A Generative and Mutational Approach for Synthesizing Bug-Exposing Test Cases to Guide Compiler Fuzzing

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

Tianmin Hu
Northwest University, China
hutianmin@stumail.nwu.edu.cn

Zhanyong Tang*
Northwest University, China
zytang@nwu.edu.cn

Zhenye Fan
Northwest University, China
fanzhenye@stumail.nwu.edu.cn

Shin Hwei Tan
Concordia University, Canada
shinhwei.tan@concordia.ca

Bo Zhang
Tencent Inc., China
cradminzhang@tencent.com

Wenxiang Qian
Tencent Inc., China
saplasqian@tencent.com

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

ABSTRACT

Random test case generation, or *fuzzing*, is a viable means for uncovering compiler bugs. Unfortunately, compiler fuzzing can be time-consuming and inefficient with purely randomly generated test cases due to the complexity of modern compilers. We present ComFuzz, a focused compiler fuzzing framework. ComFuzz aims to improve compiler fuzzing efficiency by focusing on testing components and language features that are likely to trigger compiler bugs. Our key insight is human developers tend to make common and repeat errors across compiler implementations; hence, we can leverage the previously reported buggy-exposing test cases of a programming language to test a new compiler implementation. To this end, ComFuzz employs deep learning to learn a test program generator from open-source projects hosted on GitHub. With the machine-generated test programs in place, ComFuzz then leverages a set of carefully designed mutation rules to improve the coverage and bug-exposing capabilities of the test cases. We evaluate ComFuzz on 11 compilers for JS and Java programming languages. Within 260 hours of automated testing runs, we discovered 33 unique bugs across nine compilers, of which 29 have been confirmed and 22, including an API documentation defect, have already been fixed by the developers. We also compared ComFuzz to eight prior fuzzers on four evaluation metrics. In a 24-hour comparative test, ComFuzz uncovers at least 1.5× more bugs than the state-of-the-art baselines.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Compilers**; • **Computing methodologies** → *Artificial intelligence.*

KEYWORDS

Fuzzing, Historical bug, Guided testing, Deep learning, Compiler

ACM Reference Format:
Guixin Ye, Tianmin Hu, Zhanyong Tang, Zhenye Fan, Shin Hwei Tan, Bo Zhang, Wenxiang Qian, and Zheng Wang. 2023. A Generative and Mutational Approach for Synthesizing Bug-Exposing Test Cases to Guide Compiler Fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616332

1 INTRODUCTION

Compilers play a key role in software development [37]. Most application developers treat compilers as black boxes and have to trust the compiler-generated code. However, modern compilers are intricate software systems with large codebases consisting of hundreds of thousands of lines, and like many large-scale software projects, compiler bugs are inevitable and often manifest in the deployment environment [78]. Unfortunately, detecting compiler bugs can be challenging, yet their presence can significantly impede software development, leading to runtime crashes and even catastrophic consequences when applications are deployed [70, 78].

Automated test code generation technique - or fuzzing - has been a well-established and effective way to detect compiler bugs [24]. Compiler fuzzing techniques include generation-based [29, 85, 87] and mutation-based [25, 26, 50, 88] methods, which are typically used with differential testing [77, 91]. This is achieved by executing a randomly generated test program on multiple compiler test beds and observing the outputs of the compiler and executable binary. An anomalous behavior like compiler crashing, freezing, compilation timeout, or a binary execution result that deviates from the majority of the outputs indicates a potential compiler bug.

A fundamental challenge for fuzzing techniques is generating test cases that can quickly expose buggy behaviors [16]. Existing approaches typically employ a generation or mutation-based approach. Generation-based techniques construct test cases by using either manually designed grammar rules [44, 85], generation templates [89], or by restructuring code snippets extracted from a program seeds pool [40, 65]. In contrast, mutation-driven techniques leverage pre-designed rules to synthesize test cases [83].

*Z. Tang is the corresponding author. G. Ye and T. Hu are co-first authors.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616332

These fuzzing techniques have proven useful; however, they have a fundamental limitation when applied to compilers with a large code base. These techniques often rely on random test generation to achieve coverage, which is unlikely to be effective due to the uneven distribution of software bugs across components [78]. In practice, it is common for a few modules to account for most bugs [55]. Consequently, a random test generation approach may disproportionately allocate testing efforts to modules less prone to bugs. Therefore, a more efficient and effective strategy for compiler fuzzing should direct fuzzing efforts toward modules more likely to contain bugs. By focusing on these specific modules, we can maximize the impact of our testing efforts and increase the chances of uncovering critical vulnerabilities within a given test time budget.

We present ComFuzz, a new compiler fuzzing framework that combines generative and mutational techniques while improving the compiler testing efficiency. Unlike previous approaches that rely on random test case generation for achieving coverage, which is challenging in the context of compiler testing [29, 36, 88], ComFuzz is specifically designed to target compiler components that are more likely to contain bugs. To achieve this, ComFuzz leverages historical test programs obtained from Proofs-of-Concept (PoCs) of Common Vulnerabilities and Exposures (CVEs) and compiler test suites. Our generative approach is motivated by two key observations: bugs are prevalent in a small fraction of code within software [55, 78], and the fixing of historical defects often introduces new bugs [55]. Instead of solely using historical test cases as seed programs with a random mutation strategy to test compiler components uniformly, ComFuzz conducts intensive fuzzing tests on modules that have previously exhibited bugs. These targeted modules are known to be error-prone and can potentially contain bugs introduced by fixes.

To reduce the developer's efforts in building the test program generator, we harness the potential of deep-learning-based generative techniques [71]. Particularly, we employ a Transformer-based model [71] to infer features and constructs of the target programming language. To create test inputs (programs in our case), we use the historical code as seed input for our trained model, which then produces new test programs. Since these programs are derived from bug-exposing test cases, they are apt to embody specific features like standard library calls or language constructs while exhibiting new behaviors introduced by the program generator. Therefore, these generated programs can effectively guide the fuzzing efforts toward testing the error-prone components of the compiler.

Our approach is among the recent efforts of synthesizing test programs by reassembling the code ingredients extracted from the historical test cases [50, 91]. However, existing approaches failed to conduct high-intensity testing for a buggy compiler component due to the randomness of the assembled test cases. Our key conceptual insight is that there can be *residual bugs* in previously buggy modules, and we can leverage a set of bug-guided mutators to find these residual bugs. For example, to find residual API misuse bugs, we defined `SIM`, a mutator that replaces the original API with another similar API to expose more residual bugs. During each testing iteration, a bug-guided mutator is selected to mutate a test program that has been shown to expose anomalous compiler behaviors. We also designed five general-purpose mutators to improve the diversity of the generated test cases. We use general-purpose mutators to create new test codes in cases where the developed test programs fail to expose bugs or cannot improve the code coverage.

We evaluated ComFuzz on 11 JavaScript (JS) and Java compilers. In 260 hours of automated testing runs, ComFuzz reported a total of 33 unique bugs across nine tested compilers, covering 26 previously unknown bugs. Of the 33 submitted bugs, 29 have been confirmed, and 22 bugs – including an OpenJ9 document bug – have been fixed by developers. Our extensive evaluations show that ComFuzz is highly effective in generating bug-exposing test cases. Compared to eight state-of-the-art (SOTA) fuzzers [25, 27, 40, 50, 65, 86, 89, 91], ComFuzz uncovers at least 1.5× more bugs than prior methods.

In summary, this paper makes the following contributions:

- We propose a new compiler fuzzing technique by combining the historical test programs and bug-guided mutators, which can quickly cover the defective compiler components and achieve focused intensive testing;
- We present an extensible test generation scheme that can be easily ported to test compilers for other programming languages;
- We evaluated the effectiveness of ComFuzz by comparing it with SOTA fuzzers that utilize historical test cases for software testing.

2 BACKGROUND AND MOTIVATION

2.1 Compiler Testing and Challenges

Prior work for compiler testing includes generation-based [14, 33, 49, 85] and mutation-guided [22, 80, 84, 88] methods. While promising, prior methods still suffer from the following two challenges: 1) *How to generate bug-exposing test cases that can quickly cover the defective component of a compiler?* Although many methods have been devoted to generating *valid* test cases [14, 49, 53, 59, 89], all of them are randomly generated so that they fail to quickly locate the buggy components of a compiler during the early testing phase. Furthermore, a recent study stated that bugs in software do not conform to a uniform distribution, and only 40% of code will have bugs [55]. This means randomly synthesized test cases in prior work may be wasting most of the time testing the benign compiler modules. Even if a test case triggers a compiler bug, prior work cannot continuously test the buggy compiler module in adjacent iterative tests, which is bound to decrease testing efficiency. Thus, another challenge is *how to conduct focused and high-intensive testing for a buggy compiler component in adjacent testing iterations.*

Recent studies indicate that one major cause of bugs is the incomplete or incorrect repair of historical bugs [55, 91]. Meanwhile, bugs that are the same or similar to the historical ones often arise during software evolution [31, 39, 82]. Furthermore, recent work has shown that using historical test programs as seed test programs can improve fuzzing efficiency [40, 44, 50, 65, 91]. Inspired by these studies, we derived two key intuitions: (1) historical test programs (e.g., test suites or PoCs) can be used to generate bug-exposing tests for quickly covering buggy compiler components, and (2) buggy compiler components can be continuously and intensively tested by mutating the generated bug-exposing tests. These intuitions help to address the two challenges above and motivate our work.

Unlike prior work, our work aims to balance reusing existing code fragments from historical programs and exploring new program states to identify bugs. The key questions are: (1) how to obtain high-quality seeds and (2) how to leverage historical programs for

```
1  public class JVMTest {
2    /*--COMFUZZ-generated args via bug-guided mutators--*/
3      static int i = 999374098;
4      static int limit = 0;
5      static int[] arr = { -1 };
6      int[] test2(int i, int limit, int[] arr) {
7       /*--COMFUZZ-generated code segment via DL-model--*/
8          while (i++ != 0) {
9              if (arr[arr.length - 1] >= limit)
10                 break; }
11         return arr; }
12     public static void main(String[] args) throws Exception{
13         new JVMTest.test2(i, limit, arr); }}
```

Figure 1: A ComFuzz-generated test case, obtained from a historical test case and exposed a new bug of OpenJ9.

exploring new program states. Our approach differs from other DL-based methods by reusing historical tests as the model seed input for test program generation. We extract randomly cut-out code blocks to create a high-quality initial pool of seed programs. We then combine a generation model with carefully designed mutators to explore new program states and expose bugs.

2.2 Motivation Example

Figure 1 shows a bug-exposing test case generated by ComFuzz during iterative testing, which uncovers a new performance bug of OpenJ9. For this test case, OpenJ9 fails to enable JIT optimization for the main loop in lines 8–10. Specifically, this `while-loop` appears at the entry of the `test2` function. It has a `walk.bytecodePCOffset` of 0 in OpenJ9, which disables the JIT optimization of OpenJ9, leading to significant performance degradation. Specifically, OpenJ9 takes over 60 seconds to execute the test program, while other `JVM` engines like HotSpot take 7 milliseconds. This performance bug was fixed by the OpenJ9 developers.

ComFuzz generates this bug-exposing test program by setting the appropriate variables (lines 3–5) to manifest the performance bug. These variables are generated using our bug-guided mutator, whereas the code block (`test2` function in lines 6–11) is created from a historical bug-exposing test case [8].

ComFuzz achieves this by first building a DL-based program generator. Then, the learned generator is applied to produce new test cases by taking as input a randomly chosen seed generation header (e.g., line 6). Unlike prior work [29, 86], our seed generation headers are extracted from the historical test programs, e.g., JDK test suites and PoCs collected from CVE. Our goal is to focus on compiler components that are likely to contain bugs. The bug-triggering variables, e.g., `limit = 0` at line 4, are critical for manifesting this performance bug as using a small value makes the performance difference negligible. To generate such variables, we study the historical test cases to design `BOUN`, one of our five bug-guided mutators. Finally, ComFuzz assembles the generated program, the bug-triggering variable declaration statements, and the necessary startup function into a complete, executable test case.

2.3 Automated Program Generation

The rapid advances in deep learning (DL) promote the automated program modeling methods [11, 19, 93], which have been widely used in program-related tasks, such as code optimization [30, 64, 87], program generation [47], and vulnerability detection [52, 82]. Since no expert knowledge is required, many DL-based compiler fuzzers have been proposed for automated test program generation [29, 34, 41, 57]. Newer approaches [50, 86] subsequently employ more powerful neural networks to further improve the correctness of the generated test cases. Inspired by prior work, we use an advanced neural network to model programs, aiming to automatically generate test cases. Unlike existing DL-based fuzzers that randomly create test cases, we feed historical tests into the model to generate bug-exposing test cases, achieving a better bug-exposing ability.

3 OUR APPROACH

Figure 2 provides ComFuzz overview, which uses historical programs and bug-guided mutators for focused compiler testing. To this end, we establish a test program generator based on a pre-trained model [71] (Section 3.1). The built generator is used to generate bug-exposing test cases by feeding it with the historical test programs (Section 3.2). The generated test cases are applied to test target compilers through differential testing (Section 3.3), which keeps the *interesting* test cases that discover new compiler branches or trigger inconsistent differential outcomes and apply them to focused and guided tests using bug-guided mutators (Section 3.4).

3.1 Program Generator Construction

Most existing compiler testing approaches utilize domain-specific language models (e.g., grammar or template-based generators) to generate test programs. These methods require expert knowledge to design the grammatical rules or the generation templates, making these approaches hard to extend for new programming languages. Instead, our program generator is built upon a Transformer-based model [71], which is easy to generalize to other programming languages by feeding it with the target training samples.

Data collection and preprocessing. Since our program generator is built by fine-tuning a pre-trained model, the fine-tuning process requires massive training samples. To collect enough samples, we respectively scraped the top 10k open-source `JS` and `Java` projects ranked by stars hosted on GitHub. For each project, we extracted function-level code snippets as training samples. Specifically, our approach first removes all comments. Then, we extract the function-level code segments from the programs using parsers (i.e., Esprima [2] for `JS` and JavaParser [75] for `Java`). To improve the correctness of extracted code snippets, we also extract the expected global variables and insert them into the body of the code segments. Finally, we utilize syntax analysis tools (e.g., JSHint [1] for `JS` and JavaCompiler [4] for `Java`) to ensure the syntax correctness of the code segments and store them in a codebase.

Language model. Our language model is constructed based on a Transformer-based neural network [71]. It is essentially an encoder-decoder natural language generation model with a multi-head attention mechanism. Both the encoder and decoder are composed of a stack of six identical blocks. Specifically, an encoder block consists of a multi-head self-attention layer and a position-wise fully connected feed-forward layer. A decoder block consists of an encoder block and a multi-head attention layer. For each network layer, we employ a residual connection and a normalization layer.

Fine-tuning. The pre-trained model is refined using collected training programs. During the fine-tuning phase, we encode each training sample as a suitable vector for input into neural networks. To

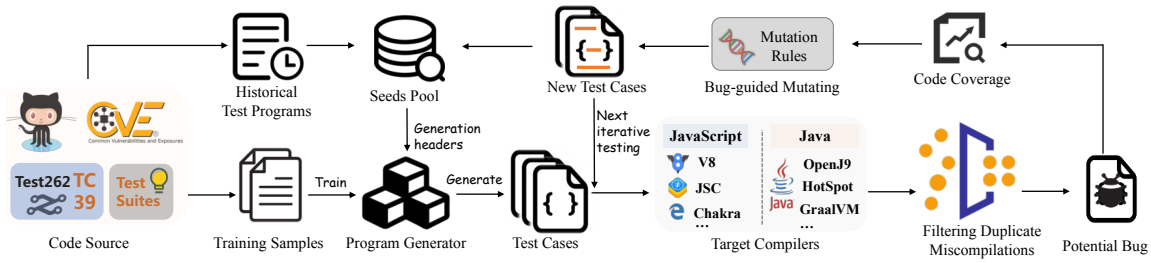


Figure 2: Overview of ComFuzz, which combines historical test cases and bug-guided mutators for focused intensive fuzzing.

do so, we employ Byte Pair Encoding (BPE) [74], a subword-based tokenization algorithm. BPE constructs a vocabulary dictionary by iteratively merging the most frequent pairs of characters or character sequences in a given corpus into subwords. This process ensures that each vocabulary item is represented as a subword based on its frequency in the training set. Using BPE, we create a vocabulary that captures common subword units in the corpus. When processing a training sample, we map each subword to an integer by referencing the vocabulary dictionary. This mapping allows us to transform the training sample (i.e., a code snippet in this work) into a sequence of integers. By collecting all the integer values associated with the subwords in the sample, we obtain a representation vector that effectively encodes the sample's information.

Given an inputting vector v, which is first fed into the pre-trained model to obtain the activation value of the last transformer block b_l^m. Then, it is passed through a classification layer to predict the next token. The process can be represented as follows:

$$P(y|v^1, ..., v^m) = softmax(b_l^m W_y) \quad (1)$$

where `softmax` represents the classification layer, y is the predicted token, and W_y is the weight matrix. Since the pre-trained model has billions of parameters, the former layers are language-independent features. Thus, the fine-tuning process only trains the last few layers. This is done by updating the weights of the last few layers while keeping others unchanged during fine-tuning. The objective is to maximize the following function:

$$\mathcal{L} = \sum_{v,y} logP(y|v^1, ..., v^m) \quad (2)$$

To accelerate model convergence, we fine-tuned our language model using the Adam optimizer [46] for 200 epochs. Re-training took around 40 hours using three NVIDIA GTX 3080Ti GPUs, which was a one-off cost. The hyperparameters we used include: `temperature=0.75, response length=500, Top P=9`, and others are default. Once trained, our language model can continuously generate test programs by feeding the seed generation headers.

3.2 Test Case Generation

Figure 1 shows that a test case contains three ingredients: (1) a `main` function (lines 12–13), (2) a test program (lines 6–11), and (3) its arguments (lines 3-5). We synthesized test programs by feeding the generation headers into a refined language model.

Generation header extraction. We feed the generation model with a seed code input (generation header) extracted from a historical test program (e.g., "`int[] test2(int i, int limit, int[] arr)`" in Figure 1). As the seed input determines the starting point of a test program, a good generation header plays an important role in generating bug-exposing programs. To obtain high-quality generation headers, we first collected as many historical test programs as possible. For `Java`, we collected the historical test programs from the test suites of HotSpot, OpenJ9, Kona, and GraalVM. For `JS`, we obtained the test programs from Test-262 [10], an official JavaScript language conformance test suite. All PoCs are collected from the CVE database. We then extract all function-level code blocks for each gathered historical test program by parsing it into an AST. The generation headers are extracted by randomly cutting off the former lines of the function-level code block. Note that we also collect 20k ordinary generation headers that are extracted from open-source projects for each programming language in order to increase the diversity of generated test cases.

Test program generation. The test program is synthesized by feeding the generation model with a randomly chosen seed generation header. During testing, the generation model first randomly selects a generation header from the seed pool, and it then yields the probabilistic vector of the next token (e.g., a subword-based token encoded by BPE). Differ from natural language generation tasks that output the token with the highest probability, we employ a Markov chain Monte Carlo (MCMC) algorithm [32], a probabilistic sampling scheme where a token with a higher prediction probability is more likely to be chosen to sample the next token. This process can improve the diversity of the generated test programs. Next, the generated token is appended to the original generation header, which is fed to the generation network to produce the next token repeatedly. This synthesized process terminates when the generation network produces the termination symbol "<EOF>" or a bracket '}' that indicates the end of a program method or exceeds the maximum token length ϵ. Here ϵ is set to be 6,000.

Arguments generation. In compiler testing, a high-quality test case not only contains a bug-exposing test program but also contains the arguments expected by the test program. To synthesize the desired arguments, we designed several heuristic rules for all basic data types, e.g., Integer, Double, String, Array, Object, etc. For `Java`, the variable type is determined according to the parameter type; for `JS`, the argument types are inferred using an existing work [86].

3.3 Differential Testing

We employ the established differential testing mechanism [61] to expose compiler bugs. A majority voting scheme is utilized to capture the anomalous compiler behaviors which are yielded by the minority compilers. The anomalous compilations need to be further confirmed manually after filtering duplicate miscompilations.

Anomalous compiler behaviors. A compiler typically consists of three components: a parser that checks if a program is correctly

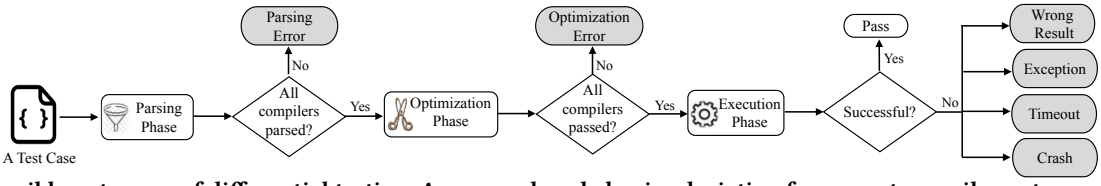


Figure 3: Possible outcomes of differential testing. An anomalous behavior deviating from most compiler outcomes indicates a potential bug. Our current do not consider "Wrong Result" and "Timeout" at the parsing and optimization stages because it is hard to establish an oracle for the intermediate results and attribute timeout to intermediate compilation stages, respectively.

coded without syntax and semantic errors; an optimizer that aims at optimizing code at a high-level intermediate representation (IR), and a generator (also known as backend) which is responsible for translating the IR into binary code. Thus, the incorrect implementation of any of the aforementioned three components may produce an anomalous behavior, indicating a possible compiler bug. This happens when the outcome for a given test input compiled using a compiler differs from the outcome from most of the tested compilers for the same input, e.g., wrong result, exception, timeout, and crash. These anomalies can manifest at various stages, including parsing, optimizing, and runtime. However, ComFuzz does not detect "Wrong Results" and "Timeout" at the parsing and optimization stages because (1) the intermediate outcome during parsing and optimization is typically implementation-dependent, and (2) it is hard to attribute compilation timeout to individual stages since we treat the tested compiler as a black box. Figure 3 shows seven potential outcomes when executing a test case. All outcomes (except for the "Pass") represent anomalous behaviors that necessitate subsequent manual analysis. A "Pass" outcome signifies that all tested compilers yield identical outcomes without any abnormal behavior. As this outcome aligns with the expected result, it is disregarded.

Suppose the test case does not trigger any anomalous outcomes during the parsing and optimization phases, the compiler backend proceeds to translate the optimized code into machine instructions to be executed on the tested platform. However, when executing the compiled binary, there are potential scenarios where four distinct anomalous behaviors may arise during runtime. Firstly, a "Wrong Result" occurs when the binary produces an output that deviates from the outputs generated by most tested compilers. This discrepancy indicates an inconsistency or error within the compiled binary. Secondly, an "Exception" is encountered when the execution of the binary results in a thrown exception, while the execution given by other compilers does not exhibit this behavior. This points towards an exception-handling flaw. Thirdly, a "Timeout" occurs if a program fails to terminate within the specified time limit, while binaries generated by other compilers terminate before the time limit. This usually indicates an optimization bug, leading to prolonged execution times. Lastly, a "Crash" can manifest if the binary itself or the compiler (e.g., for interpret execution mode) crashes during execution. This occurrence suggests a potential compiler bug.

Identifying anomalous behaviors. Among six types of anomalous behaviors, `Crash` and `Timeout` are of immediate interest, indicating the erroneous compiler implementation. Following the practices in prior work [25, 40], we set the timeout threshold for runtime execution to 30 seconds. We consider an erroneous behavior occurs when any binary given by a compiler has an execution time exceeding 30 seconds, whereas binaries generated by other

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 1
    at java.lang.AbstractStringBuilder.deleteCharAt(AbstractStringBuilder.java:824)
    at java.lang.StringBuilder.deleteCharAt(StringBuilder.java:253)                    HotSpot
    .....
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: index 1, length 1
    at java.base/java.lang.String.checkIndex(String.java:3278)
    at java.base/java.lang.AbstractStringBuilder.deleteCharAt(AbstractStringBuilder.java:916)
    at java.base/java.lang.StringBuilder.deleteCharAt(StringBuilder.java:297)          OpenJ9
    .....
```

Figure 4: Differentiated anomalous behaviors that HotSpot and OpenJ9 threw, which indicate the same `JVM` exception.

compilers for the same input complete their execution within 30 seconds. In our evaluation, we do not encounter any false positives when using this threshold, so we do not find it beneficial to increase the threshold. For the other four anomalous behaviors, a majority voting scheme is applied to identify if a compiler contains potential defeats by comparing the compilation and execution results.

Since all compilers may not have the same error or exception messages, directly comparing their outcomes can result in a high false positive rate. Figure 4 shows one of such examples, where both HotSpot and OpenJ9 threw an `Out-of-Bounds` exception with the same language semantics but different messages (highlighted with a dark background). If we were to compare the contents directly, this would mistakenly be categorized as two distinct types of differential behavior. We propose using a *key information extractor* to minimize false positives. The extractor first eliminates compiler-specific implementations, such as the location or variable-related information from the stack trace generated by the target compilers. It then extracts the essential information, such as the exception type and affected APIs (highlighted in bold red font in Figure 4), and stores them in an unordered list for each compiler output. Lists with the same elements indicate the same anomalous compiler behavior.

In addition, the extracted key information is also used to filter out duplicate mis-compilations. Specifically, we extended the tree-based classifier proposed by Comfort [86] to build our filter. Unlike Comfort, which consists of three decision layers, our augmented filter adds a new layer at the second layer of Comfort. The decision nodes in the new layer correspond to standard exit codes (also known as return codes) that are returned by the operating system.

3.4 Mutation for Focused Testing

Test case mutation is a powerful way to improve code coverage. Prior mutational approaches [50, 88] randomly choose pre-designed operators to mutate the *interesting* test cases, incurring extraordinarily costly and time-consuming testing. This is because the random mutants fail to focus on testing a specific compiler component in successive fuzzing. To do so, we designed several bug-guided mutators to generate new bug-exposing test cases by mutating the *interesting* test case that has triggered the anomalous behaviors.

Algorithm 1 Mutator Scheduling Policy

Input:
t_{prog} // A test program needed to be mutated
OM // The collection of bug-guided mutators
GM // The collection of general-purpose mutators
Output: P_{new} // A list that stores the mutated test programs

Let $OM \leftarrow$ {"SIM", "VUL", "$INSL$", "$SNIP$", "$BOUN$"} ;
Let $GM \leftarrow$ {"$REPO$", "$GENP$", "$CONF$", "$INSC$", "DEL"} ;
Let P_{new}, M be the empty lists
$t_{ast} \leftarrow parsingToAST(t_{prog})$;
if t_{prog} is an interesting test case **then**
  $M \leftarrow identifyMutators(t_{ast}, OM)$;
else
  $M \leftarrow identifyMutators(t_{ast}, GM)$;
end if
$m \leftarrow selectCandidateMutators(M)$;
$t_{new} \leftarrow mutation(t_{prog}, m)$;
$P_{new}.append(t_{new})$;
return P_{new};

Mutation operators. We designed ten kinds of mutation operators, including five bug-guided and five general-purpose mutators. All our mutators can be found at [7]. To obtain a set of bug-guided mutators for finding residual bugs, we refer to the existing literature on frequently-occurring bugs [12, 50, 73]. This leads to five bug-guided mutators representing five common classes of bugs (i.e., API misuse, security, performance, incomplete bug fixes, and missing boundary check). The bug-guided mutators aim to produce new bug-exposing test programs based on *interesting* test cases for focused testing, whereas the general-purpose mutators are applied to improve the diversity of the mutant programs to avoid convergence during the testing process. We describe bug-guided mutators below:

- **Similar API Replacement (SIM):** Replace an original API call with one that has similar semantics or the same return values. This mutator is inspired by prior work on API misuse [12].
- **Vulnerability Rules (VUL):** Mutate the target test case with vulnerability rules manually designed according to PoCs.
- **Insert Loop Statement (INSL):** Insert Loop Statement (e.g., `for`, `while`) into the target test program. This is motivated by a prior empirical study on performance issues [73].
- **Snippet Replacement (SNIP):** Replace a basic code block with a structurally-similar one. This is inspired by prior work [50] that observed that more than 95% of code fragments overlap between the historical test programs due to incomplete bug fixes.
- **Boundary Values (BOUN):** Generate boundary values (e.g., `0`, `0XFF`, `NULL`) for arguments passed to the function calls.

Given an *interesting* test case, mutator **SIM** is responsible for replacing an existing API with a new one with similar functions or the same types of return values. For example, the `Java` function `lastIndexOf(String str)` will be replaced with the similar method `lastIndexOf(String str, Int fromIndex)`. Such mutation is able to reach deeper code branches of the method `lastIndexOf()` and continuously test the `String Class` of `JVM`. For the `JS` language, there are also many similar-semantic function calls, such as `String()` v.s. `toString()`. Similar APIs are automatically collected using a script to parse the language specification document. **VUL** aims at mutating the test case by using pre-designed vulnerability patterns. Specifically, we implemented three patterns that cover three types of vulnerabilities, including CWE-1321, CWE-915, and CWE-843. We chose them because they are the top-3 most severe vulnerabilities (we count the severity by calculating the number of relevant CVEs labeled as HIGH or CRITICAL to the total number of CVEs). The first pattern is about prototype pollution vulnerability, which is achieved by modifying the prototype chain attributes through `__proto__` or `Object.setPrototypeOf` and declaring a new object accordingly. The second pattern is related to the remote code execution vulnerability, which applies the `getter/setter` or `__defineGetter__()` and `__defineSetter__()` to modify the attributes of the target objects. The third pattern about type confusion vulnerability replaces a function call with multiple calls, meanwhile changing the object type. The test cases that conform to any one of the vulnerability rules will be mutated. We would like to note that the above two mutators **SIM** and **VUL** are language-specific, and they need to mutate the *interesting* test cases according to the program context of a specific programming language. This, we think, is inevitable due to the nature of different programming languages.

INSL operator inserts the loop statement, e.g., `for` or `while`, into the test case to enrich the control dependencies of the mutant program. This operator creates a hot code region to activate the just-in-time compilation module of tested compilers for exploring the performance issue. The **SNIP** operator replaces a basic code block in the original test program with a similar one. To do so, we first extract the code blocks from gathered programs, and each code block is a complete fragment (e.g., `if` statements) that is recorded with a *block assembly constraint*, which is represented as a tuple: `<pre-constraint,post-constraint>`. Differ from pre/post-condition in Hoare logic [42], the `pre-constraint` marks its required variables or statements, and the `post-constraint` labels the return values that are required to be defined to execute the code block without a runtime error. The collected code blocks with their *block assembly constraint* are stored in a JSON file. The **SNIP** operator will first select the code blocks with expected *block assembly constraint* from the JSON file and then randomly select a code block to replace the original one. This aims to cover deep branches for tested components. The last operator **BOUN** is to generate the boundary values for the test case. For the Java test program, we defined 23 boundary values such as `0`, `1`, `-1`, `NaN`, `NULL`, `0xFFF`, and `Undefined`, etc., which are from the historical test cases that exposed compiler defects or vulnerabilities. For the `JS` program, we utilize Comfort [86] to generate the boundary values according to the ECMA-262 specification. This operation can continuously test an API and cover its deeper code branches. To increase the diversity of the test cases, we also designed five general-purpose mutation operators. They are described as follows:

- **Replace Operator (REPO):** Randomly replace a binary or a unary operator with another corresponding one. e.g., replace "--" with "++", or replace "+" with "-".
- **Generate Parameters (GENP):** Randomly generate parameters with primary and reference types.
- **Change Control Flow (CONF):** Change the control flow by replacing a conditional statement, e.g., replace `if` with `switch`.
- **Insert Conditional Statement (INSC):** Insert conditional statements (e.g., `if...else`) into the original test case.

Table 1: Target compilers we have tested.

Category	Compilers	Versions	Build No.	Release Date
JS	V8	v9.9.1	8a05d7a	Dec. 2021
	ChakraCore	v1.11.24	a75335b	Dec. 2020
		v1.13.0.0-beta	418a27c	Jun. 2022
	SpiderMonkey	C96.0	fd8da16	Jan. 2022
	JavaScriptCore	v286936	-	Dec. 2021
	GraalJS	v21.3.0	ede7e2b	Oct. 2021
	JerryScript	v3.0.0	6fe763f1	Sep.2022
	Hermes	v0.10.0	7d3e091	Aug. 2022
	QuickJS	v2021-03-27	b5e6289	Mar. 2021
JVM	OpenJ9	v8u332-b02	1e4e4ae	Feb. 2022
		v11.0.15+1	c902226	Feb. 2022
	HotSpot	V8.0.332	e1f6c13	Feb. 2022
		V11.0.14	b8cdf1a	Jan. 2022
	GraalVM	v22.0.0.2	bd6570e	Jan. 2022
Total	**11**	**14**	**14**	-

- **Delete Code Snippet (DEL):** Randomly delete a basic code block from the original test case.

The general-purpose mutators change the data and control dependencies that significantly deviate from the original programs to guide towards testing more uncovered compiler components. Specifically, mutators **REPO** and **GENP** change the data dependencies while **CONF** and **INSC** alter the control dependencies. The mutator **DEL** can change both the data and control dependencies.

Mutator scheduling policy. Algorithm 1 presents our mutation scheduler. The scheduler takes in the mutators and the test program required to be mutated, producing a list of new mutated test cases. Given a test program t_{prog}, our scheduler first determines if it is an *interesting* test program. Here the *interesting* test programs refer to those that have ever triggered anomalous compiler behaviors or discovered the new branches of the tested compilers. Our insight is using both code coverage and anomalous compiler behaviors as guidance can help to discover more new code branches. If t_{prog} is an *interesting* test case, the scheduler then identifies which bug-guided mutators are suitable for mutating (lines 5–6); otherwise, the apposite general-purpose mutators will be chosen (lines 7–8). To determine M, we first parse t_{prog} into an AST (line 4) and search the potential mutation positions by traversing the parsed AST. A mutation position essentially refers to an AST node whose context program conforms to the pattern of any of our ten mutators. For example, the `test2` function in Figure 1 expects two parameters of type `integer`, which can be generated by the bug-guided mutator `BOUN`. Note that the mutator determination process may produce multiple mutators. The scheduler will randomly choose no more than $M.MAX$ mutators to generate new test programs (lines 10–12).

4 EXPERIMENTAL SETUP

Target Compilers. We apply ComFuzz to test JS and JVM compilers. Table 1 lists the tested compilers and the versions used. Specifically, we apply ComFuzz to 8 JS and 3 JVM compilers using their latest trunk branches. In total, we have tested 14 target compiler-version configurations.

Table 2: Statistics for exposed bugs per target compiler.

Compiler	#Submitted	#Confirmed	#Fixed of #Conf.
ChakraCore	5	4	4
SpiderMonkey	2	1	1
GraalJS	3	3	3
JerryScript	4	3	1
Hermes	3	3	1
QuickJS	1	0	0
OpenJ9	12	12	10
HotSpot	2	2	1
GraalVM	1	1	1
Total	**33**	**29**	**22**

Competitive Baselines. We choose eight prior methods, covering both generation- and mutation-based fuzzers for compiler testing. Specifically, we compare ComFuzz with four fine generative fuzzers: Comfort [86] and PolyGlot [27] for the JS engine; JavaTailor [91] and JAttack [89], the latest two test program synthesizers for JVM testing. We also compare ComFuzz with four mutational fuzzers: CodeAlchemist [40], DIE [65], and Montage [50] for JS engines as they represent the SOTA methods; and Classming [25] for JVM.

Implementation and Evaluation Platforms. Our program generator is built upon a Transformer architecture [71] in PyTorch v1.11.0. Our mutators are implemented in JS and Java. The differential testing engine is written in Python. Our evaluation platform is a multi-core server with a 3.6GHz 8-core (16 threads) Intel Core i7 CPU, four NVIDIA GTX 3080Ti 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 GPUs.

5 EXPERIMENTAL RESULTS

5.1 Bug Summary

This subsection exhibits the number of identified bugs and presents their various summary statistics for the purpose of evaluating the ability of ComFuzz to discover previously unknown bugs. The experiment started with testing JS engines first in November 2021 and then extending our testing framework to JVM in April 2022. The total testing time is about 260 hours on approximately 400k test cases for JS and 200k test cases for Java that are generated from around 20k historical test programs.

Number of Bugs. Table 2 gives the distribution of the ComFuzz-exposed bugs according to the tested compilers. ComFuzz discovered bugs in all the tested compilers except for V8 and JavaScriptCore. Among the confirmed bugs, we found a total of three unique bugs exposed by the same test cases. Listing 3 shows one of these bugs. This implies that bugs are prevalent in different compilers. Overall, as of February 2023, we have reported 33 unique bugs. To date, 29 bugs have been confirmed, of which 22 have been fixed by the developers. For the remaining four reported JS bugs, two bugs were rejected due to the special design of the compiler; the other two bugs are waiting to be verified. In addition to the above four bugs, three more bugs were silently fixed in the beta version of the tested compilers after submitting our bug reports. In total, ComFuzz exposed 26 out of 33 bugs that were previously unknown.

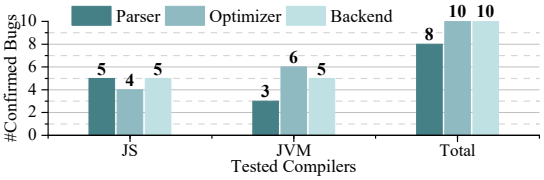


Figure 5: #Confirmed bugs per compiler component.

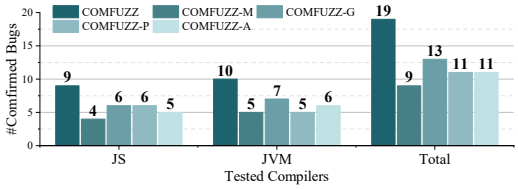


Figure 6: #Confirmed bugs found by ComFuzz variants.

It is worth mentioning that for OpenJ9, ComFuzz found 12 bugs, far more than the number of bugs exposed in other compilers. This is mainly because OpenJ9 introduces many optimization schemes that are prone to defects due to incorrect implementation. Here five such bugs were found by ComFuzz in the optimizer of OpenJ9.

Affected compiler components. As discussed in Section 3.3, a compiler is composed of a parser, an optimizer, and a backend. Each of the three components inevitably has defects due to wrong implementations. To assess how ComFuzz performs in covering these three components, we grouped the ComFuzz-discovered bugs into three categories: `Parser`, `Optimizer`, and `Backend`, according to the phase where the bug is caused. Figure 5 gives the number of confirmed bugs discovered by ComFuzz for each component. Note that the OpenJ9 document bug (see Listing 4) is excluded from Figure 5. For `JS` engines, ComFuzz exposed around 4–5 bugs in the three components, indicating that bugs are prevalent in different components of a compiler. For `JVM`, the most error-prone component is `Optimizer`, which has exposed 6 bugs, followed by the `Backend` and `Parser`. Overall, bugs in `Optimizer` are prevalent - 6 `JVM` and 4 `JS` bugs belong to this group. According to the developers' feedback, this is often due to erroneous implementation of the optimization schemes. This is in line with the current research trend that mainstream compiler vendors are striving to enhance the optimization level and depth.

5.2 Ablation Study

Recall that ComFuzz consists of three components: (1) a generation model that leverages the historical test programs (see Section 3.2); (2) the bug-guided or (3) general-purpose mutators that mutate interesting test cases (see Section 3.4). To illustrate how they perform in bug-exposing capability, we evaluate their effects in ComFuzz with an ablation study. In ComFuzz-M and ComFuzz-A, we removed the mutation and generation part and kept other modules unchanged, respectively. Likewise, in ComFuzz-G and ComFuzz-P, we respectively remove the general-purpose mutators and bug-guided mutators and keep other components. We compared ComFuzz with all four variants with a test time budget of 48 hours. All the variants are evaluated using the same seed programs to avoid the test bias caused by the randomness of the test case generator.

Figure 6 reports the number of bugs discovered by each implementation variant. ComFuzz-M discovered four bugs for `JS` and five bugs for `JVM`, respectively. This confirms that our generation model is effective in generating bug-exposing test cases. Furthermore, ComFuzz-A respectively exposed five and six bugs for `JS` and `JVM`, indicating the effectiveness of our mutation strategy alone. By augmenting the generation model with bug-guided mutators, ComFuzz-G improves ComFuzz-M by exposing four more bugs. Likewise, comparing ComFuzz-P with ComFuzz-M, we can see that with general-purpose mutators, ComFuzz-P discovered two more bugs, suggesting a better bug-exposing capability. This indicates the usefulness of our bug-guided and general-purpose mutators in augmenting our generation model for exposing compiler bugs. The ablation study also shows that compared to its variants, ComFuzz achieves the best performance by giving at least 1.5× improvements in bug detection. This indicates the effectiveness of ComFuzz in combining the generative and mutational techniques.

5.3 Bug Examples

ComFuzz is capable of finding diverse kinds of bugs on tested compilers according to the historical test programs and bug-guided mutators. To provide a convincing glimpse of the diversity of the exposed bugs, we give four ComFuzz-generated test cases that expose the `JS` and `JVM` compiler bugs.

```
1  public class JVMTest {
2    public static void main(String[] args) throws
         Exception {
3        StringBuilder sb = new StringBuilder();
4        sb.append('J');
5        sb.deleteCharAt(1);
6        System.out.println(sb.toString());}}
```

Listing 1: OpenJ9 for JDK 8 fails to throw an `OutOfBound` exception for this test code.

OpenJ9 Parser bug. The bug-exposing test code in Listing 1 is a program that throws an `OutOfBounds` exception because it deletes an element that does not exist at line 5. When executing this test code, OpenJ9 for JDK 8 fails to throw an exception. The root cause is that OpenJ9 incorrectly returns the boundary value when calling the inner function `StringBuilder delete (int start, int end)`. This bug-exposing test code is mutated via our bug-guided mutator **SIM** (see Section 3.4) by replacing the `delete()` function with the `deleteCharAt()` function. This bug was quickly confirmed and added to the repair list for the next release version.

```
1  public class JVMTest {
2      static void foo() {
3          Integer i = 100;
4          do {
5              return;
6          } while (i++ < 10); }
7      public static void main(String[] args) {
8          foo();     }}
```

Listing 2: HotSpot for JDK 8 throws an AssertionError exception while the code is correct.

HotSpot Backend bug. The tested HotSpot for JDK 8 threw an AssertionError exception when executing the test case shown in Listing 2. As the test code is syntactically correct, HotSpot should compile the code successfully and transform the test code into a bytecode. The root cause of this bug is that the HotSpot backend incorrectly maps flat `do...while` loop statements into the bytecode.

ComFuzz produces this bug-exposing test program by applying the bug-guided mutator **INSL** to insert the `do...while` loop statement into the body of the `foo` function at lines 4-6. The bug has been confirmed and assigned for repair.

```
1  var foo = function (t1) {
2      t1.sort(function (a, b) {
3          return a - b; });
4      print(t1); }
5  var Parameter1 = ['a','b'];
6  foo(Parameter1);
```

Listing 3: It triggers an optimizer bug of SpiderMonkey.

SpiderMonkey Optimizer bug. This test program in Listing 3 contains a `sort` function that invokes an inline comparison function (at line 2), which is synthesized via replacing the original body of the `foo` function with `sort` function call[1] by using our bug-guided mutator **SNIP** (Section 3.4). The correct outputs of this test program should be "a, b" because the JS specification, ECMA-262 states that the `sort` function should return the original array `t1` (Here is "a, b") when the value of the statement `a-b` in the inline comparison function equals `NAN`. While SpiderMonkey yields "b, a", the wrong results. The root cause is that SpiderMonkey misused a specific optimization scheme for the comparison function instead of actually calling the comparator function, leading to an incorrect result. This bug is immediately verified and classified as `P1` priority - the most urgent level that should be fixed soon, as Bugzilla states. Moreover, a similar test case also exposed a confirmed bug of JerryScript.

```
1  public class JVMTest {
2    boolean testLatin1() {
3      try {
4          StringBuilder sb = new StringBuilder();
5          System.out.println(sb.capacity());
6          sb.ensureCapacity(Integer.MAX_VALUE/2+1);
7        } catch (OutOfMemoryError oom) {
8          oom.printStackTrace();       }
9        return true; }
10   public static void main(String[] args) throws
         Exception {
11       new JVMTest().testLatin1();  }}
```

Listing 4: The test program that revealed the incomplete documentation problem in OpenJ9.

Incomplete OpenJ9 documentation. The ComFuzz-generated test program presented in Listing 4 triggered an anomalous behavior of OpenJ9 for JDK 11 (i.e., catching an `OOM` exception). Such behavior is expected because the `-XX:+CompactStrings` option responsible for causing this exception is disabled by default in OpenJ9 for JDK 11, while it is enabled in HotSopt and OpenJ9 for JDK 8. During the manual analysis, we found that the OpenJ9 documentation had no description for this option. We have reported this defect to the developer, and it was fixed quickly after reporting.

5.4 Evaluation of Differential Testing

To quantify the role of our differential testing module, we count the number of confirmed bugs exposed by ComFuzz and divide them into either *crash* (bugs that lead to a runtime crash) or *inconsistency* (bugs discovered by differential testing). As shown in Table 3, approximately 85% of the bugs were uncovered due to inconsistency, showing the importance of employing differential testing.

[1]The `sort` function is originated from the link [9].

Table 3: The number of ComFuzz-uncovered bugs found by execution crashes or differential testing.

Compilers	#Crash	#Inconsistency	Ratio of Inconsis.
JS	3	15	83.3%
JVM	2	13	86.7%
Total	**5**	**28**	**84.8%**

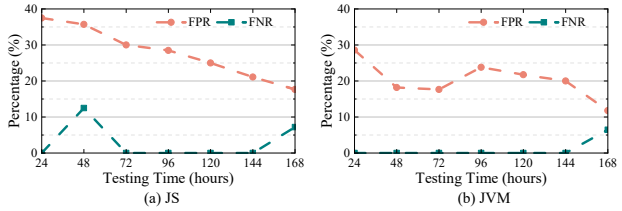


Figure 7: How the false negative rate (FNR) and false positive rate (FPR) change as we increase the testing time.

Recall that our differential testing methodology incorporates a filtering mechanism designed to duplicate mis-compilation behaviors. We consider two metrics to assess the filter's effectiveness: the false positive rate (FPR) and the false negative rate (FNR). In this context, a *false positive* refers to the number of cases mistakenly classified as bugs, while a *false negative* represents the number of actual inconsistent results that were erroneously filtered. Figure 7 shows how FNR and FPR change throughout the testing process. We observe that throughout the entire testing period, the FPR remains consistently low (below 10%). As the testing progresses, we see a gradual decrease in the FNR of our filtering mechanism, showing its increasing efficiency in accurately identifying inconsistent results.

5.5 Compare to Prior Compiler Fuzzers

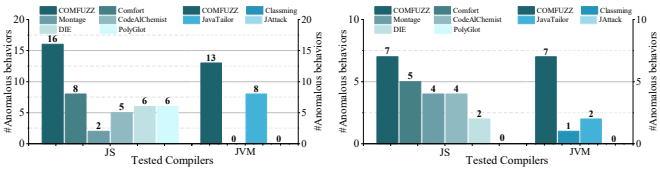
We use the following metrics to compare ComFuzz against eight baselines [25, 27, 40, 50, 65, 86, 89, 91] introduced in Section 4:

Bug exposing capability. This metric quantifies the number of anomalous compiler behaviors. Note that we have checked and removed all the duplicate anomalous behaviors; hence, each of them indicates a potential compiler bug that needs to be verified by developers. For a fair comparison, we tested each fuzzer for 24 hours of consecutive testing runs using the ComFuzz's seed programs and seed programs from the baselines, respectively.

Syntax passing rate. It measures the ratio of the generated test cases that are syntactically correct. For each fuzzer, we leverage 50k-generated test cases to compute the syntax passing rate.

Code coverage. We use three widely used coverage criteria: statement coverage, function coverage, and branch coverage for the comparison. To collect the coverage information, we use Gcov [3] and Lcov [5] for `JVM`, and llvm-cov [6] for `JS` engine, the code profiling tools for instrumenting C code in `JS` and `JVM` compilers.

Throughput. Following the practices in [13, 92], we compute the fuzzing throughput by measuring the number of test cases processed per minute. This is computed by applying each fuzzer to the same 10k test cases using their default settings.



(a) w/ ComFuzz's seed programs (b) w/ baselines' seed programs

Figure 8: Anomalous behaviors found by different fuzzers.

Figure 9: Comparison results of syntax passing rate.

5.5.1 Bug-exposing capability. Figure 8 shows that ComFuzz exposed more unique anomalous behaviors than other individual fuzzers, either using ComFuzz's or baselines' seed programs. With ComFuzz's seed programs, ComFuzz discovered 16 anomalous behaviors for target JS engines, achieving an average improvement of 270% than the number of anomalous behaviors discovered by other fuzzers. For JVM, ComFuzz found a total of 13 anomalous behaviors, 1.5× over the number of anomalous behaviors found by JavaTailor. Among all 29 anomalous behaviors discovered by ComFuzz, 15 were found by the test cases generated from historical test programs, and 6 were discovered by our bug-guided mutators. Likewise, ComFuzz exposed a total of 14 anomalous behaviors with baselines' seed programs, also achieving a 1.5× more than that of other baselines. This demonstrates ComFuzz's bug-exposing capability.

5.5.2 Syntax passing rate. Figure 9 shows how many automatically generated test programs can pass the syntax checks. ComFuzz gives an average passing rate of 82%, achieving a 10% improvement over most alternative methods. Among the syntactically incorrect test cases generated by ComFuzz, nearly 90% of them were created by general-purpose mutators, which are error-prone as they randomly mutate the test cases without any syntax guidance. In contrast, JAttack and JavaTailor applied well-designed grammatical rules to synthesize test cases, reaching higher passing rates of 100% and 98.4%, respectively. However, the grammatical rules limit their bug-exposing abilities. As we show in Section 5.5.1, ComFuzz discovered at least 1.5× more anomalous behaviors than any of the baselines.

5.5.3 Code coverage. Figure 10 presents the comparison results of code coverage, where ComFuzz gives the best statement and branch coverage compared to all evaluated fuzzers. The results demonstrate that using historical test programs for generating test cases is more helpful in covering deeper code of the tested compiler. For the JS engine, Montage and CodeAlChemist achieve higher function coverage than ComFuzz. The reason is that their seed programs cover more JS functions, but they give a lower statement and branch coverage than ComFuzz due to the low syntax passing rate of their generated tests. This also illustrates Montage and CodeAlChemist have lower bug-exposing capabilities than ComFuzz.

5.5.4 Throughput. Table 4 compares the fuzzing throughput computed as the number of test cases processed per minute. We ran

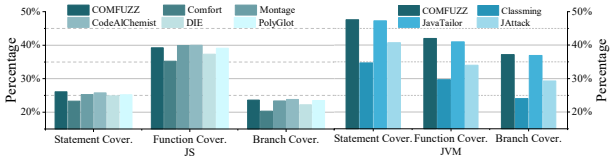


Figure 10: Compared results of code coverage.

all fuzzers with the same 10k test cases and then calculated the throughput. Compared to other fuzzers, Classming takes the longest time to fuzz a test case. The generation time accounts for a much larger part of this, as Classming needs to get the scope of the live code by executing test cases (both origin and mutate programs). As for ComFuzz, it generates test cases with large loops when using **INSL** mutator, leading to a long run time in the total test and a lower throughput than other baselines. Nonetheless, the fuzzing throughput of ComFuzz is comparable to other fuzzers.

6 DISCUSSIONS AND THREATS TO VALIDITY

Our work focuses on the context of compiler testing by combining historical test cases and bug-guided mutation rules. ComFuzz provides a focused and efficient compiler fuzzing framework, achieving a higher bug-exposing capability than SOTA solutions. We emphasize that ComFuzz is not designed to replace existing fuzzers. Instead, we aim to generate bug-exposing test cases for quickly discovering buggy compiler behaviors. Hence, we employ a DL-based model to learn a test program generator from historical test cases. Unlike JAttack, our program generator cannot guarantee the correct syntax of all synthesized test cases due to the usage of probabilistic prediction mechanisms during the sampling process. Still, the efficiency in generating syntactically valid programs would remain largely unchanged compared to JAttack. In the future, we will try to employ more powerful neural networks with a larger number of training samples. Unlike existing fuzzers, our work does not pursue full code coverage for the tested compiler. In contrast, we focus on covering the buggy compiler components via a set of carefully designed bug-guided mutators.

Threats. Our experiments may not generalize beyond the evaluated fuzzers and languages beyond Java and JS. We mitigate this by evaluating eight SOTAs. Porting our technique to a new program language would require the DL-based test program generator on new historical test cases collected for the targeting language, redesign some of the mutation rules and key information extractor for differential testing but the model training can be largely automated.

7 RELATED WORK

Generative fuzzers. Generation-based testing often utilizes stochastic grammar rules [14, 15, 33, 35, 49, 58, 59] or generation templates [20, 21, 45] to synthesize tests. The representative methods are EdSketch [45] and jsfunfuzz [72]. EdSketch is an open-source template-based JVM fuzzer that uses hand-written generation templates to synthesize Java programs. Similarly, jsfunfuzz employs pre-defined context-free grammars to generate JS test cases. Subsequent studies have proposed increasingly complex grammar and templates to improve the syntactic or semantic passing rates of generated test programs [62, 63, 76]. JAttack [89] provides customized generation templates where developers can encapsulate

Table 4: Fuzzing throughput (#test cases/minute).

Fuzzers (JS)	Throughput	Fuzzers (JVM)	Throughput
ComFuzz	15.58	**ComFuzz**	17.00
Comfort	9.58	Classming	2.19
Montage	24.39	JavaTailor	18.07
CodeAlchemist	29.85	JAttack	20.91
DIE	19.35		
POLYGLOT	10.45		

the expected code features for JVM testing. SPE [90] introduces a syntactic template that consists of a skeletal program and variables set. It generates random equivalent C programs by enumerating the combinations of the skeletal program and the variables. CL-Smith [53], the extended method of CSmith [85], added multiple generation options to generate OpenCL kernels for covering more compiler features. However, these approaches pursue full coverage of target compilers, leading to an inefficient bug-revealing ability. By contrast, ComFuzz is devoted to quickly exposing the buggy compiler component by generating bug-revealing test cases based on historical test programs. We are the first to do so.

Mutational fuzzers. Mutational testing aims to improve the code coverage for target compilers, which is achieved by reassembling or modifying a set of seed programs [22, 23, 44, 80, 84]. EMI [48] and its subsequent works [28, 54, 56] are among the representative mutation-based fuzzers. They generate semantic-equivalent test cases by performing equivalent mutations. LangFuzz [44] uses and recombines code fragments that previously exposed bugs to generate random JS programs. SYMFUZZ [22] mutates parent test programs based on the optimal mutation ratio that is determined by white-box symbolic analysis. IFuzzer [81] utilizes genetic programming techniques to generate unusual input code fragments for testing JS engines. CodeAlChemist [40] and its improvement work DIE [65] breaks the historical JS PoCs into code segments and reassembles these segments into new test programs. Classming [25] mutates the parent test cases by introducing a live bytecode mutation technique for JVM testing. Differ from the aforementioned fuzzers, ComFuzz utilizes the bug-guided mutators to generate new test cases by mutating the parent bug-revealing test programs, which can cover the deeper code branches and implement highly-intensive testing for the buggy compiler component. This mutation insight could be valuable for mutation-guided compiler testing.

Guided compiler testing. Since random test case generation methods for compiler testing are blind and time-consuming, recent studies have proposed a set of guided fuzzers. AFL [88] is the first coverage-guided testing framework, and it employs compile-time instrumentation and genetic algorithms to assist in generating random test cases for covering more code branches. The subsequent works [17, 38, 51, 60, 66, 69, 84] further improve code coverage for domain-specific testing by mutating the seed programs. Poloto et al. [67] proposed an interpreter-guided unit testing solution on the JIT compiler. It employs concolic testing to explore all possible execution paths and the corresponding values of an interpreter and uses these concrete values to implement differential unit testing on multiple JIT compilers. Classfuzz [26] is a coverage-guided method on JVM compilers, it employs MCMC sampling to guide mutator selection. Confuzzion [18] introduces a mutational feedback-guided fuzzer on JVM for exposing the type of confusion vulnerabilities. It uses historical execution information to randomly select mutation methods to generate new test cases. JavaTailor [91] is a closely related work that produces randomly generated tests by mutating historical test programs. The key difference is that ComFuzz-generated test cases are bug-directed that can perform focused and highly intensive testing for a buggy compiler component, leading to more effective testing than other guided fuzzers.

DL-based testing. To reduce human involvement, deep-learning models have been used to generate test cases. DeepSmith [29] and Learn&fuzz [34] start with the recurrent neural network (RNN) to generate test code, opening the DL-based testing trend. The subsequent work [50, 57, 86] explored different deep learning architectures, e.g., LSTM [43], Seq2Seq [79], and GPT-2 [68], to improve the syntactic passing rate of the generated test codes. Inspired by existing methods, ComFuzz also uses the deep learning model for test code generation, but it focuses on a directed generation by feeding the neural network with historical test cases.

8 CONCLUSIONS

We have presented ComFuzz, a fuzzing framework for detecting compiler bugs. ComFuzz leverages historical bug-exposing test programs to generate test cases. This strategy increases the test coverage of compiler components that are likely to contain bugs. Rather than solely depending on past test cases and applying random mutations across all compiler components, ComFuzz focuses in on modules previously known for bugs. Such modules have been historically prone to errors and can potentially contain bugs introduced by fixes. A unique feature of ComFuzz is its use of bug-driven mutators to uncover these residual bugs. To further enrich our testing diversity, we incorporated five multipurpose mutators designed to produce new test cases when existing ones fall short in revealing bugs or enhancing code coverage.

We evaluate ComFuzz on 11 distinct compilers, covering both JS and Java. In 260 testing hours, it unveiled 33 distinct bugs in nine of those compilers. Of these detected issues, 29 were verified, with 22 being rectified by the developers. Compared to eight prior fuzzers, ComFuzz uncovers at least 1.5× more bugs than its counterparts.

DATA AVAILABLE

The data and code associated with this paper are openly available at https://github.com/NWU-NISL-Fuzzing/COMFUZZ.

ACKNOWLEDGEMENTS

We thank the ESEC/FSE anonymous reviewers for their constructive feedback. We also thank all the JVM and JS developers for analyzing and replying to the bugs we reported. This work was supported in part by the National Natural Science Foundation of China (NSFC) under grant agreements 62102315, 61972314, the China Postdoctoral Science Foundation Fellowship (2022M712575), the Shaanxi International Science and Technology Cooperation Program (2023-GHZD-04), the Shaanxi Province "Engineers + Scientists" Team Building Program (2023KXJ-055), and the CCF-Tencent Open Fund.

REFERENCES

[1] [n. d.]. https://jshint.com/ Accessed on August 2023.

[2] [n. d.]. Esprima: ECMAScript parsing infrastructure for multipurpose analysis. https://esprima.org/ Accessed on August 2023.

[3] [n. d.]. Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html Accessed on August 2023.

[4] [n. d.]. JavaCompiler. https://docs.oracle.com/javase/8/docs/api/javax/tools/JavaCompiler.html Accessed on August 2023.

[5] [n. d.]. Lcov. https://github.com/linux-test-project/lcov Accessed on August 2023.

[6] [n. d.]. llvm-cov. https://www.llvm.org/docs/CommandGuide/llvm-cov.html Accessed on August 2023.

[7] [n. d.]. Mutators of COMFUZZ. https://github.com/NWU-NISL-Fuzzing/COMFUZZ/blob/main/docs/ComFuzz-mutators.md Accessed on August 2023.

[8] [n. d.]. OpenJDK Test7052494. https://github.com/openjdk/jdk7u/blob/master/hotspot/test/compiler/7052494/Test7052494.java#L56 Accessed on August 2023.

[9] [n. d.]. The sort function. https://github.com/oowuyue/notebook/blob/9efd633366c4d2721d76bb9ce7f95b89cc1260cb/js/js-gist/fortune500.js#L166 Accessed on August 2023.

[10] [n. d.]. Test262: ECMAScript Test Suite (ECMA TR/104). https://github.com/tc39/test262 Accessed on August 2023.

[11] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*.

[12] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1170–1188.

[13] Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: high-throughput fuzzing of network applications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 340–351. https://doi.org/10.1145/3533767.3534376

[14] 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

[15] 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.

[16] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Software* 38, 3 (2021), 79–86.

[17] 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

[18] William Bonnaventure, Ahmed Khanfir, Alexandre Bartel, Mike Papadakis, and Yves Le Traon. 2021. CONFUZZION: A Java Virtual Machine Fuzzer for Type Confusion Vulnerabilities. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 586–597.

[19] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*. 201–211.

[20] Andrea Calvagna, Andrea Fornaia, and Emiliano Tramontana. 2014. Assessing the correctness of JVM implementations. In *2014 IEEE 23rd International WETICE Conference*. IEEE, 390–395.

[21] Andrea Calvagna and Emiliano Tramontana. 2013. Automated conformance testing of Java virtual machines. In *2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems*. IEEE, 547–552.

[22] 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

[23] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding Typing Compiler Bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 183–198. https://doi.org/10.1145/3519939.3523427

[24] 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

[25] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1257–1268.

[26] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.

[27] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One engine to fuzz'em all: Generic language processor testing with semantic validation. In *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 642–658.

[28] Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T Johnson, and Christoph Csallner. 2020. SLEMI: Equivalence modulo input (EMI) based mutation of CPS models for finding compiler bugs in Simulink. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 335–346.

[29] 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

[30] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 219–232.

[31] Marco D'Ambros and Michele Lanza. 2006. Software bugs and evolution: A visual approach to uncover their relationship. In *Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE, 229–238.

[32] Walter R Gilks, Sylvia Richardson, and David Spiegelhalter. 1995. *Markov chain Monte Carlo in practice*. CRC press.

[33] 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

[34] 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

[35] 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

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

[37] Dick Grune, Kees Van Reeuwijk, Henri E Bal, Ceriel JH Jacobs, and Koen Langendoen. 2012. *Modern compiler design*. Springer Science & Business Media.

[38] 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

[39] Varuna Gupta, N Ganeshan, and Tarun Kumar Singhal. 2015. Determining the root causes of various software bugs through software metrics. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE, 1211–1215.

[40] 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

[41] 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

[42] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[43] 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

[44] 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

[45] Jinru Hua and Sarfraz Khurshid. 2017. EdSketch: Execution-driven sketching for Java. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. 162–171.

[46] 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

[47] Triet HM Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–38.

[48] 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

[49] 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

[50] 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.

[51] 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

[52] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *Network and Distributed System Security Symposium*.

[53] 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

[54] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. *ACM SIGPLAN Notices* 50, 6 (2015), 65–76.

[55] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1547–1559.

[56] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-guided tensor compiler fuzzing with joint IR-pass mutation. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–26.

[57] 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

[58] 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

[59] 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

[60] 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

[61] W. M. Mckeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[62] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2022. Fuzzing Class Specifications. https://doi.org/10.48550/ARXIV.2201.10874

[63] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 329–340. https://doi.org/10.1145/3293882.3330576

[64] Eunjung Park, John Cavazos, and Marco A Alvarez. 2012. Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 196–206.

[65] 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

[66] Chao Peng and Ajitha Rajan. 2020. Automated Test Generation for OpenCL Kernels Using Fuzzing and Constraint Solving. In *Proceedings of the 13th Annual Workshop on General Purpose Processing Using Graphics Processing Unit* (San Diego, California) *(GPGPU '20)*. Association for Computing Machinery, New York, NY, USA, 61–70. https://doi.org/10.1145/3366428.3380768

[67] Guillermo Polito, Stéphane Ducasse, and Pablo Tesone. 2022. Interpreter-guided differential JIT compiler unit testing. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 981–992.

[68] 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.

[69] 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

[70] Alan Romano, Xinyue Liu, Yonghwi Kwon, and Weihang Wang. 2021. An Empirical Study of Bugs in WebAssembly Compilers. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 42–54. https://doi.org/10.1109/ASE51524.2021.9678776

[71] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. In *NeurIPS EMC2 Workshop*.

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

[73] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*. 61–72.

[74] 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

[75] Nicholas Smith, Danny Van Bruggen, and Federico Tomassetti. 2017. Javaparser: visited. *Leanpub, oct. de* (2017).

[76] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective Grammar-Aware Fuzzing *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 244–256. https://doi.org/10.1145/3460319.3464814

[77] 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

[78] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 294–305.

[79] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

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

[81] 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

[82] 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* 16 (2020), 1943–1958. https://doi.org/10.1109/TIFS.2020.3044773

[83] 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

[84] 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

[85] 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

[86] 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*. 435–450. https://doi.org/10.1145/3453483.3454054

[87] 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

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

[89] Zhiqiang Zang, Nathaniel Wiatrek, Milos Gligoric, and August Shi. 2022. Compiler Testing using Template Java Programs. In *International Conference on Automated Software Engineering*. To appear. https://doi.org/10.1145/3551349.3556958

[90] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 347–361.

[91] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-driven test program synthesis for JVM testing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 1133–1144.

[92] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation.. In *USENIX Security Symposium*. 1099–1114.

[93] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).

Received 2023-02-02; accepted 2023-07-27