceresa, and P. Buiras, "Quickfuzz: An automatic random fuzzer for common file formats," *ACM SIGPLAN Notices*, vol. 51, no. 12, pp. 13–20, 2016.

[20] S. Groß, "Fuzzil: Coverage guided fuzzing for javascript engines," Ph.D. dissertation, TU Braunschweig, 2018.

[21] R. Guo, "Mongodb's javascript fuzzer," *Queue*, vol. 15, no. 1, pp. 38–56, 2017.

[22] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2345–2358.

[23] H. Han, D. Oh, and S. K. Cha, "Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines." in *NDSS*, 2019.

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

[25] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2123–2138.

[26] A. LEE, "Fuzzing javasscript engines for fun and pwnage," 2018.

[27] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided javascript engine fuzzer," *arXiv preprint arXiv:2001.04107*, 2020.

[28] https://llvm.org/docs/LibFuzzer.html, llvm, 2019.

[29] G. McGraw, S. Migues, and J. West, "Bsimm8," 2017, https://www.bsimm.com/content/dam/bsimm/reports/bsimm8.pdf.

[30] Michal Zalewski, "afl-fuzz: making up grammar with a dictionary in hand," 2015, https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html.

[31] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1629–1642.

[32] "Peach," Peach Tech, https://www.peach.tech/.

[33] J. Pereyda, "boofuzz," https://github.com/jtpereyda/boofuzz.

[34] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the 2017 Network and Distributed System Security Symposium*, 2017.

[35] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 861–875. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671280

[36] J. Ruderman, "Introducing jsfunfuzz," 2007, https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

[37] C. Salls, A. Machiry, A. Doupe, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Exploring abstraction functions in fuzzing," in *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2020, pp. 1–9.

[38] 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 *Proceedings of the 2016 Network and Distributed System Security Symposium*, 2016.

[39] The Chromium Project, https://chromium.googlesource.com/chromium/src/+/master/styleguide/c++/c++.md#CHECK_DCHECK_and-NOTREACHED.

[40] G. Vranken, "libfuzzer-gv: new techniques for dramatically faster fuzzing," https://guidovranken.wordpress.com/2017/07/08/libfuzzer-gv-new-techniques-for-dramatically-faster-fuzzing/, 2017.

[41] D. Wang, X. Zhang, T. Chen, and J. Li, "Discovering vulnerabilities in cots iot devices through blackbox fuzzing web management interface," *Security and Communication Networks*, vol. 2019, 2019.

[42] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 579–594.

[43] ——, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[44] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization," in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2020.

[45] M. Zalewski., "American fuzzy lop," 2017, http://lcamtuf.coredump.cx/afl/technical_details.txt.

[46] ——, "American fuzzy lop," 2017, http://lcamtuf.coredump.cx/afl/.

[47] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "Greybox fuzzing," 2019, https://www.fuzzingbook.org/html/GreyboxFuzzer.html.