Proceedings Paper:
Ye, G, Tang, Z, Tan, SH et al. (6 more authors) (2021) Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing. In: PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 20-25 Jun 2021, Online. Association for Computing Machinery (ACM) , pp. 435-450. ISBN 978-1-4503-8391-2

https://doi.org/10.1145/3453483.3454054

Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing

Guixin Ye
gxye@nwu.edu.cn
Northwest University
Xi'an, China

Zhanyong Tang
zytang@nwu.edu.cn
Northwest University
Xi'an, China

Shin Hwei Tan
tansh3@sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

Songfang Huang
songfang.hsf@alibaba-inc.com
Alibaba DAMO Academy
Beijing, China

Dingyi Fang
dyf@nwu.edu.cn
Northwest University
Xi'an, China

Xiaoyang Sun
scxs@leeds.ac.uk
University of Leeds
Leeds, United Kingdom

Lizhong Bian
Alipay (Hangzhou) Information & Technology Co., Ltd.
Hangzhou, China

Haibo Wang
schw@leeds.ac.uk
University of Leeds
Leeds, United Kingdom

Zheng Wang
z.wang5@leeds.ac.uk
University of Leeds
Leeds, United Kingdom

Abstract

JavaScript (JS) is a popular, platform-independent programming language. To ensure the interoperability of JS programs across different platforms, the implementation of a JS engine should conform to the ECMAScript standard. However, doing so is challenging as there are many subtle definitions of API behaviors, and the definitions keep evolving.

We present Comfort, a new compiler fuzzing framework for detecting JS engine bugs and behaviors that deviate from the ECMAScript standard. Comfort leverages the recent advance in deep learning-based language models to automatically generate JS test code. As a departure from prior fuzzers, Comfort utilizes the well-structured ECMAScript specifications to automatically generate test data along with the test programs to expose bugs that could be overlooked by the developers or manually written test cases. Comfort then applies differential testing methodologies on the generated test cases to expose standard conformance bugs. We apply Comfort to ten mainstream JS engines. In 200 hours of automated concurrent testing runs, we discover bugs in all tested JS engines. We had identified 158 unique JS engine bugs, of which 129 have been verified, and 115 have already been fixed by the developers. Furthermore, 21 of the Comfort-generated test cases have been added to Test262, the official ECMAScript conformance test suite.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Language features**; • **Computing methodologies** → *Artificial intelligence.*

Corresponding authors: Zhanyong Tang, Songfang Huang and Zheng Wang.

PLDI '21, June 20–25, 2021, Virtual, Canada
2021. ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454054

Keywords: JavaScript, Conformance bugs, Compiler fuzzing, Differential testing, Deep learning

ACM Reference Format:
Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454054

1 Introduction

JavaScript (JS) is one of the most popular programming languages [2]. It is a core technology that underpins the web browser, server-side web deployments and many embedded and smartphone applications. The implementation of a JS engine (compiler) should conform to the ECMAScript specification, ECMA-262 [1], that ensures the interoperability of JS code across different JS platforms. Non-standard JS engine implementations can confuse the application developer, leading to unexpected software behavior and poor user experience during deployment. Still, writing a JS engine that conforms to ECMAScript is hard due to the complexity of modern JS interpreters, the large number of JS APIs[1] and object types, the constantly evolving language specification, and the diversity of JS code and inputs seen in real-life deployments. Hand-written JS test cases for testing JS engines, while important, are inadequate for covering all parts of the JS language standard to test a sophisticated JS engine.

Random test case generation - or *fuzzing* - is a widely used technique for automated compiler bug detection [11, 13, 39]. It is often used together with differential testing [31, 45, 54]

[1]The ECMA-262 2020 Edition defines over 750 JS APIs.

to discover unexpected program behavior. In the context of fuzzing JS compilers, a randomly generated JS program and its input forms a *test case*, which is executed by *multiple* JS engines. Any unexpected behavior, including crashing, freezing or inconsistent compliation or execution outcomes among engines, indicates a JS compiler bug.

The success of compiler fuzzing requires generating bug-exposing test cases with the right program and test input to trigger buggy behavior [12]. Unfortunately, doing so is challenging as there are many ways to construct a program and its input. Existing fuzzing techniques for JS typically follow a generative or mutational approach. Generative approaches build a new test case from the ground up using predefined grammar rules [27, 65] or by reassembling synthesizable code segments from a program corpus [24, 49]. By contrast, mutational approaches synthesize a test case from existing seed programs and inputs [61]. Both strategies require expert involvement to construct the grammar rules or preparing a high-quality seed code corpus to ensure the coverage of the test cases, but doing so becomes increasingly difficult due to the complexity and constantly evolving nature of the JS language standard.

We present Comfort[2], a *generative* fuzzer for JS engines. Unlike existing JS fuzzers that aim to detect crashing bugs or vulnerabilities [49], Comfort focuses on exposing standard conformance bugs. Comfort leverages the advances in deep-learning (DL) based program synthesis [14] to generate JS programs by automatically learning a generation model. Specifically, Comfort employs GPT-2 [50], a recently proposed language generation model, to generate JS code by learning from a corpus of open-source JS programs. GPT-2 improves the long short-term memory (LSTM) model used in state-of-the-art DL-based fuzzers [14, 33, 37] by generating valid JS programs with a higher success rate, as it can model longer dependencies in the program source code. Comfort then uses differential testing to detect buggy JS engine behavior.

Existing compiler fuzzers use a random input generation strategy by relying on the typing information of a variable [14, 31] to generate kernel or function parameters. However, JS is a weakly typed language, where a variable can be of an arbitrary type and can be of multiple types throughout the execution. This feature increases the space of possible input settings, making it harder for a random input generation strategy to trigger compiler bugs with reasonable cost. To effectively generate test program inputs, Comfort draws hints from ECMA-262. It leverages the well-structured specification rules defined in the specification document to narrow down the scope of argument types and boundary values for JS APIs and edge cases that are likely to trigger unusual behavior. By narrowing down the scope, Comfort is able to generate inputs that are more likely to cover the special cases overlooked by JS engine developers and manually written test cases. The result is a new way of leveraging the language specification for compiler fuzzing.

We evaluate Comfort[3] by applying it to ten mainstream JS engines. The JS engines that we target include those used in mainstream web browsers: JavaScriptCore (JSC) in Apple Safari, V8 in Google Chrome, ChakraCore in Microsoft Edge, SpiderMonkey in Firefox, JS engines for mobile and embedded systems: Hermes, QuickJS, Rhino, Nashorn, and JerryScript, and Graaljs that is specifically designed to be compatible with ECMAScript 2020. Our large-scale evaluation shows that Comfort is highly effective in generating syntactically correct JS programs with a better test coverage, where 80% of the generated code is syntactically correct. This success rate translates to 2.6× improvement over the success rate given by DeepSmith [14], a state-of-the-art DL-based generative fuzzer. We show that Comfort is more efficient in producing bug-exposing test cases by uncovering at least 2× more unique bugs within the same test running time, when compared to state-of-the-art JS fuzzers [22, 24, 33, 49]. In 200 hours of automated concurrent testing runs, Comfort discovers bugs in all tested JS engines. We have identified 158 unique JS compiler bugs, covering 109 newly discovered bugs. Of the submitted bugs, 129 have been verified and 115 have been fixed by the relevant JS compiler developers. Moreover, 21 of the test cases produced by Comfort has been added to Test262 [4], the official ECMAScript conformance test suite.

This paper shares our experience and findings of exploiting ECMA-262 to detect JS compiler bugs through fuzzing. It makes the following contributions:

- It is among the first studies on employing compiler fuzzing to expose conformance bugs in JS compilers;
- It is the first random program generator for leveraging the language specification document to generate test data for compiler fuzzing (Section 3.3);
- It provides a large study independently validating the effectiveness of the recently proposed DL-based test program generation method [14] in a new domain (Section 3.2).

2 Background and Motivation

2.1 JavaScript Standard

ECMA-262[4] is the standardized specification for JavaScript. The first edition of ECMA-262 was developed in 1997, and the 11th edition, officially known as ECMAScript 2020, was

[2] Comfort = **COM**piler **F**uzzing f**O**r javasc**R**ip**T**

[3] Code and data are available at: https://github.com/NWU-NISL-Fuzzing/COMFORT.

[4] Another ECMAScript document called ECMA-402 provides additional definitions for internationalization APIs for, e.g., the numbering system, calendar, and formats, used by different human languages and countries. We do not target ECMA-402 in this work.

String.prototype.substr(***start*, *length*** **)**

…

2. Let *S* be ToString(*O*).
3. ReturnIfAbrupt(*S*).
4. Let *intStart* be ToInteger(*start*).
5. ReturnIfAbrupt(*intStart*).
6. If *length* is **undefined**, let *end* be +∞; else let *end* be ToInteger(*length*).
7. ReturnIfAbrupt(*end*).
8. Let *size* be the number of code units in *S*.
9. If *intStart* < 0, let *intStart* be max(*size* + *intStart*, 0).
10. Let *resutLength* be min(max(*end*, 0), *size* - *intStart*).
11. If *resutLength* ≤ 0, return the empty String " ".
12. Return a String containing *resutLength* consecutive code units from …

Figure 1. Expected behavior of the `substr` function defined in the ECMAScript 2015 specification.

published in June 2020. ECMA-262 is a well-structured document that uses a mixture of natural language descriptions and pseudo-code to describe the expected observed behavior for JS APIs. Figure 1 shows an example pseudo-code for the `substr` function defined in ECMA-262.

Test262 [4] is an ECMA-262 conformance test suite for checking how closely a JS implementation follows the ECMAScript specification. As of August 2020, this test suite has 140 contributors, containing over 38,000 test cases - each of which tests some specific ECMA-262 specification rules.

2.2 Problem Scope

We now define the notion of the *conformance bug* for the JS engine, and then introduce the concept of the *conformance testing* before scoping our work.

Definition 2.1 (Conformance bug)**.** Given implementations of multiple JS engines $\mathbb{J}$ and a version of the ECMA-262 specification $\mathbb{E}$, a *conformance bug* is an unexpected behavior in $\mathbb{J}$ that occurs due to a violation to the specification in $\mathbb{E}$.

Definition 2.2 (Conformance testing)**.** Given an implementation τ_i of a JS engine and an ECMA-262 specification $\mathbb{E}$, *conformance testing* is a technique that aims to identify if τ_i meets the requirements of $\mathbb{E}$. Conformance testing is used to discover conformance bugs.

Prior work on JS compiler fuzzing is primarily concerned about detecting crashing bugs or vulnerabilities [22, 24, 33, 49], but detecting engine conformance using fuzzing is largely overlooked. Comfort is designed to expose JS compiler bugs, including conformance bugs. We remark that Comfort is not designed to automate bug confirmation. Instead, it will report the potential buggy behavior to the JS engine developers to ask for developer confirmations. In this work, when testing a JS engine, we restrict our scope to the ECMA-262 edition that the targeting engine version claims to be compatible with. Therefore, we remove test programs that contain unsupported JS APIs in our test dataset when fuzzing a JS engine. We also manually inspected the failed test cases of a compiler version and only reported bugs if the feature is supported by the relevant ECMA-262 version.

```
1  function foo(str, start, len) {
2      var ret = str.substr(start, len);
3      return ret;
4  }
5  var s = "Name:␣Albert";
6  var pre = "Name:␣";
7  var len = undefined;
8  var name = foo(s, pre.length, len);
9  print(name);
```

Figure 2. A test case generated by Comfort, which exposes a conformance bug of Rhino.

2.3 Using JS Specifications for Fuzzing

To demonstrate how ECMA-262 can help in identifying conformance bugs, consider the JS example shown in Figure 2.

This code uses the `substr` API (line 2) to extract a substring of `str`, starting at the position defined by the `start` argument and extending for a given number (defined by `len`) of characters afterwards. According to the ECMA-262 rules in Figure 1 (line 6), if the parameter `length` (`len` in Figure 2) is *undefined*, the function should extract the substring from `start` to the end of the string. For this example program, it should print "Albert" as an output. However, the latest version of the Rhino JS engine produces an empty string, which is thus a standard conformance bug. To generate this bug-exposing input, the fuzzer needs to be aware of the context, i.e., it needs to produce a String object and ensure the `len` variable (line 7 in Figure 1) is *undefined* before passing to `substr`. Existing compiler fuzzers would struggle to generate such a test case because they typically use an input generation strategy to assign random values to variables; since simply leaving a variable undefined before it is used without knowing the context will frequently trigger many correctly handled runtime exceptions. As we will show later in the paper, by using ECMA-262 to guide the test input generation, Comfort successfully produced this test case and uncovered a new bug in Rhino, which was not covered by the hand-written Test262 test suite.

2.4 Transformer-based Test Program Generation

Our program generator is based on GPT-2, a Transformer-based neural generation architecture [50]. Most of the recently proposed deep-learning-based compiler fuzzers [14, 19, 25, 33, 37] use a recurrent neural network (RNN), e.g., LSTM [26], to model the program source code sequence to generate test programs. A Transformer architecture does not rely on any RNN mechanism. Instead, it uses an attention mechanism to model the sequence dependence. The attention mechanism takes an input sequence and decides at each step which other parts of the sequence are important. Prior studies showed that the Transformer outperforms RNN in modeling long-term sequence dependence for natural language processing [42]. In this work, we utilize an open-source pre-trained GPT-2 model [50] but re-target it to model and generate JS programs (Section 3.2).

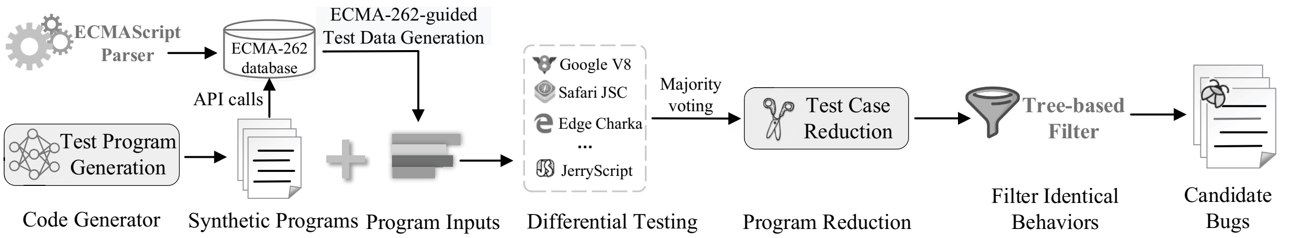
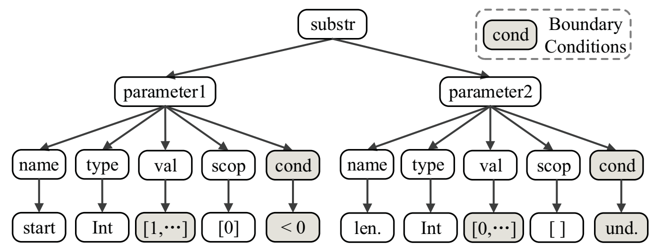


Figure 3. Overview of Comfort. We use GPT-2 to generate test JS programs. For JS APIs in a test program, we utilize the API definitions and boundary conditions from the ECMA-262 database to generate test data. We execute multiple JS engines on the test cases and use differential testing to identify potential buggy behaviors.

3 Comfort

Figure 3 provides a high-level overview of Comfort. Comfort is designed to use the language specification to guide test case generation. To this end, we build an automated parser to extract the pseudo-code-like JS API rules (see Figure 1 for an example) from ECMA-262. We store the parsing results in a structured database. To generate JS *test programs*, we first use our GPT-2 program generator to produce random JS programs (Section 3.2). To create *test data* for a test program, Comfort first extracts the APIs and their arguments from the program (Section 3.3). It then looks up the extracted API rules to generate inputs (i.e., by assigning values to variables in the JS code) that can trigger the boundary conditions of an API definition. To enrich the pool of inputs, we also generate some random input values. A JS test program and one of its datasets then form a *test case*, and a test program can be associated with multiple input datasets. The generated test cases are used to test JS engines through differential testing (Section 3.4). Before presenting a potentially buggy-exposing test case to the developer, we apply a simple test case reduction algorithm to reduce the test case (Section 3.5). Finally, to minimize developer involvement, we use an incrementally built knowledge base to automatically analyze the testing outcomes to filter out test cases that may trigger identical miscompilation behaviors seen before (Section 3.6).



(a) The extended AST

```
{
  "String.prototype.substr": [{
    "name": "start",
    "type": "integer",
    "values": [1, -1, "NaN", 0, "Infinity", "-
        Infinity"],
    "scopes": [0],
    "conditions": ["start < 0"]
  }, {
    "name": "length",
    "type": "integer",
    "values": ["undefined", "NaN", 0, "Infinity", "-
        Infinity"],
    "scopes": [],
    "conditions": ["length === undefined"]
  }]
}
```

(b) The JSON file of the AST in (a)

Figure 4. The AST (a) and its JSON format (b) for encoding specifications of the `substr` function in Figure 1.

3.1 Extracting ECMA-262 Specification

The goal of our ECMA-262 parser is to extract specification rules from ECMA-262. In this work, we use the HTML version of the ECMA-262 document. Given the ECMA-262 document, our parser first detects the scope of a function, class or object by analyzing the metadata. Specifically, we use Tika [58], a content analysis library, with the help of handwritten regular expressions (regex) to extract the metadata (i.e., ECMA-262 specification rules) of the HTML document. For example, we use the regex `^Let $Var be $Func$Edn$` to extract the initialization condition defined at line 4 of Figure 1. Our initial regular expression rules that account for 80% of the rules used in experiments were written by a postgraduate student within a week. We have since improved and grown our regular expression set.

The extracted specification is organized in the form of an abstract syntax tree (AST) where the root of a tree is an ECMA-262 function, class, or object and the children node is a rule defined for a specific object or function. We tag boundary conditions with special attributes on the AST. Figure 4(a) shows the extended AST for the specification given in Figure 1 where the boundary conditions and literals are tagged with unique attributes. We then store the AST in the JSON format as illustrated in Figure 4(b) where the boundary conditions and value ranges are recorded. Since we primarily target JS APIs, the extracted rules are used to generate boundary conditions for mutating test programs related to JS APIs.

Note that there are other JS language specifications and definitions described in the natural language form. We do not extract rules from such definitions. Overall, our extracted rules cover around 82% of API and object specification rules in ECMA-262 10th Edition (2019 version).

3.2 Test Program Generation

Language model. We employ a DNN-based language model to generate JS test programs. Our JS code generator is built by fine-training a pre-trained GPT-2 [46] (that was trained on natural language documents by independent researchers) on JS programs collected from open-source projects hosted on GitHub. The model requires each training input (i.e., JS programs) to be represented as a sequence of numerical values. To do so, we map every instruction, constant, and variable to an integer by looking up the token in a vocabulary table. We construct the vocabulary using the Byte Pair Encoding (BPE) tokenization method [53], which is also used by GPT-2. This scheme works by first counting each word's frequency in the training data. It then breaks each word into chunks (or subwords) based on the word frequency. For example, commonly seen language keywords, e.g., `var`, `for`, and `if`, will be tokenized as whole words, while rarer words like variable names will be broken into smaller chunks (e.g., a few characters) and can be used to create the rest of the words. The algorithm tries to find a way to represent the training dataset with the least amount of tokens and map each sub-word or token to an integer to be stored in the vocabulary table. This scheme allows us to deal with a potential infinite number of words seen in real-life JS programs by reusing tokens from a finite list of subwords.

Model training. To collect training data to port the pre-trained GPT-2 for generating JS programs, we developed an automated tool to collect 140,000 JS programs from 4,000 top-ranked (i.e., projects with the greatest numbers of stars) GitHub projects with JS as the main programming language. We use the collected JS programs to update the weights of the last two fully-connected layers of the pre-trained GPT-2 model while keeping weights of other layers unchanged. The network is trained using the Adam optimizer [29] for 100 epochs (over 150,000 iterations) with an initial learning rate of 0.0001 and decaying by 10% every epoch. Training the GPT-2 model took around 30 hours using four NVIDIA GTX 2080Ti desktop GPUs, which was a one-off cost. Note that we provided the model with no prior knowledge of the structure or syntax of JS.

JS program generation. We use the trained GPT-2 to generate JS test programs. Each of the test programs contains a JS function (e.g., function `foo` at lines 1–4 in Figure 2), which may invoke some standard JS APIs. To generate a test program, we feed the network with a randomly chosen seed generation header (e.g., "`var a = function(assert) {`"). The seed generation header is chosen from a corpus of 2,000 function header samples, which were automatically collected from our JS training dataset. We ask the network to produce the next token based on the current text string, s_c. To choose the next token, we employ a top-k sampling scheme, by randomly choosing a token from the top-k tokens that are predicted to have the highest possibilities by following s_c (we empirically set k to 10). We then append the chosen character to the existing string, s_c and feed the new string to the network to repeat the process to generate the next token. This generation process terminates when the brackets (i.e., '{' and '}') are matched, or a dedicated termination symbol "<EOF>" is produced by the language model, or the number of words of the synthetic program is over 5,000. For each generated test program, we use JSHint [3] to *statically* check its syntax to remove test programs with syntax errors. Our current implementation also randomly keeps 20% of the syntactically-invalid test programs for test runs. A better approach for choosing syntax-incorrect programs for testing would consider program characteristics like API coverage and code length. We leave this as our future work. Later in the paper, we show that compared to DeepSmith [14], GPT-2 can model longer dependences among tokens in the source code, which in turns leads to a high success rate in producing syntactically correct test programs (see Section 5.3.3).

3.3 ECMA-262-guided Test Data Generation

Our test data is embedded into the JS code by assigning values to variables that are passed to a JS function. In addition to generating variables, we also generate code to call functions with supplied parameters and print out the results. For example, lines 5–9 in Figure 2 show the test data and the corresponding test-driven code produced by Comfort.

Algorithm 1 presents our test data generation. Our approach takes in a JS test case program and outputs multiple test cases. For each statement in the input JS test program, we first check if the statement contains a function invocation, and then locate the JS API definitions and the potential arguments of the API, by using the API name to look up our ECMA-262 database (lines 1–7). As the *training corpus* for our test program generation model contains human-written programs with code and data of various API calling patterns, the test programs generated by Comfort may already contain code and data to invoke a JS API in different ways. Our approach keeps these test cases but will use the ECMA-262 rules to mutate the values assigned to function arguments to generate additional test data samples. Specifically, it uses the ECMA-262 rules extracted offline to determine how many parameters should be passed to a function, and the type and value for each parameter. For each parameter type, we mutate the values based on (1) boundary conditions according to the ECMA-262 specification (e.g., in Figure 1, arguments `len` is set to `undefined`), and (2) normal conditions (using random values). To mutate the variable values, we associate an argument passed to a function with its definition by traversing the JS program's control and data flow graph (line 8). These parameter values and the input JS test program are then store in our list of test case programs (line 9).

Algorithm 1 ECMA-262-guided Test Data Generation

Input:
t_{prog}: A JS test case program
$ecma$: A supported edition of ECMA-262
Output:
T_{new}: A list of mutated test case programs
$Specs \leftarrow$ extractFuncSpecs($ecma$)
Let T_{new} be a list
for $st \in t_{prog}$ **do**
  if isFunc(st) **then**
    $funcName \leftarrow$ getFuncName(st)
    if $Specs$.containsName($funcName$) **then**
      $spec_{func} \leftarrow$ getSpecs($Specs$, st)
      $t_{new} \leftarrow$ mutate(t_{prog}, $func$, $spec_{func}$)
      T_{new}.append(t_{new})
    end if
  end if
end for
return T_{new}

3.4 Differential Testing

We employ the established differential testing methodology [12] to expose JS compiler defects by running a test case across multiple JS engines (or Testbeds). We use a majority voting scheme to determine which compiler's behavior deviates from others by comparing the results of compilation and execution. Differential testing typically requires the test program to yield a deterministic, well-defined outcome [35]. The use of ECMA-262 specifications to generate test data enable us to create test cases with expected deterministic behaviors. The test programs generated by our language model may have non-deterministic outcomes like floating-point rounding errors. However, we did not experience this problem in our test runs - if the behavior of a JS engine deviates from others, it was typically due to a bug of the compiler implementation during our test runs.

Executing a test case on a JS engine leads to one of seven possible outcomes, illustrated in Figure 5. A consistent parsing error occurs when the parsing results (both successful or not successful) are inconsistent. We ignore test JS programs that fail to be successfully parsed by all test engines. The successfully parsed JS programs will be executed, which can lead to five runtime outcomes. A wrong output occurs if the execution results are inconsistent among JS engines when a deterministic behavior is expected according to ECMA-262. This often happens when a JS engine produces a result (e.g., throwing an exception when it should not) that is different from the expected behavior defined in ECMA-262. A runtime crash occurs if the JS engine crashes when executing a test case. A runtime timeout happens if a JS test case fails to terminate when it is running with a JS engine within a period of $2t$, where t is the longest time for all other JS engines to return the result. Note that we ignore test cases where all JS engines do not terminate within ten minutes because

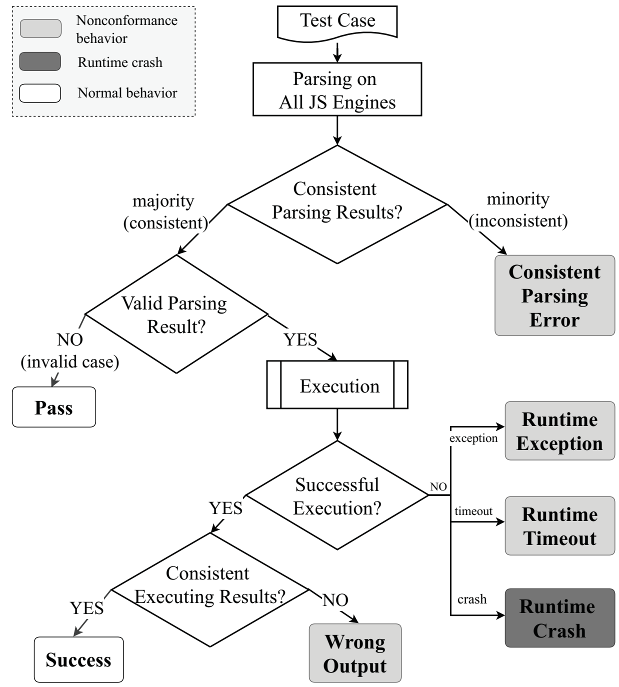


Figure 5. Possible outcomes when executing a test case.

this is likely to be due to a large or infinite loop in the test program. Finally, we consider a test case to be a passing one if all tested JS engines can successfully execute it, and the executions lead to a consistent outcome.

When evaluating the outcomes of test cases, compile crash and timeout outcomes are of immediate interest, indicative of erroneous compiler behavior. For all other outcomes, we use differential testing methodology to confirm anomalous behavior. Specifically, we compare the results obtained for a test case using multiple JS engines, including JS engines in the same family but from different trunk builds. We look for test cases where a JS compiler's behavior deviates from all others, which can then be investigated by developers.

3.5 Test Case Reduction

To help developers in examining bug-exposing test cases, we develop a simple yet effective approach to reduce the size of a test case that triggers an anomalous compiler behavior. Our idea is to traverse the abstract syntax tree of the input test program to iteratively remove code structures and test if the resulted program can still trigger the same compilation or execution outcome. We repeat this process until a fixpoint where no additional reduction from can be done while still reproducing the same anomalous behavior as the original test case does. We note that there are other test case reduction tools like HDD [44] and Perses [55] to be used for the same purpose.

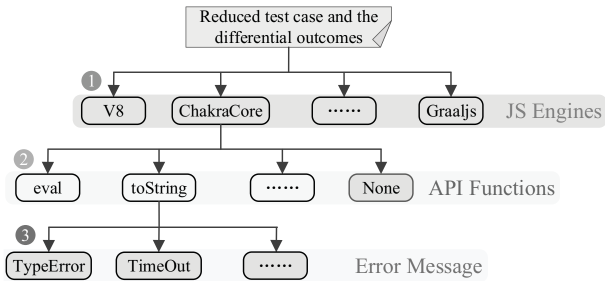


Figure 6. Our tree-based identical bug filter.

3.6 Filtering Identical Miscompilation

Random program generation often produces test cases that trigger the same compiler bugs. To reduce developer involvement, we employ an automated scheme to filter out test cases that are likely to cause identical miscompilation behaviors. To do so, we construct a simple tree-based knowledge base from historical bug-exposing test cases. Our tree-based classifier to identify identical bugs is inspired by prior work on employing decision trees for predictive modeling [21, 62, 63]. We choose this technique because the model is interpretable.

Figure 6 presents a high-level overview of our tree-based knowledge base, which provides three layers to predict if a test case triggers a bug that was seen by an already analyzed test case. Every decision node in the top layer corresponds to a JS engine, which checks if the test case triggers the same bug seen for a JS engine. Likewise, the API function nodes in the second layer check if the test case triggers a bug of a specific JS API. If the test case does not contain a JS API function, it will be classified into the None leaf node in the second layer. The leaf nodes in the last layer group the differential results based on the miscompilation behaviors (such as TypeError, TimeOut, Crash, etc.) that the tested JS engine yields. If a test case triggers a buggy behaviour during test runs as described in Section 3.4, we then traverse this tree to see if there already exists a path that gives identical information as exposed by the test case. If so, we consider a previously identified bug is found. If not, we consider a new bug is triggered for a given JS engine and JS API. For the latter case, we then add a new leaf node to the tree according to the bug the test case triggers.

We have so far created a knowledge base consisting of 2,300 leaf decision nodes. In our current implementation, the tree is realized as a set of rules written in Python. These rules already helped us filter out over ten of thousands of repeated miscompilation behaviors, significantly reducing the overhead and human involvement of examining test cases to confirm bugs.

4 Experimental Setup

4.1 JS Engines

Table 1 lists the JS engines and versions used in our evaluation. We apply Comfort to ten JS engines and test several trunk branches of each engine. All the tested JS engines were claimed to be compatible with a version of ECMA-262. In evaluation, we ensure that we only test each JS engine against the corresponding supported ECMA-262 edition. In total, we have tested 51 JS engine-version configurations.

Table 1. JS engines we have tested.

JS Engine	Versions	Build No.	Release Date	Supported ES Spec.
V8	V8.5	d891c59	Jun. 2020	ES2019
	V8.5	e39c701	Aug. 2019	
	V8.5	0e44fef	Apr. 2019	
ChakraCore	v1.11.19	5ed2985	May 2020	ES2019
	v1.11.16	eaaf7ac	Nov. 2019	
	v1.11.13	8fcb0f1	Aug. 2019	
	v1.11.12	e1f5b03	Aug. 2019	
	v1.11.8	dbfb5bd	Apr. 2019	
JSC	261782	dbae081	May 2020	ES2019
	251631	b96bf75	Oct. 2019	
	246135	d940b47	Jun. 2019	
	244445	b3fa4c5	Apr. 2019	
SpiderMonkey	v78.0	C69.0a1	Jun. 2020	ES2018/2019
	gecko-dev	2c619e2	May 2020	
	gecko-dev	201255a	Jun. 2019	
	v60.1.1	mozjs60.1.1pre3	Jul. 2018	
	v52.9	mozjs52.9.1pre1	Jul. 2018	
	v38.3.0	mozjs38.3.0	Oct. 2017	
	v1.7.0	js-1.7.0	Sep. 2017	
Rhino	v1.7.12	d4021ee	Jan. 2020	ES2015
	v1.7.11	f0e1c63	May 2019	
	v1.7.10	1692f5f	May 2019	
	v1.7.9	3ee580e	Mar. 2018	
	v1.7R5	584e7ec	Jan. 2015	
	v1.7R4	82ffb8f	Jun. 2012	
	v1.7R3	d1a8338	Apr. 2011	
Nashorn	v13.0.1	JDK13.0.1	Sep. 2019	ES2011/2015
	v12.0.1	JDK12.0.1	Apr. 2019	
	v11.0.3	JDK11.0.3	Mar. 2019	
	v1.8.0_201	JDK8u201	Jan. 2019	
	v1.7.6	JDK7u65	May 2014	
Hermes	v0.6.0	b6530ae	May 2020	ES2015
	v0.4.0	044cf4b	Dec. 2019	
	v0.3.0	3826084	Sep. 2019	
	v0.1.1	3ed8340	Jul. 2019	
JerryScript	v2.3.0	bd1c4df	May 2020	ES2011/2015
	v2.2.0	996bf76	Nov. 2019	
	v2.2.0	7df87b7	Oct. 2019	
	v2.1.0	84a56ef	Oct. 2019	
	v2.1.0	9ab4872	Sep. 2019	
	v2.0	351acdf	Jun. 2019	
	v2.0	b6fc4e1	May 2019	
	v2.0	40f7b1c	Apr. 2019	
	v2.0	e944cda	Apr. 2019	
QuickJS	2020-04-12	1722758	Apr. 2020	ES2019
	2020-01-05	91459fb	Jan. 2020	
	2019-10-27	eb34626	Oct. 2019	
	2019-09-18	6e76fd9	Sep. 2019	
	2019-09-01	3608b16	Sep. 2019	
	2019-07-09	9ccefbf	Jul. 2019	
Graaljs	v20.1.0	299f61f	May 2020	ES2020

4.2 Testbeds

For each JS configuration, we create two testbeds. In the first, the engine runs under the normal mode. In the second, the engine runs under the strict mode. The "strict mode" is introduced since ECMAScript 5, which allows one to run a JS program in a "strict" operating context. Programs running in the strict mode can have different semantics from normal code. For example, the strict mode eliminates some JS silent errors by forcing them to throw errors per the ECMA-262 standard. This testing mechanism gives us a total of 102 testbeds (51 JS engine version configurations × 2 testbeds

Table 2. Bug statistics for each tested JS engine.

JS Engine	#Submitted	#Confirmed		#Acc. by Test262
		#Verified	#Fixed in #Verf.	
V8	4	4	3	1
ChakraCore	7	7	5	1
JSC	12	11	11	3
SpiderMonkey	3	3	3	0
Rhino	44	29	29	4
Nashorn	18	12	2	1
Hermes	16	16	15	4
JerryScript	35	31	31	3
QuickJS	17	14	14	4
Graaljs	2	2	2	0
Total	**158**	**129**	**115**	**21**

per configuration) to evaluate. For the remainder of the paper, unless state otherwise, the bugs are reported to be found under both the normal and the strict modes.

4.3 Test Case Generation

We use Comfort to produce a total of 300k synthetic test cases. We use JSHint [3], a static JS parser to remove syntactically incorrect test programs. However, we still keep a small number (10k) of randomly chosen test cases that are considered to have syntax errors to test the parser of a JS engine. On average, we keep 250k test cases.

4.4 Competitive Baselines

We compare Comfort against both generation- and mutation-based fuzzers. Specifically, we compare Comfort with DeepSmith [14], a closely related DNN-based program generator for compiler fuzzing. We also compare Comfort to four mutation-based JS fuzzers: Fuzzilli [22], CodeAlchemist [24], DIE [49] and Montage [33], where the last three represent the state-of-the-art JS compiler fuzzers.

4.5 Hardware Platforms

Our evaluation platform is a multi-core server with a 3.6GHz 8-core (16 threads) Intel Core i7 CPU, four NVIDIA GTX 2080Ti GPUs and 64GB of RAM, running Ubuntu 18.04 operating system with Linux kernel 4.15. All DNN models run on the native hardware using the GPUs. For fuzzing tests, we run each testbed in a Docker container (version 19.03.5) so that we can run 16 fuzzing processes simultaneously.

5 Experimental Results

From May 2019, we have started experimenting with and refining Comfort to find bugs in Rhino and then gradually extended our tests to other JS engines. From May 2020, we started our extensive testing of all JS engines. In total, we test each JS testbed through 200 hours automated runs on 250k automatically generated test cases. Unless stated otherwise, Comfort code listings are presented verbatim, with only minor formatting changes applied to save space. No manual test case reduction was performed.

Table 3. The number of bugs found per JS engine version.

JS Engine	Versions	#Submitted	#Confirmed		
			#Verified	#Fixed	#New
v8	V8.5	4	4	3	4
ChakraCore	v1.11.16	3	3	1	3
	v1.11.13	1	1	1	0
	v1.11.12	1	1	1	1
	v1.11.8	2	2	2	2
JSC	261782	1	1	1	1
	251631	2	1	1	1
	246135	8	8	8	6
	244445	1	1	1	0
SpiderMonkey	v52.9	1	1	1	0
	v38.3	1	1	1	0
	v1.7	1	1	1	0
Rhino	v1.7.12	25	19	19	19
	v1.7.11	17	8	8	4
	v1.7.10	2	2	2	2
Nashorn	v13.0.1	4	4	0	4
	v12.0.1	14	8	2	7
Hermes	v0.6.0	2	2	2	2
	v0.4.0	1	1	0	1
	v0.3.0	6	6	6	5
	v0.1.1	7	7	7	4
JerryScript	v2.3.0	2	2	2	2
	v2.2.0	18	16	16	15
	v2.1.0	6	5	5	4
	v2.0	8	7	7	7
	v1.0	1	1	1	1
QuickJS	2020-04-12	1	1	1	1
	2020-01-05	2	2	2	2
	2019-10-27	4	3	3	3
	2019-09-18	3	1	1	1
	2019-09-01	4	4	4	4
	2019-07-09	3	3	3	1
Graaljs	v20.1.0	2	2	2	2
Total	**33**	**158**	**129**	**115**	**109**

Highlights. As of November 2020, we have indentified[5] 158 unique bugs (of which 109 were found to be newly discovered bugs by developers). So far, 129 have been verified, of which 115 have been fixed by the developers. For the remaining 29 unverified bugs, 9 were rejected by the developers as the feature was either not clearly defined in ECMA-262 or not supported by the compiler version; and others were either under discussion or yet to be confirmed. Many of the bugs discovered by Comfort were not exposed by the state-of-the-art compiler fuzzing methods. Moreover, 21 of the Comfort-generated test cases have been added to Test262, of which we submitted 18 test cases, and the JSC developers submitted 3 Comfort-generated test cases after we have reported the bug.

5.1 Quantitative Results

This subsection presents various summary statistics on results from our JS compiler testing effort.

5.1.1 Bug count. Table 2 gives the distribution of the Comfort-exposing bugs across the tested JS engines. Each of all the ten evaluated JS engines has at least one conformance bug even though all of them claimed to adhere to the ECMA-262 specification tested. This shows the prevalence

[5]A list of our bug reports can be found at: https://github.com/NWU-NISL-Fuzzing/COMFORT/blob/main/artifact_evaluation/docs/Bug-List.md

Table 4. Bug statistics for each group.

Category	#Submitted	#Confirmed	#Fixed	#Acc. by Test262
Test program generation	97	78	67	5
ECMA-262 guided mutation	61	51	48	16

of conformance bugs across different JS engines. Note that when a bug is confirmed and triaged, it corresponds to a new defect. Therefore, all confirmed bugs that we reported were unique and independent. Although we have checked that all reported bugs have different symptoms, some of them were actually linked to the same root cause. A total of eight such bug reports were later marked as *duplicate* by developers.

Note that not all confirmed bugs of Nashorn were fixed because their developers ceased maintaining this engine after June 2020. It is not surprising that JS engines like V8 and SpiderMonkey with a long development and testing history and a larger developer community have fewer bugs than some newer open-source JS engines like JerryScript. We found the SpiderMonkey implementation to be well-conformed to ECMA-262, and we only found three conformance bugs in a previous release.

Table 3 shows the number of unique bugs found per JS engine version. Note that some of the bugs may exist across different versions of the same JS engine. For clarity, we only attribute the discovered bugs to the earliest bug-exposing version used in our evaluation. In total, Comfort discovered 38 new bugs from the latest versions listed in Table 3. Furthermore, Comfort has also found a considerable number of bugs in stable releases that had been latent for years.

It is worth mentioning that for Rhino version 1.7.12 and JerryScipt version 2.2.0, Comfort found over 15 conformance bugs, far more than the number of bugs found in other versions of these two engines. This is because Rhino and JerryScipt initially supported ECMA-262 version 5 but have recently added the support for ECMA-262 version 6 at these two versions. This larger number of conformance bugs introduced when switching to a new ECMA-262 edition is expected as some of the API implementations in ECMA-262 version 6 are different from those in version 5.

5.1.2 Bug types.

We distinguish two kinds of bugs: (1) ones that manifest through our test program generation (Section 3.2) and (2) ones that manifest by exploiting the ECMA-262 specification to generate test data (Section 3.3). Table 4 classifies the bugs found by Comfort into these two groups. Our GPT-2 test program generator is effective in generating bug-exposing test cases which contribute to 97 of the submitted bugs. By exploiting ECMA-262, Comfort is able to discover further 61 bugs of which 51 have been confirmed. All of the bugs discovered in this category are standard conformance bugs. Furthermore, 16 of the automatically generated test cases under this category have been added into Test262. This shows the usefulness and importance in exploiting the language standard for exposing compiler bugs.

Table 5. Statistics on top-10 buggy object types.

API Type	#Submitted	#Confirmed	#Fixed
`Object`	23	21	18
`String`	22	20	19
`Array`	17	12	9
`TypedArray`	8	5	5
`Number`	5	4	4
`eval function`	4	4	4
`DataView`	4	2	2
`JSON`	3	3	2
`RegExp`	2	2	1
`Date`	2	1	1
Total	**90**	**74**	**65**

5.1.3 API distribution.

Table 5 groups the buggy JS API implementations found by Comfort according to the object type. To investigate the APIs that are more likely to contain conformance bugs, we analyze the top-10 buggy object types. Most of the bugs found by Comfort are object and string operations. This is due to the large number of standard JS APIs provided for these two data types. For example, eight of the confirmed and fixed bugs found by Comfort were found for `String.prototype.replace()` due to improper handling the type and number of the arguments. We have also found four confirmed bugs for the `eval` function for ChakraCore, Hermes and QuickJS due to the inappropriate implementations for expression parsing and evaluation. One of such examples is given in Listing 7. This table shows that conformance bugs can be found on a range of APIs.

5.1.4 Affected compiler components.

We grouped the Comfort-discovered bugs according to the typical JS engine components: code generation (CodeGen), API and library implementation (Implementation), Parser, the regular expression engine (Regex Engine), and Optimizer. We also list bugs found solely in the strict mode. Figure 7 shows the number of bugs discovered by Comfort for each component. Most of the bugs found were due to erroneous implementations in the back-end code generator. Bugs due to the library and API implementation are also common - 45 confirmed and 41 fixed bugs belong to this category. According to the developer feedback, this is often due to an oversight or misunderstanding of the ECMA-262 specification.

5.2 Bug Examples

Comfort is capable of finding diverse types of JS engine bugs. To provide a glimpse of the diversity of the uncovered bugs, we highlight here several of the Comfort-produced test cases that expose a JS compiler bug.

`defineProperty` API. All versions of V8 tested fail to correctly compile the test case shown in Listing 1. This test case contains a "type error" because the `length` property at line 3 is *not* configurable but the code tries to change it. Such an

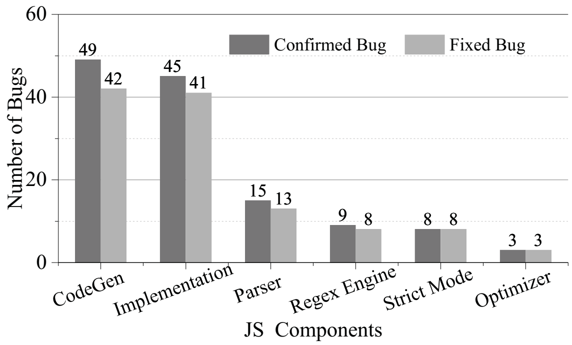


Figure 7. #Comfort-found bugs per compiler component.

attempt should lead to a TypeError exception, but V8 allows the code to be successfully compiled. This bug is exposed by generating a test data to manipulate a non-configurable property of the array object according to ECMA-262. We were the first to report this bug. This test case also exposes a bug of Graaljs for the same reason.

```
var foo = function(){
  var arrobj = [0, 1];
  Object.defineProperty(arrobj, "length", {
    value: 1, configurable: true
  });
};
foo();
```

Listing 1. All versions of V8 tested fail to throw a TypeError exception for this test case.

Performance bug. The test code in Listing 2 exposes a performance issue of Hermes's memory allocation policy. Hermes took more than half an hour to execute the code while other JS engines use less than a second. Due to its memory allocation strategy, Hermes has to relocate the array for every new element inserted on the left (when the array elements are added in reverse order), leading to significant runtime overhead with a large array. This test case was generated by our GPT-2 test program generator. This bug affects all versions prior to V0.3.0, and we were the first to report this. Our bug report was welcomed and quickly fixed by the Hermes developers.

```
var foo = function(size) {
  var array=new Array(size);
  while (size--){
    array[size] = 0;
  }
};
var parameter = 904862;
foo(parameter);
```

Listing 2. Hermes took 30+ minutes to execute this code while other engines took less than a second.

`Uint32Array`. SpiderMonkey prior to v52.9 incorrectly threw a TypeError exception for the code shown in Listing 3. This is because it does not convert the function argument to an integer type per ECMA-262 before calling the `Uint32Array` API at line 2. A standard conforming implementation would convert 3.14 to 3 to be used as the array length.

```
1 var foo = function(length){
2   var array = new Uint32Array(length);
3   print(array.length);
4 };
5 var parameter = 3.14;
6 foo(parameter);
```

Listing 3. SpiderMonkey before v52.9 incorrectly throws a TypeError exception.

`Number.prototype.toFixed`. Rhino compiled and executed the test case in Listing 4 to produce an output of "-634619", when it should throw a RangeError exception. ECMA-262 states `Number.prototype.toFixed` only takes a value between 0 and 20. Hence, this is a conformance bug.

```
var foo = function(num){
  var p = num.toFixed(-2);
  print(p);
};
var parameter = -634619;
foo(parameter);
```

Listing 4. Rhino fails to throw a RangeError exception for this test case.

`%TypedArray%.prototype.set`. The tested JSC trunk builds prior to v261782 threw a TypeError exception when executing the test case in Listing 5. However, the expected output is "1,2,3,0,0" per ECMA-262. The root cause of this bug is because JSC does not convert the String object `e` at line 2 to an Array object to be used at line 4. Comfort can generate a bug-exposing test case by exploiting the ECMA-262 rule for `%TypedArray%.prototype.set`. This bug is confirmed and fixed by JSC developers. This test case also triggered a similar bug in Graaljs.

```
1 var foo = function(){
2   var e = '123';
3   A = new Uint8Array(5);
4   A.set(e);
5   print(A);
6 };
7 foo();
```

Listing 5. JSC prior to v261782 throws a TypeError while the code is correct per ECMA-262.

Array allocation bug. Listing 6 gives a bug-exposing test case generated by Comfort. When setting the object property at line 4, QuickJS appended the right-hand-side value to the end of the `obj` array (i.e., leading to an array of 4 elements rather than setting the object property). As a result, when running this code with QuickJS, the program gives an erroneous output of "1,2,5,10\n undefined". This bug will only be triggered if the `property` is set to `true`. By utilizing

ECMA-262 to generate the test data, Comfort is able to expose this implementation bug. This bug was verified by the developer on the same day of submitting the bug report.

```
1 var foo = function(){
2   var property = true;
3   var obj = [1,2,5];
4   obj[property] = 10;
5   print(obj);
6   print(obj[property]);
7 };
8 foo();
```

Listing 6. QuickJS incorrectly appends the property value (line 4) as a new element of the array object.

`eval` function. When parsing the test case in Listing 7, a JS compiler should throw a SyntaxError exception. This is because according to ECMA-262, the *for-loop* expression passed to `eval` should contain a loop body or end up with a semicolon (i.e., an empty loop). However, ChakraCore allows this code to successfully compile and run, which is thus a conformance bug. Comfort generates this test case by exploiting the edge cases defined in ECMA-262. This bug was confirmed and fixed by the ChakraCore developers, and our test case was also added into Test262 later.

```
var foo = function(cmd){
  eval(cmd);
  print("Run␣Here␣1");
};
var str = "for(;false;)";
foo(str);
```

Listing 7. ChakraCore fails to throw a SyntaxError when parsing the expression passed to `eval`.

`String.prototype.split`. Listing 8 shows a test case that triggered a conformance bug of JerryScript. The program applies a regular expression to split a String object starting with the capital letter 'A'. In this case, the program should produce "anA" as an output as the string does not match the regular expression. JerryScript yields "an" as the output due to the incorrect implementation of it regular expression parsing engine. This test program was automatically generated by our GPT-2 code synthesizer by learning from open-source JS programs, and it was also added to Test262.

```
var foo = function(){
  var a = "anA".split(/^A/);
  print(a);
};
foo();
```

Listing 8. This test program exposes a bug of the JerryScript regular expression parser.

Crash. QuickJS crashed when compiling the test case in Listing 9 when an empty string invokes `normalize()`. Comfort generates this test data by learning from open-source JS code. It was confirmed by the QuickJS developers that this is a memory safety issue that could be exploitable to run arbitrary code. Hence, this bug was quickly fixed.

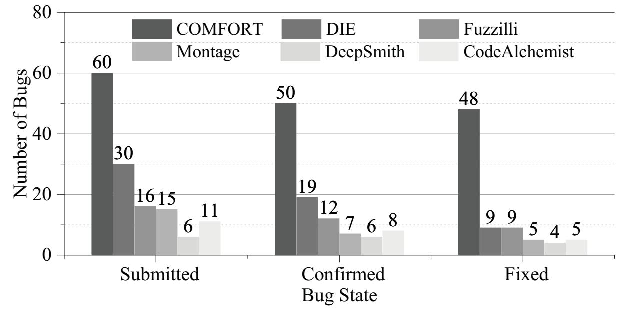


Figure 8. The number of bugs found by different fuzzers during 72 hours automated testing runs and a 3-month bug confirmation and fixing window.

```
var foo = function(str){
  str.normalize(true);
};
var parameter = "";
foo(parameter);
```

Listing 9. Test case that leads to a QuickJS compilation crash.

5.3 Compare to Prior Compiler Fuzzers

We compare Comfort against five compiler fuzzers [14, 22, 24, 33, 49], of which four were specifically designed for fuzzing JS compilers [22, 24, 33, 49]. We consider the capability in exposing bugs (Sections 5.3.1 and 5.3.2) and the quality of the generated test cases (Section 5.3.3).

5.3.1 Bug exposing capability. In this experiment, we test each JS Testbed for 72 hours of consecutive testing runs, using test cases generated by different fuzzers. An average of 20k offline generated, syntactically correct test cases were evaluated on each Testbed. Here, the testing time refers to the engine execution runtime (by excluding test case generation time). To provide fair comparison, for mutation-based fuzzers, we use the seed programs provided in the source publications for test case generation; and we train DeepSmith [14] using the same training JS corpus as Comfort. In this experiment, we leave at least three months for the relevant JS engine developers to confirm and fix a submitted bug. Based on our experience, an important bug is usually confirmed by the developers within two weeks and gets fixed within three months. Test runs were conducted in May 2020, and we ran the bug confirmation and fixing window until October 2020. We exclude Nashorn in this experiment as it was no longer under active development since June 2020.

As can be seen from Figure 8, Comfort discovered more distinct bugs than any other individual fuzzer during the testing period of this experiment. In 200 hours of testing, Comfort discovered a total of 60 unique bugs across all Testbeds. It helped the vendors fixed over 95% of the confirmed bugs in the 3-month time frame. These numbers are more than the sum of fixed bugs discovered by other fuzzers. DeepSmith, the most closely related DNN-based generative based fuzzer, discovered a total of six bugs from four tested JS engines. By contrast, Comfort discovered bugs in all tested engines. Furthermore, Comfort alone discovered 31 confirmed bugs that were not uncovered by other fuzzers. By comparison, a total of 29 confirmed bugs discovered by five other fuzzers all togehter were not exposed by Comfort during the test runs. This experiment shows that Comfort is effective in uncovering JS conformance bugs.

5.3.2 Test cases generated by other fuzzers. We now discuss some bug-exposing test cases generated by other fuzzers, which were not covered by Comfort-generated test cases during this experiment.

CodeAlChemist. The CodeAlChemist test case in Listing 10 exposes a conformance bug of Rhino. For this case, a TypeError exception should be thrown because a `null` argument is passed to `String.prototype.big.call`. Comfort does not generate such a case because none of the training JS programs used to train our program generator uses `String.prototype.big.call` and hence our program generator does not learn to generate test cases using this API.

```
var v0 = (function(){
    print(String.prototype.big.call(null));
});
v0();
```

Listing 10. CodeAlChemist generated test case.

Fuzzilli. For the Fuzzilli generated test case in Listing 11, Rhino crashed when executing the seal function at line 3. This is an implementation error that is not defined as a ECMA-262 boundary case. Thus, Comfort does not generate a similar test case.

```
1 function main(){
2     var v2 = new String(2477);
3     var v4 = Object.seal(v2);
4 }
5 main();
```

Listing 11. Fuzzilli generated test case.

DIE. The test case generated by DIE in Listing 12 exposes a bug of Rhino and JerryScript. At lines 2-5, the JS code set the `lastIndex` property of the `regexp5` object (defined at line 1) to be *non-writable*. The program later at line 5 compiles the regular expression, which effectively will set the `lastindex` to 0 per the ECMA-262 standard. For this case, a TypeError exception should be thrown when executing the statement at line 5, but Rhino and JerryScript permit such a change (which thus is a bug). This ECMA-262 definition was given in the natural language form and hence is not captured by our ECMA-262 parser (Section 3.1). Comfort was able to generate test case to trigger a similar type of bug in Listing 1 because that definition is written in the pseudo-code form.

```
1 var regexp5 = new RegExp(/abc/);
2 Object.defineProperty(regexp5, "lastIndex",
      {
3     value: "\w?\B", writable: false
4 });
5 regex5 = regexp5.compile("def");
6 print(regexp5.lastIndex);
```

Listing 12. DIE generated test case.

Montage. The test code generated by Montage in Listing 13 is an *immediately invoked function expression* because the function at line 1 and the variable name at line 2 are both named `v1`. Hermes and Rhino produced an output of "false\n number", while other JS engines output "true\n function". This is an undefined behavior in ECMA-262 and hence is not exposed by Comfort. After submitting the report, the developers of Hermes and Rhino decided to label it as a bug.

```
1 (function v1(){
2     v1 = 20;
3     print(v1 !== 20);
4     print(typeof v1);
5 }());
```

Listing 13. Montage generated test case.

5.3.3 Quality of test cases generation. To evaluate the quality of the generated test cases, we consider two metrics:

Syntax passing rate. This quantifies the ratio of the generated JS programs that are syntactically valid (judging by JSHint - a static JS parser). We ask each fuzzer to generate 10,000 JS programs and compute the passing rate.

Coverage. We consider three widely-used coverage metrics of the generated test cases [30, 37]: statement coverage, function coverage, and branch coverage. The three metrics respectively measure the average ratio of statements, functions and branches of a *test* JS *program* that get executed during the test run. For fair comparison, we randomly select 9,000 syntactically valid test programs generated by each fuzzer to compute the coverage. We use `Istanbul` [28], a JS code coverage tool, to collect the information.

Results. As can be seen from Figure 9, Comfort gives a passing rate of 80%, an improvement over the less than 60% passing rate given by all alternative methods. Note that among the syntactically correct test cases generated by Comfort, about 18% of them triggered a runtime exception during testing due to semantic errors. Comfort also gives the best statement and branch coverages by ensuring most of the branches of a test program will be executed. Fuzzilli gives

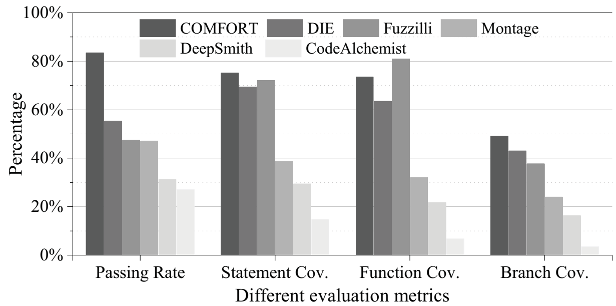


Figure 9. Compare to other fuzzers, COMFORT generates larger and more syntactically corrected test programs with a higher coverage of JS APIs and branches.

the best overall function coverage through hand-crafted function generation rules using seed programs, but it gives lower statement and branch coverage rates compared to COMFORT. While the test cases generated by Fuzzilli cover more functions than COMFORT, many of the statements and branches do not get executed during execution.

5.4 Bug Importance

It is reasonable to ask that if the bugs we found matter. It is not easy to provide a quantified answer to this question. Here, we follow the discussion from the source publication of CSmith [65]. Some of our reported bugs have been independently discovered and reported by application developers. This suggests we indeed report bugs that occurred in real-world applications. As of November 2020, 21 of the test cases generated by COMFORT have been added to the ECMA-262 official test suite because they triggered bugs in mainstream JS engines, of which 3 of the COMFORT-generated test cases were submitted by the relevant JS compiler vendor. This shows that the standard body and JS compiler developers appreciate the importance of the bugs triggered by COMFORT. Finally, most of our reported bugs have been confirmed and fixed by the developers, illustrating their relevance and importance (as it often takes substantial effort to fix a compiler defect); and the development teams of four tested JS engines marked a total of 25 of our bugs as a release-blocking priority for functional bugs. Furthermore, eight COMFORT-exposing bugs were welcomed and quickly fixed by the compiler developers in less than 48 hours.

5.5 Discussions and Future Work

Our study has been in the context of JS by utilizing the well-structured ECMA-262 specification to generate test data. We stress that COMFORT is not designed to replace existing fuzzers. Instead, our experiment shows that COMFORT can provide useful complementary test cases to discover conformance bugs by exploiting the pseudo-code-like definitions in ECMA-262. There are still some definitions were given in the natural language form in ECMA-262. These are not covered by COMFORT, and may still require expert involvement to understand to manually create test-case generation rules or write test cases to cover.

Like most supervised learning methods, our program synthesizer can suffer from insufficient training data. In which case, we expect the time in generating syntactically valid programs would remain largely unchanged, but the generated programs will be less diverse and, therefore, will affect test coverage and the ability for exposing bugs. However, we believe this is not an issue for popular programming languages like JS with a large open-source code base.

Our approach could be applied to other languages. Doing so would require the API semantics and expected behavior to be described in the pseudo-code form like ECMA-262. Our current implementation cannot exploit language specification written in free-form natural languages like the C standard. We view this as an exciting open challenge: can we transfer the decades' research in the natural language comprehension to translate language specifications to a form that can be exploited by a fuzzer? We have showcased that Transformer-like natural language processing models (GPT-2 used in this work) can be useful in generating validate test programs. Our language model can generate syntax correct test programs with a higher success rate than the state-of-the-art DL-based program generator. This approach is readily transferable to many other code generation tasks as the language model infers the language syntax and semantics directly from the training corpus. Like DeepSmith and many other JS fuzzers, COMFORT does not generate floating-point test programs. However, methods for testing floating-point programs [36] are orthogonal to our work.

6 Related Work

Random test case generation, in general, falls into two categories: program generation and program mutation. The former generates a test case from scratch while the later modifies existing test cases to expose anomalous behavior.

Program generation. Program generation often relies on stochastic context-free grammars. The fuzzer takes a grammar describing the syntax of the language being tested to produce syntactically valid programs whose various expressions conform to a given probability distribution of the grammar's productions. The Mozilla jsfunfuzz [52] was the first publicly available JS fuzzer. It uses a set of hand-crafted generation rules based on the JS grammars to generate test cases. Similarly, Domato [17] employs pre-defined templates to generate random web programs (include JS code) to test web browsers. Other works also use customized grammar-based constraint rules to generate test cases [7, 8, 18, 20, 32, 38, 40]. PHOG [8] can generate programs with rich contexts features with probabilistic context-free grammars. CSmith [65] is an effective program generator for generating C test programs using pre-defined grammar rules. Subsequent generators influenced by CSmith, like CLSmith [35], have extended random program generation to other programming languages. These approaches all require expert involvement and significant engineering effort to create the generation rules, grammars or templates. Comfort builds upon these past foundations by combining random program generation and language standard-guided test data generation to generate test cases for JS engines. Comfort reduces human involvement by leveraging deep learning to learn the language syntax and semantics from real-world code.

Program mutation. Mutation-based fuzzing modifies a set of seed programs to generate test cases [10, 27, 57, 64]. Equivalence modulo input testing is a representative mutation-based test case generation method [31]. Langfuzz [27] mutates test cases by inserting code segments that previously exposed bugs. SYMFUZZ [10] utilizes the white-box symbolic analysis to search for useful program inputs by mutating a seed program and a seed input together. IFuzzer [59] uses evolutionary algorithms to generate unusual input code fragments. AFL [67] and its subsequent works [9, 23, 34, 41, 51, 64] mutate the seed program to improve the test run coverage. Such techniques can be useful for Comfort in improving the test run coverage. DIE [49] and CodeAlChemist [24] are two mutation-based fuzzers for JS. DIE employs a stochastic process to preserve properties which are likely to expose bugs across mutations. CodeAlChemist breaks the seed programs into fragments and uses the fragments to assemble new JS test cases. By contrast, Comfort is a generative approach that does not require access to a set of seed program inputs. It is a pure black-box approach, requiring no source code, seed test programs, or other knowledge of the target compiler. AutoTest [43] is a contract-based random testing tool. It uses fuzzing techniques to test the conformance of the Eiffel program against the given contracts. It can expose bugs in the runtime system and the associated library implementation. By contrast, Comfort is designed to test compiler-specific implementations of APIs, non-API features, and compiler components like the parser, code generator, and optimizer. Nonetheless, extending Comfort to mutate bug-exposing test cases could be valuable.

Deep learning for compiler testing. Recently, deep learning (DL) models have been used for code modeling [5, 6, 15, 60, 66], random program generation [16, 56] and input fuzzing [19]. DeepSmith [14] and DeepFuzz [37] are two closely related works. Both approaches use the recurrent neural network (RNN), e.g., LSTM, to generate test programs. Montage [33] is a mutational JS fuzzer. It produces test cases by replacing the code snippets of the seed program's abstract syntax tree with a new code fragment generated by a LSTM model. Due to the limitation of RNN in capturing the long-term dependence of source code, they often generate many syntactically invalid programs that are rejected by the JS engines in the parsing stage. Our approach replaces the RNN generation model with a more advanced neural network, significantly improving the number of syntactically valid generated programs. None of the existing DL-based fuzzers has exploited the language specification to assist test case generation. Comfort is the first in doing so.

Conformance testing for JS. Our work was conducting concurrently with JEST [47] that also leverages differential testing and JS specifications to expose conformance bugs in JS compiler implementations. JEST utilizes JISET [48], a JS semantics extraction tool, to extract specification from the ECMA-262 document. Unlike Comfort, JEST is a program mutation approach that relies on a set of seed programs to create the JS test cases. Comfort thus has the advantages of not depending on the quality of the seed programs. Nonetheless, it would be interesting to extend Comfort to use the semantics extracted by JISET to perform differential testing.

7 Conclusions

We have presented Comfort, a novel compiler fuzzing framework for testing standard conformance bugs for JS compilers. Comfort leverages the recent advance in the deep-learning-based language model to automatically generate test JS programs without hand-crafted grammar or generation rules. It then augments the test programs with code and variables derived from the JS standard rules as test data to expose JS compiler bugs. We evaluate Comfort by applying it to test ten mainstream JS engines. In 200 hours of automated concurrent test runs, we found bugs in all the JS compilers we tested. At the time of submission, Comfort has discovered 129 unique, confirmed bugs, of which 115 bugs have been fixed by the developers, and 21 of the Comfort-generated test cases have been added into the official JS conformance test suite. Our work showcases a new way to leverage a structured language specification to produce bug-exposing test data to detect standard conformance bugs in compiler implementations, opening up an exciting research avenue.

Acknowledgements

We thank our shepherd, José Fragoso Santos, and the anonymous PLDI reviewers for their valuable feedback. We also thank Yang Tian, Houyou Yao, Xing Qu, Wen Yi and Yuan Wang for their help in verifying and submitting bug reports. This work was supported in part by the National Natural Science Foundation of China (NSFC) under grant agreements 61972314, 61872294 and 61902170, the International Cooperation Project of Shaanxi Province under grant agreements 2021KW-04, 2020KWZ-013 and 2021KW-15, an Ant Financial Science funded project and an Alibaba Innovative Research Programme grant.

References

[1] [n.d.]. ECMAScript 2020 Language Specification. http://www.ecma-international.org/ecma-262/.

[2] [n.d.]. GitHut 2.0: A SMALL PLACE TO DISCOVER LANGUAGES IN GITHUB. https://madnight.github.io/githut/#/pull_requests/2021/1

[3] [n.d.]. JSHint: A JavaScript Code Quality Tool. https://jshint.com/.

[4] [n.d.]. Test262: ECMAScript Test Suite (ECMA TR/104). https://github.com/tc39/test262.

[5] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*. OpenReview.net. https://doi.org/arXiv:1808.01400

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages (PACMPL)* 3 (2019), 1–29. https://doi.org/10.1145/3290353

[7] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* 52, 6 (2017), 95–110. https://doi.org/10.1145/3062341.3062349

[8] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: probabilistic model for code. In *Proceedings of the International Conference on Machine Learning (ICML)*. 2933–2942.

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506. https://doi.org/10.1109/TSE.2017.2785841

[10] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 725–741. https://doi.org/10.1109/SP.2015.50

[11] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. 180–190. https://doi.org/10.1145/2884781.2884878

[12] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Computing Surveys (CSUR)* 53 (2020). https://doi.org/10.1145/3363562

[13] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 197–208. https://doi.org/10.1145/2491956.2462173

[14] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 95–105. https://doi.org/10.1145/3213846.3213848

[15] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 219–232. https://doi.org/10.1109/PACT.2017.24

[16] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing benchmarks for predictive modeling. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 86–99. https://doi.org/10.5555/3049832.3049843

[17] Ivan Fratric. 2017. Domato. https://github.com/googleprojectzero/domato.

[18] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 206–215. https://doi.org/10.1145/1375581.1375607

[19] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59. https://doi.org/10.1109/ASE.2017.8115618

[20] Rahul Gopinath, Björn Mathis, Mathias Höschele, Alexander Kampmann, and Andreas Zeller. 2018. Sample-free learning of input grammars for comprehensive software fuzzing. *arXiv preprint arXiv:1810.08289* (2018). https://doi.org/arXiv:1810.08289

[21] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. 2013. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–10. https://doi.org/10.1109/CGO.2013.6494993

[22] Samuel Groß. 2018. *FuzzIL: Coverage guided fuzzing for JavaScript engines*. Ph.D. Dissertation. Master's thesis, Karlsruhe Institute of Technology.

[23] Tao Guo, Puhan Zhang, Xin Wang, and Qiang Wei. 2013. Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation. In *Proceedings of the 2th International Conference on Informatics & Applications (ICIA)*. IEEE, 212–215. https://doi.org/10.1109/ICOIA.2013.6650258

[24] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*. https://doi.org/10.14722/NDSS.2019.23263

[25] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 531–548. https://doi.org/10.1145/3319535.3363230

[26] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/NECO.1997.9.8.1735

[27] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security)*. 445–458. https://doi.org/10.5555/2362793.2362831

[28] Istanbul. 2015. JavaScript test coverage made simple. https://istanbul.js.org/.

[29] Diederik P. Kingma and Jimmy Lei Ba". 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*. https://doi.org/arXiv:1412.6980

[30] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2123–2138. https://doi.org/10.1145/3243734.3243804

[31] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 216–226. https://doi.org/10.1145/2594291.2594334

[32] Xuan-Bach D Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. 2019. SAFFRON: Adaptive grammar-based fuzzing for worst-case analysis. *ACM SIGSOFT Software Engineering Notes* 44, 4 (2019), 14–14. https://doi.org/10.1145/3364452.3364455

[33] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Fuzzer. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. USENIX, 2613–2630.

[34] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 475–485. https://doi.org/10.1145/3238147.3238176

[35] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2737924.2737986

[36] Daniel Liew, Cristian Cadar, Alastair F Donaldson, and J Ryan Stinnett. 2019. Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 521–532. https://doi.org/10.1145/3338906.3338921

[37] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 33. 1044–1051. https://doi.org/10.1609/aaai.v33i01.33011044

[38] Rupak Majumdar and Ru-Gang Xu. 2007. Directed test generation using symbolic grammars. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated software Engineering (ASE)*. 134–143. https://doi.org/10.1145/1321631.1321653

[39] Valentin Jean Marie Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2946563

[40] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 548–560. https://doi.org/10.1145/3314221.3314651

[41] Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning input tokens for effective fuzzing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 27–37. https://doi.org/10.1145/3395363.3397348

[42] Danny Merkx and Stefan L Frank. 2020. Comparing Transformers and RNNs on predicting human sentence processing data. *arXiv preprint arXiv:2005.09471* (2020). https://doi.org/arXiv:2005.09471

[43] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. 2009. Programs that test themselves. *Computer* 42, 9 (2009), 46–55. https://doi.org/10.1109/MC.2009.296

[44] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. 142–151. https://doi.org/10.1145/1134285.1134307

[45] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2016. RandIR: differential testing for embedded compilers. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala (Scala)*. 21–30. https://doi.org/10.1145/2998392.2998397

[46] OpenAI. 2019. GPT-2: 1.5B Release. https://openai.com/blog/gpt-2-1-5b-release/.

[47] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. JEST: N+ 1-version Differential Testing of Both JavaScript Engines and Specification. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*.

[48] Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu. 2020. JISET: JavaScript IR-based Semantics Extraction Toolchain. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 647–658. https://doi.org/10.1145/3324884.3416632

[49] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*. 1629–1642. https://doi.org/10.1109/SP40000.2020.00067

[50] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1, 8 (2019), 9.

[51] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*. 861–875. https://doi.org/10.5555/2671225.2671280

[52] Mozilla Security. 2007. funfuzz. https://github.com/MozillaSecurity/funfuzz.

[53] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*. 1715–1725. https://doi.org/10.18653/v1/P16-1162

[54] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 203–213. https://doi.org/10.1145/2884781.2884879

[55] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 361–371. https://doi.org/10.1145/3180155.3180236

[56] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural cnn decoder for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 33. 7055–7062. https://doi.org/10.1609/aaai.V33i01.33017055

[57] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery*. Pearson Education.

[58] Apache Tika v1.21. 2019. Apache Tika - a content analysis toolkit. https://tika.apache.org/.

[59] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 581–601. https://doi.org/10.1007/978-3-319-45744-4_29

[60] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2020. Combining Graph-based Learning with Automated Data Collection for Code Vulnerability Detection. *IEEE Transactions on Information Forensics and Security (TIFS)* (2020). https://doi.org/10.1109/TIFS.2020.3044773

[61] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. 724–735. https://doi.org/10.1109/ICSE.2019.00081

[62] Zheng Wang, Dominik Grewe, and Michael FP O'boyle. 2014. Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 4 (2014), 1–26. https://doi.org/10.1145/2677036

[63] Zheng Wang and Michael O'Boyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901. https://doi.org/10.1109/JPROC.2018.2817118

[64] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 511–522. https://doi.org/10.1145/2508859.2516736

[65] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. 283–294. https://doi.org/10.1145/1993498.1993532

[66] Guixin Ye, Zhanyong Tang, Huanting Wang, Dingyi Fang, Jianbin Fang, Songfang Huang, and Zheng Wang. 2020. Deep Program Structure Modeling Through Multi-Relational Graph-based Learning. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 111–123. https://doi.org/10.1145/3410463.3414670

[67] Michal Zalewski. 2014. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.