Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation

Jueon Eom
Yonsei University
Seoul, South Korea
jueoneom@yonsei.ac.kr

Seyeon Jeong
Suresofttech Inc.
Seongnam-si, South Korea
best6653@gmail.com

Taekyoung Kwon
Yonsei University
Seoul, South Korea
taekyoung@yonsei.ac.kr

Abstract

JavaScript interpreters, crucial for modern web browsers, require an effective fuzzing method to identify security-related bugs. However, the strict grammatical requirements for input present significant challenges. Recent efforts to integrate language models for context-aware mutation in fuzzing are promising but lack the necessary coverage guidance to be fully effective. This paper presents a novel technique called CovRL (Coverage-guided Reinforcement Learning) that combines Large Language Models (LLMs) with Reinforcement Learning (RL) from coverage feedback. Our fuzzer, CovRL-Fuzz, integrates coverage feedback directly into the LLM by leveraging the Term Frequency-Inverse Document Frequency (TF-IDF) method to construct a weighted coverage map. This map is key in calculating the fuzzing reward, which is then applied to the LLM-based mutator through reinforcement learning. CovRL-Fuzz, through this approach, enables the generation of test cases that are more likely to discover new coverage areas, thus improving bug detection while minimizing syntax and semantic errors, all without needing extra post-processing. Our evaluation results show that CovRL-Fuzz outperforms the state-of-the-art fuzzers in enhancing code coverage and identifying bugs in JavaScript interpreters: CovRL-Fuzz identified 58 real-world security-related bugs in the latest JavaScript interpreters, including 50 previously unknown bugs and 15 CVEs.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

fuzzing; coverage; reinforcement learning; large language model

ACM Reference Format:
Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680389

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680389

1 Introduction

JavaScript (JS) interpreters are essential for modern web browsers, enabling interactive web and embedded applications through the parsing, interpreting, compiling, and executing of JavaScript code. With JavaScript being employed as a client-side programming language by 98.9% of web browsers as of January 2024 [59], the security of JavaScript interpreters is crucial. Vulnerabilities can lead to severe security threats, including information disclosure and the bypassing of browser security measures [19, 38]. Given their critical role and complex nature, JavaScript interpreters require rigorous and continuous testing methods like fuzzing.

Previous research on fuzzing JavaScript interpreters primarily falls into two categories: *grammar-level* and *token-level*, each addressing the strict grammar requirements of JavaScript. Grammar-level fuzzing aims to generate grammatically correct inputs, ensuring syntactic accuracy [1, 20–22, 45, 46, 58, 60, 61]. Token-level fuzzing techniques adopt a more flexible approach by manipulating sequences of tokens without strict adherence to grammar rules [4, 50]. Both approaches have employed coverage-guided fuzzing, such as AFL [39], to enhance the fuzzing effectiveness by promoting an exhaustive examination of code paths [4, 22, 45, 50, 61]. However, the evolving nature of the JavaScript language, with its constantly updating grammar, poses significant challenges. Grammar-level fuzzing focuses intensely on generating mutations that precisely follow syntax rules, limiting mutation diversity. Given fuzzing's ability to produce vast amounts of input per second, it can manage some noise and inaccuracies, even when not strictly adhering to grammar rules. This focus on syntax can paradoxically reduce mutation variety and constrain program path exploration. Token-level fuzzing, although more flexible, struggles with maintaining syntactical correctness over successive mutations, often leading to syntax errors and hindering the discovery of deeper bugs.

To address these challenges, recent advancements have led to research into fuzzing techniques that employ LLMs, which are adept at producing syntactically informed, well-formed inputs for compilers and JavaScript interpreters [65, 67]. A standout example is Fuzz4All [65], which utilizes pretrained Code-LLMs for compiler fuzzing. These models, trained on extensive datasets across various programming languages, offer application for LLM-based mutation without further finetuning. They inherently grasp the language's context, enabling the generation of inputs that are grammatically accurate and contextually relevant, thereby enhancing fuzzing effectiveness. However, current LLM-based fuzzing methods are generally considered black-box fuzzing and lack integration with internal program information like code coverage. In contrast to black-box fuzzing, coverage-guided fuzzing utilizes internal program data to enhance fuzzing effectiveness. The technique employs

Table 1: Average coverage achieved by four LLM-based mutations compared to random mutation was measured using AFL for 5 hours on the JavaScript interpreter V8 with a single core. Among these variants, the baseline is Token-Level AFL [50] setting in Table 2, and prompt setting used was "Please mutate the following program". The experiment for GPT-4 [42] was conducted by requesting and receiving mutations through the OpenAI API [43].

Variants		Error (%)	Coverage		Improv
Strategy	LLM		Valid	Total	Ratio (%)
Random	✗ (Baseline [50])	88.79%	44,705	53,936	-
Prompt	GPT-4 [42]	27.50%	46,454	46,738	-13.35%
	StarCoder (1B) [31]	82.72%	41,331	45,034	-16.50%
Mask	Incoder (1B) [17]	49.08%	46,427	47,385	-12.15%
	CodeT5+ (220M) [63]	62.68%	55,459	56,576	4.89%

an evolutionary strategy for generating "interesting" seeds aimed at expanding the program's coverage, thus potentially increasing the likelihood of bug discovery. By effectively using code coverage feedback to guide the mutation of inputs, it can uncover bugs more efficiently than traditional black-box methods [7]. However, as we describe below, this is quite challenging.

Problem. LLMs typically generate sentences at the word level, which leads to the assumption that LLM-based mutations operate at the token level. Replacing traditional random mutators with pre-trained LLM-based mutators in coverage-guided fuzzing generally reduces error rates but does not enhance coverage. Our experimental results, detailed in Table 1, show that using the AFL fuzzing tool, LLM-based mutations on V8 for five hours resulted in 12-16% lower coverage compared to the baseline in most cases. Even in cases where coverage increased, the rise was only slight. This suggests that while LLM-based mutations reduce errors, their constrained predictions may limit diversity and effectiveness.

We hypothesize that LLM-based mutations, focusing on context, often predict common tokens and unintentionally reduce diversity. This is similar to grammar-level mutations, which target grammatical accuracy but also limit variability. Consequently, in coverage-guided fuzzing, while LLM-based mutators decrease errors, their context-aware approach diminishes diversity, making them less effective than random fuzzing.

Our Approach. To address the limitation of LLM-based mutations, we propose a novel technique that integrates coverage-guided feedback directly into the mutation process. Our approach leverages internal program information to enhance fuzzing effectiveness, aiming to generate diverse mutations that go beyond mere grammatical correctness. We employ Term Frequency-Inverse Document Frequency (TF-IDF) [56] to weight coverage data, establishing a feedback-driven reward system. This method not only increases coverage but also improves bug detection capabilities of LLM-based fuzzing, eliminating the need for additional post-processing. We term our approach *CovRL-Fuzz*. Unlike other LLM-based fuzzing techniques, *CovRL-Fuzz* is the first to effectively integrate LLM-based mutation with coverage-guided fuzzing, thereby distinguishing it from existing methods [65, 67].

To sum up, this paper makes the following contributions:

- We introduce CovRL, a novel method integrating LLMs with coverage feedback by reinforcement learning, using TF-IDF. We directly feed code coverage to LLMs via a new reward scheme.
- We implement CovRL-Fuzz, a new JavaScript interpreter fuzzer that outperforms existing methods in code coverage and bug detection by employing the CovRL technique.
- CovRL-Fuzz identified 58 real-world security-related bugs, including 50 unknown bugs (15 CVEs) in the latest JavaScript interpreters.
- To foster future research, we release our implementation of CovRL-Fuzz at https://github.com/seclab-yonsei/CovRL-Fuzz.

2 Background

2.1 JavaScript Interpreter Fuzzing

Fuzzing is a powerful automated method for finding software bugs [41] and is highly regarded in both academia and industry. However, it faces challenges with JavaScript interpreters due to their need for strictly grammatical input. When inputs are not syntactically correct, the JavaScript interpreter returns a syntax error, while semantic inconsistencies (e.g., errors with reference, type, range, or URI) can lead to semantic errors [21]. In both scenarios, the interpreter's internal logic, which may contain hidden bugs, is not executed. To tackle these challenges, researchers have developed grammar-level and token-level fuzzing methods. The grammar-level approach converts seeds into an Intermediate Representation (IR) to ensure grammatical accuracy of inputs [1, 20–22, 45, 46, 58, 60, 61]. While effective in maintaining syntax, adhering strictly to grammar rules limits mutation diversity, making it difficult to detect bugs caused by grammar violations or unexpected input patterns. On the other hand, token-level fuzzing offers greater flexibility by breaking inputs into tokens and replacing them selectively, enhancing bug detection capabilities [4, 50]. However, it often fails to maintain syntactical correctness due to its random substitution method, which does not consider relationships between tokens. Recent studies also explore bugs in Just-In-Time (JIT) compilers of JavaScript engines through differential testing in behavior with JIT enabled and disabled [3, 62].

LLM-based Fuzzing. Recently, there has been a shift toward using deep learning-based Language Models (LMs) in fuzzing to address the limitations of traditional methods. Initially, RNN-based LMs were used to mutate seeds [11, 18, 29, 35], and more recently, Large Language Models (LLMs) like GPTs [42, 47] and StarCoder [31] have been employed for seed generation and mutation [65, 67], benefiting from extensive training on diverse programming languages datasets.

Coverage-guided Fuzzing. Coverage-guided fuzzing, which leverages coverage feedback to explore diverse code paths, has proven more effective than traditional black-box methods in detecting software bugs [7]. This approach, exemplified by tools like American Fuzzy Lop (AFL) [39], emphasizes maximizing code coverage through mutations [4, 22, 45, 50, 61], and has been particularly effective in finding security vulnerabilities. Despite the success of coverage-guided fuzzing, existing LLM-based fuzzing techniques, including COMFORT and Fuzz4All, primarily use black-box methods [65, 67] and have not achieved coverage-guided fuzzing for

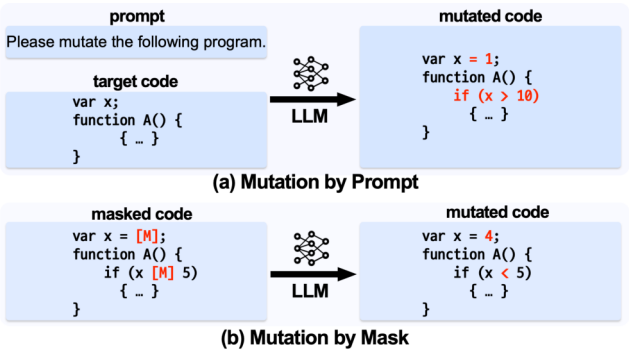

Figure 1: Types of LLM-based Mutation.

JavaScript interpreters, as these approaches typically do not incorporate coverage feedback into their mutation processes.

2.2 Large Language Models for Code

Following the success of LLMs in Natural Language Processing (NLP) tasks [6, 9, 10], the field of programming languages is advancing with significant contributions from Large Language Models for code (Code-LLMs) such as CodeT5+ [63], Codex [8], InCoder [17] and StarCoder [31]. These advancements are facilitating various downstream tasks, including code completion [68], program synthesis [2, 27, 34, 55], program repair [16, 66], and many others.

LLM-based Mutation. In fuzzing, Code-LLMs have recently shown their effectiveness not only in seed generation but also in mutation [12, 13, 65]. The use of LLMs in mutation can be broadly categorized into two types: mutation by prompt [13, 65] and mutation by mask [12]. Figure 1 illustrates the types of LLM-based mutation. Mutation by prompt involves inputting a code to be mutated along with a mutation request prompt (e.g., "Please mutate the following program.") into a pretrained Code-LLM to generate a mutated seed ((a) in Figure 1). The Code-LLM fully relies on its understanding of the structure and function of the code to perform mutations such as changing variable names or adding loops and conditional statements as requested. Mutation by mask is a methodology where masks are added or replaced in the middle of code, allowing the model to fill in only parts of it, thereby mutating the code ((b) in Figure 1). The mutation technique allows for direct selection of the parts to be mutated, enabling more precise adjustments of the mutation.

We used mutation by mask in our coverage-guided fuzzing as it preserves the overall structure while allowing specific tokens to be mutated. This targeted mutation strategy helps incrementally increase coverage, efficiently explore new code paths, and conserve resources. The advantages of mutation by mask over mutation by prompt in the context are also evidenced in Table 1.

Finetuning LLMs. Methods for finetuning LLMs, including Code-LLMs are categorized into supervised finetuning (SFT), instruction finetuning [64], and RL-based finetuning [28, 44, 49]. While prompt engineering controls the output of LLMs at inference time through input manipulation alone, SFT, instruction finetuning, and RL-based finetuning aim to steer the model during training time by learning from specific datasets tailored to particular tasks. Particularly, RL-based finetuning has been proven effective in guiding LLMs using feedback to optimize factual consistency and reduce the toxic generation [28, 44, 49]. Recently, there have also been proposed for applying RL-based finetuning to Code-LLMs aimed at generating unit tests that are not only grammatically correct but also capable of solving complex coding tasks [27, 34, 55].

RL-based finetuning consists of the following phases: reward modeling and reinforcement learning. In reward modeling, an LLM-based rewarder is trained to evaluate the suitability of output results when input is provided to the LLM created in the previous phase. There are various approaches to feedback depending on how the rewarder is trained: utilizing an oracle [27, 34, 55], using deep learning models [28], and using human feedback [44]. We also adopt the strategy of employing the JavaScript interpreter as a feedback oracle. In reinforcement learning, training commonly employs Kullback-Leibler (KL) divergence-based optimization. The method is designed to optimize the balance between maximizing rewards and minimizing deviation from the initial training distribution.

3 Design

In this section, we describe the design of CovRL-Fuzz. The key accomplishment of CovRL-Fuzz is integrating coverage-guided feedback directly into the LLM-based mutation process. This ensures effective fuzzing by guiding mutations to be not only grammatically correct but also diverse.

Figure 2 illustrates the workflow of CovRL-Fuzz, which operates in three phases based on a coverage-guided fuzzer. Initially, CovRL-Fuzz selects a seed from the queue. The seed then undergoes an LLM-based mutation where specific tokens are masked and predicted using a masked language model approach resulting in a new test case (❶). The technique of mutation by mask helps to maintain structural integrity while exploring new code paths [14, 48]. After mutation, the test case is then executed by the target JavaScript interpreter. If the test case discovers new coverage previously unseen, it is considered an interesting seed and is added to the seed queue for further mutation. At the same time, CovRL-Fuzz stores the coverage map measured by the test case and validity information, whether the test case led to syntax errors, semantic errors, or passed successfully. Our rewarding approach uses validity information to impose penalties on inputs that result in syntax or semantic errors. Following this, it generates rewarding signals by multiplying the current coverage map with a coverage-based weight map (❷). After completing a mutation cycle, we proceed to finetune the LLM-based mutator using CovRL by utilizing the gathered interesting seeds and rewarding signals. We define the notion of one cycle as a predetermined number of mutations. The CovRL employs the PPO [51] algorithm, a method that seeks to improve the current model while adhering closely to the previous model's framework. The signal during training prevents the LLM from making syntax or semantic errors and induces prediction to find new coverage (❸).

Note that we do not perform any heuristic post-processing on the LLM-based mutation, except for CovRL-based finetuning. We demonstrated a minimal error rate in using solely CovRL that is

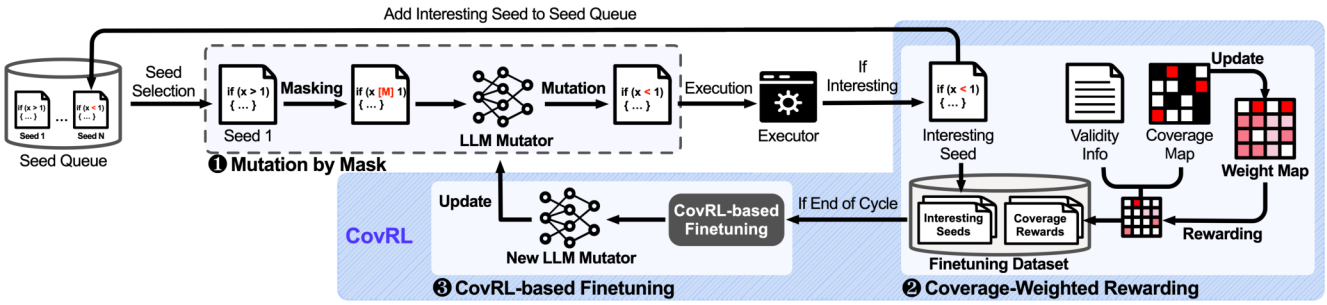

Figure 2: Workflow of CovRL-Fuzz: The blue-shaded area illustrates the operation of CovRL.

comparable to other latest JavaScript interpreter fuzzing techniques in Section 5.2.

3.1 Phase 1. Mutation by Mask

To mutate the selected seed, CovRL-Fuzz performs a simple masking strategy for mutation by mask (❶ in Figure 2). Given the input sequence $W = \{w_1, w_2, .., w_n\}$, we use three masking techniques: insert, overwrite, and splice. The strategy results in the mask sequence $W^{MASK} = \{[MASK], w_3, .., w_k\}$ and the masked sequence $W^{\setminus MASK} = \{w_1, w_2, [MASK], .., w_n\}$. The detailed operations are described as follows:

Insert Randomly select positions and insert [MASK] tokens into the inputs.

Overwrite Randomly select positions and replace existing tokens with the [MASK] token.

Splice Statements within a seed are randomly divided into segments. A portion of these segments is replaced with a segment from another seed with [MASK], formatted as [MASK] statement [MASK].

After generating a masked sequence $W^{\setminus MASK}$ via masking, the seed is mutated by inferring in the masked positions via LLM-based mutator. The mutation design of CovRL-Fuzz is based on a span-based masked language model (MLM) that can predict variable-length masks [17, 48]. Therefore, the MLM loss function we utilize for mutation can be represented as follows:

$$L_{MLM}(\theta) = \sum_{i=1}^{k} -logP_\theta(w_i^{MASK} | w^{\setminus MASK}, w_{<i}^{MASK}) \quad (1)$$

θ represents the trainable parameters of the model that are optimized during the training process, and k is the number of tokens in W^{MASK}. $w^{\setminus MASK}$ denotes the masked input tokens where specific tokens are replaced by mask tokens. w^{MASK} refers to the original tokens that have been substituted with the mask tokens in the input sequence.

3.2 Phase 2. Coverage-Weighted Rewarding

We developed a method called Coverage-Weighted Rewarding (CWR) to guide the LLM-based mutator. The approach employs TF-IDF to emphasize less common coverage points, effectively focusing on discovering new areas of code coverage (❷ in Figure 2). TF-IDF prioritizes less frequent tokens by assigning them higher weights, and more common tokens receive lower weights. We apply this method to create a weighted coverage map, focusing on underexplored areas. Figure 3 illustrates the Coverage-Weighted Rewarding (CWR) process in action.

Example 1. *Consider the scenario of executing a program depicted by Control Flow Graph (CFG) in Figure 3 (a). There is a loop C, and branches D and E in the CFG. We executed the program with a test case generated by a fuzzer and obtained the coverage results.*

Building on previous RL-based finetuning methods using Code-LLM [27, 34, 55], we further extend the idea by applying a rewarding signal based on software output. Notably, errors in the JavaScript interpreter can be broadly grouped into syntax errors and semantic errors, which include reference, type, range, and URI errors. Given that W^* is the concatenation of the masked sequence $W^{\setminus MASK}$ and the mask sequence W^{MASK}, the following returns can be deduced based on input to the target:

$$r(W^*) = \begin{cases} -1.0 & \text{if } W^* \text{ is syntax error} \\ -0.5 & \text{if } W^* \text{ is semantic error} \\ +R_{cov} & \text{if } W^* \text{ is passed} \end{cases} \quad (2)$$

To enhance the LLM-based mutator's ability to discover diverse coverage, we introduce an additional rewarding signal as outlined in Eq. 2. Unlike traditional RL-based fuzzing techniques that use the ratio of current to total coverage, our approach assigns weights based on the frequency of reaching specific coverage points. This method involves adjusting the coverage map using the IDF weight map, calculating the weighted sum for each coverage data point, and normalizing these sums to derive scores.

Initially, we observed that the coverage map functions like Term Frequency (TF) by tracking how often certain coverage points are hit. However, in JavaScript interpreters, identical sections of code in a test case, such as repeated lines like 'a=1; a=1;', can trigger the same coverage area multiple times, leading to redundant counts. To address this, we underscore the importance of recognizing unique instances of coverage. Hence, we introduce TF^{cov}, defining it as a map that records each coverage point uniquely, reducing redundancy and emphasizing the significance of varied coverage.

$$TF^{cov} = \text{unique coverage map} \quad (3)$$

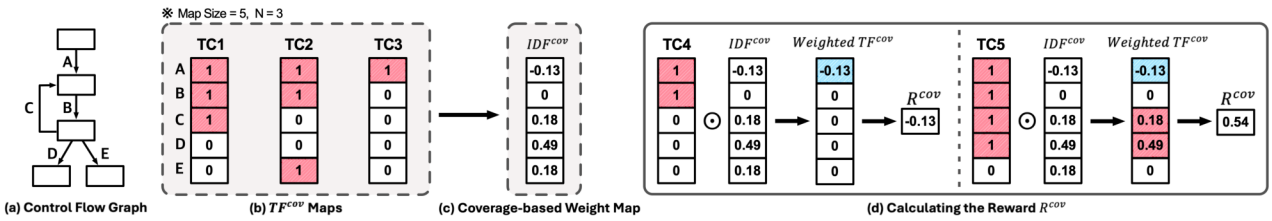

Figure 3: Example of Coverage-Weighted Rewarding process. (a) represents the control flow graph of an example program. (b) shows the TF^{cov} Maps calculated based on the coverage areas traversed when each Test Case (TC) is executed in the program represented by (a). (c) is the IDF^{cov} calculated based on the TF^{cov} Maps, which we refer to as the Coverage-based Weight Map. (d) describes the process of calculating the R_{cov}. Additional TCs are element-wise multiplied by IDF^{cov} to become the weighted coverage map, called the Weighted TF^{cov} map, and the sum of this map is rewarded to the model as the reward R^{cov}. Note that the calculation of R^{cov} in this figure is a simplified version of Eq.5.

Example 2. *Assume we obtained coverage from test case 1 (TC1) as follows: [1, 3, 4, 0, 0]. Applying Eq. 3, the coverage transforms into a binary map to indicate whether a path was executed (1) or not (0), regardless of the number of times it was executed. Thus, the TF^{cov} map for TC1 is updated to: [1, 1, 1, 0, 0]. And this process is applied identically to other test cases as well ((b) in Figure 3).*

We also define the coverage-based weight map IDF^{cov} using the TF^{cov} of each seed to weight which code paths are accessed frequently and which are not, as follows:

$$IDF^{cov} = \frac{1}{\sqrt{M}} log(\frac{N}{1 + DF^{cov}}) \tag{4}$$

where N denotes the total number of unique coverage obtained. DF^{cov} denotes the number of seeds that have achieved the specific coverage point. The weight map IDF^{cov} is obtained by taking the inverse of DF^{cov}, resulting in higher weights for less common coverage. The variable M denotes the overall size of the coverage map, which we utilized as a scale factor to adjust the weight value. The scaling ensures that the weights remain consistent regardless of the size M of the coverage map, thus preserving the stability of the training process.

Example 3. *Consider in Figure 3 (b) that three TF^{cov} maps, [1, 1, 1, 0, 0], [1, 1, 0, 0, 1], and [1, 0, 0, 0, 0], were generated. Based on these TF^{cov} maps, we calculate the coverage-based weight map, IDF^{cov}. By applying Eq. 4, IDF^{cov} is computed as a weight map with values [-0.13, 0.0, 0.18, 0.49, 0.18] ((c) in Figure 3).*

The reward is acquired by taking the weighted sum of TF^{cov} and IDF^{cov} to create the weighted coverage map, which is then weighted to obtain as

$$R_{cov} = \sigma(log(\sum_{i=1}^{M} tf_{i,t} \cdot idf_{i,t-1})) \tag{5}$$

where t represents the current cycle. $tf_{i,t}$ refers to an element in TF_t^{cov}, and $idf_{i,t-1}$ refers to an element in IDF_{t-1}^{cov} at the previous time step before updating the weights. σ is a sigmoid function used to map R_{cov} to a value between 0 and 1.

The R_{cov} is calculated only if the test case is free from any syntax or semantic problems. Our rewarding scheme incentivizes the LLM-based mutator to explore a wider range of coverage by providing higher payouts for test cases that achieve uncommon levels of coverage.

Example 4. *Assume we generated two new test cases, TC4 and TC5, through fuzzing and executed the program to obtain TF^{cov} map. We create a weighted TF^{cov} by element-wise multiplying each TC with IDF^{cov}, and then calculate R^{cov} as the sum of the elements according to Eq. 5. In the case of TC4, because it executed mostly the code paths that TC1, TC2, and TC3 had executed, it received a penalty of -0.13, whereas TC5, having executed rare or previously unexecuted code paths, obtained a reward of 0.54 ((d) in Figure 3).*

Update Weight Map with Momentum. Although we used the logarithmic function in Eq. 4 to alleviate dramatic changes in IDF^{cov}, instability can still occur due to abrupt shifts in the reward distribution. To address this issue, we employed momentum. Following each cycle, CovRL updates the IDF weight map. To mitigate dramatic changes in the reward distribution, we use momentum at a rate of α to incorporate the prior weight when recalculating the map. The updated weight map is as follows:

$$IDF_{t-1}^{cov} = \alpha IDF_{t-1}^{cov} + (1 - \alpha) IDF_t^{cov} \tag{6}$$

where IDF_t^{cov} means new weight map and IDF_{t-1}^{cov} means previous weight map.

3.3 Phase 3. CovRL-Based Finetuning

The fuzzing environment with mutation by mask can be conceptualized as a bandit environment for RL. In this environment, a masked sequence $W^{\setminus MASK}$ is provided as input (x), and the expected output is a mask sequence W^{MASK} (y). Inspired by previous studies [27, 34, 55], we finetune our model using the PPO algorithm [51], an actor-critic reinforcement learning (❸ in Figure 2). In our situation, it can be implemented by finetuning two LLMs in tandem: one LLM acts as a mutator (actor), while the other LLM serves as a rewarder (critic). We utilize a pretrained LLM to initialize the parameters both of mutator and rewarder. The rewarder is trained using the Eq. 2. It plays a crucial role in training the mutator.

For CovRL-based finetuning with PPO, we define the CovRL loss as follows:

$$L_{CovRL}(\theta) = -\mathbb{E}_{(x,y)\sim D_t}\left[R(x,y)_t \cdot \log\left(\frac{\pi_\theta^t(y|x)}{\pi^{t-1}(y|x)}\right)\right] \tag{7}$$

Algorithm 1: Fuzzing with CovRL

```
   Input: finetuning dataset D_T
1  R_prev : Previous LLM-based rewarder
2  R_cur : Current LLM-based rewarder
3  M_prev : Previous LLM-based mutator
4  M_cur : Current LLM-based mutator

5  Function FuzzOne(seed_queue):
6      for i = 1 to iter_cycle do
7          seed ← SelectSeed(seed_queue)
8          T ← Mutate(M_cur, seed)
9          I_val, cov ← Execute(T)
10         if IsInteresting(T) then
11             T_interest ← T
12             seed_queue.append(T_interest)
13             R_cov = calcReward(I_val, cov)
14             data ← T_interest, R_cov
15             D_T.append(data)
16     FinetuneCovRL(D_T)
17 Function FinetuneCovRL(D_T):
18     R_prev, M_prev ← R_cur, M_cur
19     R_cur ← FinetuneRewarder(R_prev, D_T)
20     M_cur ← FinetuneMutator(M_prev, R_cur, D_T)
```

where $R(x, y)$ represents the reward of CovRL, and D_t refers to the finetuning dataset that has been collected up to time step t. $\pi_\theta^t(y|x)$ with parameters θ is the trainable RL policy for the current mutator, and $\pi^{t-1}(y|x)$ represents the policy from the previous mutator.

To mitigate the overoptimization and maintain the LLM-based mutator's mask prediction ability, we also use KL regularization. The reward after adding the KL regularization is

$$R(x, y)_t = r(W^*) + \log\left(\frac{\pi_\theta^t(y|x)}{\pi^{t-1}(y|x)}\right) \tag{8}$$

Fuzzing with CovRL Algorithm. Algorithm 1 details one cycle of the fuzzing loop with CovRL. The cycle iterates for a predetermined number of *iter_cycle* (Lines 6-15). The LLM-based mutator uses a seed chosen from the *seed_queue* to generate the test case T (Lines 7-8). By executing T on the target interpreter, we obtain validity information I_{val} and coverage map *cov* (Line 9). If T is deemed a noteworthy seed, it is added to the seed queue, and the reward for the particular $T_{interest}$ is calculated and added to the finetuning dataset D_T (Lines 10-15). After completing these *iter_cycle* iterations, the gathered D_T is utilized as training data to call the **FinetuneCovRL** function, which carries out CovRL-based finetuning (Line 16). The procedure of **FinetuneCovRL** involves the finetuning of the LLM-based Rewarder $\mathcal{R}$ and the LLM-based Mutator $\mathcal{M}$ (Lines 18-20). Initially, we designate the existing model as $\mathcal{R}_{prev}$ and $\mathcal{M}_{prev}$ (Line 18). Following that, $\mathcal{R}_{prev}$ is finetuned using the D_T to generate a new rewarder $\mathcal{R}_{cur}$ (Line 19). At this point, the rewarder has been trained to predict the rewarding signal as described in Eq. 2. By utilizing the finetuned $\mathcal{R}_{prev}$ and D_T, we finetune the mutator $\mathcal{M}_{prev}$ to generate $\mathcal{M}_{cur}$ (Line 20). For finetuning the mutator, we apply reward or penalty to the model using the CovRL loss from Eq. 7.

4 Implementation

We implemented a prototype of CovRL-Fuzz using pytorch v2.0.0, transformers v4.38.2 and afl 2.52b [39].

Dataset. We collected data from regression test suites in several repositories including V8, JavaScriptCore, ChakraCore, JerryScript, Test262 [57], and js-vuln-db [23] as of December 2022. We then pre-processed the data for training data and seeds, resulting in a collection of 52K unique JavaScript files for our experiments.

Pre-Processing. We performed a simple pre-processing on the regression test suites of the JavaScript interpreters mentioned above to remove comments, filter out grammatical errors, and simplify identifiers. We then used the processed data directly for training. The pre-processing was conducted utilizing the `-m` and `-b` options of the UglifyJS tool [40].

Training. We utilized the pretrained Code-LLM, CodeT5+ (220m) [63], as both the rewarder and the mutator. For the process of CovRL-based finetuning, we trained the rewarder and mutator for 1 epoch per mutation cycle. We used a batch size of 256 and learning rate of 1e-4. The optimization utilized the AdamW optimizer [36] together with a learning rate linear warmup technique. The LLM-based rewarder uses the encoder from CodeT5+ to predict the rewarding signal through a classification approach. We utilized the contrastive search method, incorporating a momentum factor α of 0.6 and a top-k setting of 32, to enhance the effectiveness of CovRL. In addition, we aligned the coverage map size with AFL's recommendations by setting the scaling factor M for the map size. This ensures that the instrumentational capacity is optimized. For moderate-sized software (approx. 10K lines), we employed a map size of 2^{16}. For larger software exceeding 50K lines, we used a map size of 2^{17}, striking a balance between granularity and performance. For detailed analysis related to the hyperparameters that we have set, such as finetuning epochs and α, please refer to the Appendix.

5 Evaluation

To evaluate CovRL-Fuzz, we set four research questions.

- **RQ1:** Is CovRL-Fuzz more effective than other JavaScript interpreter fuzzers?
- **RQ2:** Is CovRL-Fuzz more effective compared to the other fuzzer using LLM-based mutation?
- **RQ3:** How does each component contribute to the effectiveness of CovRL-Fuzz?
- **RQ4:** Can CovRL-Fuzz find real-world bugs in JavaScript interpreters?

5.1 Experimental Design

Experimental setup. Our setup included a 64-bit Ubuntu 20.04 LTS OS on an Intel(R) Xeon(R) Gold 6134 CPU @ 3.20GHz (64-core). Additionally, we harnessed three NVIDIA GeForce RTX 3090 GPUs for both training and mutation.

Benchmarks. We tested it on four JavaScript interpreters, using the latest versions as of January 2023: JavaScriptCore (2.38.1), ChakraCore (1.13.0.0-beta), V8 (11.4.73), JerryScript (3.0.0). We also conducted additional experiments on QuickJS (2021-03-27), Jsish (3.5.0), escargot (bd95de3c), Espruino (2v20) and Hermes (0.12.0) for real-world bug detection experiments.
We built each target JavaScript interpreter with Address Sanitizer (ASAN) [52] to detect bugs related to abnormal memory access and in debug mode to find bugs related to undefined behavior.

Fuzzing Campaign. For a fair evaluation, we used the same set of 100 valid seeds. For **RQ1** and **RQ2**, we operated on 3 CPU cores, considering other fuzzing approaches. For **RQ3**, we used a single CPU core. For **RQ4**, we also used 3 CPU cores and conducted experiments with the four major JavaScript interpreters and five additional ones. To consider the randomness of fuzzing, we executed each fuzzer five times and then averaged the coverage results. Additionally, to ensure fairness in fuzzing, we measured the results of each experiment, including the finetuning time through CovRL. The average finetuning time is 10 minutes, occurring every 2.5 hours.

Each tool was run on four JavaScript interpreters with default configurations which details can be found in Table 2.

Metrics. We use three metrics for evaluation.

- **Code Coverage** represents the range of the software's code that has been executed. We adopt edge coverage from the AFL's coverage bitmap, following FairFuzz [30] and Evaluate-Fuzz-Testing [25] settings. We conducted a comparison of coverage in two categories: total and valid. Total refers to the coverage across all test cases, while valid refers to the coverage for valid test cases. We also employed the Mann-Whitney U-test [37] to assess the statistical significance and verified that all p-values were less than 0.05.
- **Error Rate** measures the rate of syntax errors and semantic errors in the generated test cases. The metric provides insight into how effectively each method explores the core logic in the target software. For detailed analysis, semantic errors are categorized into type errors, reference errors, URI errors, and internal errors based on the ECMA standard [15]. It should be noted that while COMFORT [67] utilized jshint [24] for measurement, focusing their error rate on syntax errors, we used JavaScript interpreters, allowing us to measure the error rate including both syntax and semantic errors.
- **Bug Detection** is what the fuzzer is trying to find.

Table 2: Baseline fuzzers targeting JavaScript interpreters. CGF indicates the use of coverage-guided fuzzing, LLM denotes the usage of LLMs, and Mutation Level refers to the unit of mutation.

Fuzzer	CGF	LLM	Mutation Level	Post Processing
Coverage-guided Baselines				
AFL(w/Dict) [39]	✓		Bit/Byte	
Superion [61]	✓		Grammar	
Token-Level AFL [50]	✓		Token	
LM Baselines				
Montage [29]			Grammar	✓
COMFORT [67]		✓	Grammar	✓
CovRL-Fuzz	✓	✓	Token	

Table 3: Comparison with other JavaScript interpreter fuzzers listed in Table 2.

Target	Fuzzer	Error (%)	Coverage Valid	Coverage Total	Improv Ratio (%)
V8	AFL (w/Dict)	96.90%	29,929	33,531	134.79%
	Superion	77.35%	33,812	36,985	112.87%
	Token-Level AFL	84.10%	39,582	42,303	86.11%
	Montage	56.24%	38,856	40,155	96.06%
	Montage (w/o Import)	94.08%	33,487	36,338	116.66%
	COMFORT	79.66%	44,324	46,522	69.23%
	CovRL-Fuzz	**48.68%**	**75,240**	**78,729**	-
JSC	AFL (w/Dict)	74.42%	18,343	20,496	215.86%
	Superion	72.02%	17,619	19,772	227.42%
	Token-Level AFL	69.70%	52,385	53,719	20.51%
	Montage	**42.34%**	55,511	56,861	13.85%
	Montage (w/o Import)	93.72%	43,861	47,754	35.57%
	COMFORT	79.64%	36,074	36,542	77.16%
	CovRL-Fuzz	48.59%	**61,137**	**64,738**	-
Chakra	AFL (w/Dict)	81.32%	83,038	87,587	27.30%
	Superion	**42.63%**	92,314	94,237	18.32%
	Token-Level AFL	90.64%	92,621	95,677	16.54%
	Montage	82.21%	101,470	103,589	7.63%
	Montage (w/o Import)	94.72%	90,940	98,643	13.03%
	COMFORT	79.47%	81,171	83,142	34.11%
	CovRL-Fuzz	54.87%	**105,121**	**111,498**	-
Jerry	AFL (w/Dict)	77.32%	9,307	14,259	63.03%
	Superion	86.23%	8,944	15,061	54.35%
	Token-Level AFL	80.52%	14,361	17,152	35.53%
	Montage	95.55%	13,114	13,285	74.98%
	Montage (w/o Import)	95.34%	12,662	15,598	49.03%
	COMFORT	79.83%	12,268	14,026	65.74%
	CovRL-Fuzz	**58.84%**	**20,844**	**23,246**	-

5.2 RQ1. Comparison with Existing Fuzzers

To answer **RQ1**, we compare CovRL-Fuzz with fuzzers targeting JavaScript interpreters from open-source projects, focusing on those that are either coverage-guided [39, 50, 61] or LM-based[29, 65, 67] listed in Table 2 with the 24-hour timeout. In the case of Montage, it imports code from its test suite corpus, which might affect coverage by increasing the amount of executed code. As a result, we included a version of Montage (w/o Import) in our experimental study, which does not import the other test suites. In the case of COMFORT, we evaluated it solely with the black-box fuzzer, excluding the differential testing component. We focused on finding bugs in general JavaScript interpreters, so we did not include JIT fuzzers [3, 62] targeting JIT compilers embedded in some JavaScript engines.

Code Coverage. Table 3 depicts the valid and total coverage for each fuzzing technique. The results of our evaluation demonstrate that CovRL-Fuzz outperforms state-of-the-art JavaScript interpreter fuzzers. Our observation revealed that CovRL-Fuzz attained the highest coverage across all target interpreters, resulting in an average increase of 102.62%/98.40%/19.49%/57.11% in edge coverage. To emphasize the effectiveness of CovRL-Fuzz, we monitored a growth trend of edge coverage, depicted in Figure 4. In every experiment, CovRL-Fuzz consistently achieved the highest edge coverage more

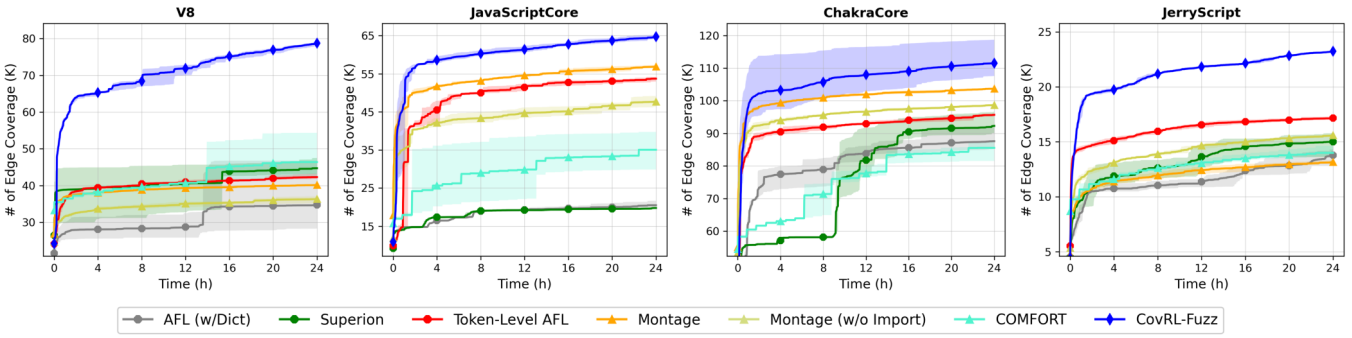

Figure 4: Number of edge coverage between CovRL-Fuzz and other JavaScript interpreter fuzzers. The solid line represents the average coverage, while the shaded region depicts the range between the lowest and highest values five times.

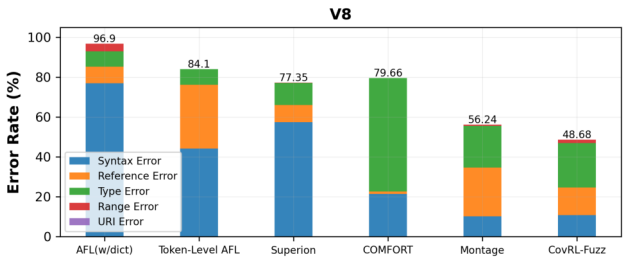

Figure 5: The error rate of generated test cases on V8. The four error types, excluding syntax error, are classified as semantic errors.

Table 4: Unique bugs discovered by CovRL-Fuzz and other JavaScript interpreter fuzzers.

Target	Bug Type	AFL	Superion	TokenAFL	Montage	COMFORT	CovRL-Fuzz
JSC	Undefined Behavior		✓				✓
JSC	Out-of-bounds Read						✓
Chakra	Undefined Behavior			✓		✓	✓
Chakra	Out of Memory						✓
Chakra	Out of Memory						✓
Jerry	Undefined Behavior	✓	✓	✓			✓
Jerry	Memory Leak		✓	✓			✓
Jerry	Undefined Behavior	✓		✓			✓
Jerry	Undefined Behavior						✓
Jerry	Heap Buffer Overflow						✓
Jerry	Out of Memory						✓
Jerry	Stack Overflow						✓
Jerry	Undefined Behavior						✓
Jerry	Heap Buffer Overflow						✓
	Total	2	3	4	0	1	14

rapidly than any other fuzzer. In contrast to coverage-guided baselines, CovRL-Fuzz immediately and significantly achieved higher coverage. This suggests that the LLM-based mutation of CovRL-Fuzz are more effective than the heuristic mutations in coverage-guided fuzzing. CovRL-Fuzz achieved high coverage compared to LM baselines. However, in ChakraCore, there was only a marginal difference in coverage between Montage and CovRL-Fuzz, which can be attributed to Montage's strategy of importing and executing code from its own test suite corpus, achieving higher coverage as a result. In this respect, without the code import feature (Montage w/o Import), CovRL-Fuzz recorded significantly higher coverage than Montage. Additionally, as shown in Figure 4, while Montage's coverage tends to converge over time, CovRL-Fuzz's coverage continues to increase. It suggests that CovRL-Fuzz is likely to obtain more coverage than Montage as time progresses. Note that, while other LM baselines did not account for training time, CovRL-Fuzz included the time required for CovRL-based finetuning during the experiment. Additionally, we observed that CovRL-Fuzz continues to increase coverage even near the 24 hour mark. It displays its effectiveness in obtaining coverage.

Syntax and Semantic Correctness. CovRL-Fuzz is not a grammar-level fuzzing approach that post-processes for syntax and semantic validity. However, it is assumed that CovRL-Fuzz, which uses RL-based finetuning driven by the reward signal derived from testing results, can achieve higher validity compared to random fuzzing (Token-Level AFL). To verify the assumption, we evaluate the error rate of unique test cases.

The experimental results are shown in Table 3. CovRL-Fuzz demonstrated a lower error rate than Token-Level AFL for all JavaScript interpreters. Furthermore, CovRL-Fuzz showed a lower error rate in comparison to most of the fuzzers. While it did not achieve the lowest error rate in JavaScriptCore (JSC) and ChakraCore (Chakra), CovRL-Fuzz still induced a significantly low error rate compared to the most of baselines. Please note that the high error rate of Montage (w/o Import) is due to its inability to access functions from other test suites. For a more detailed analysis of the error rate, we analyzed the types of errors triggered by fuzzers on V8, which is the largest and most dependable JavaScript interpreter, as shown in Figure 5. The results showed that CovRL-Fuzz triggered fewer syntax errors in comparison to coverage-guided baselines. Furthermore, it also produced less syntax and semantic errors than LM baselines, even without using the post-processing techniques used by COMFORT and Montage. These results indicate that CovRL-Fuzz is successful in reducing error rates exclusively through CovRL, without requiring heuristic post-processing.

Finding bugs. To determine whether the coverage improvement and low error rate achieved by CovRL-Fuzz aid in detecting bugs, we conducted experiments by JavaScript interpreters compiled in debug mode with ASAN. We relied on the output reports generated

Table 5: Comparison with state-of-the-art fuzzer using LLM-based mutation.

Target	Fuzzer	Error (%)	Coverage		Improv Ratio (%)
			Valid	Total	
V8	Fuzz4All	**48.56%**	57,524	60,153	30.88%
	CovRL-Fuzz	48.68%	**75,240**	**78,729**	-
JSC	Fuzz4All	60.16%	47,705	48,765	32.76%
	CovRL-Fuzz	**48.59%**	**61,137**	**64,738**	-
Chakra	Fuzz4All	**50.77%**	97,723	99,329	12.25%
	CovRL-Fuzz	54.87%	**105,121**	**111,498**	-
Jerry	Fuzz4All	72.30%	18,895	22,681	2.49%
	CovRL-Fuzz	**58.84%**	**20,844**	**23,246**	-

Table 6: Unique bugs discovered by CovRL-Fuzz and compared LLM-based fuzzer.

Target	Bug Type	Fuzz4All	CovRL-Fuzz
JSC	Undefined Behavior		✓
JSC	Out-of-bounds Read		✓
Chakra	Undefined Behavior		✓
Chakra	Out of Memory		✓
Chakra	Out of Memory		✓
Jerry	Undefined Behavior	✓	✓
Jerry	Memory Leak	✓	✓
Jerry	Undefined Behavior		✓
Jerry	Undefined Behavior		✓
Jerry	Heap Buffer Overflow		✓
Jerry	Out of Memory		✓
Jerry	Stack Overflow		✓
Jerry	Undefined Behavior		✓
Jerry	Heap Buffer Overflow	✓	✓
Total		3	14

by ASAN for stack trace analysis to eliminate duplicate bugs. We also manually analyzed and categorized the results by bug types.

Table 4 shows the number and types of unique bugs found by the CovRL-Fuzz and the compared fuzzers. CovRL-Fuzz discovered the most unique bugs compared to other fuzzers. In detail, CovRL-Fuzz found 14 unique bugs and 9 of these bugs were exclusively detected by CovRL-Fuzz, including stack overflow and heap buffer overflow. These results highlight its effectiveness in bug detection. As observed in the experimental results, LM-based fuzzers, despite achieving higher coverage, tend to find fewer bugs, while heuristic fuzzers, although achieving lower coverage, generally find more bugs. However, irrespective of this trend, CovRL-Fuzz demonstrated superior performance in effectively discovering the most bugs.

5.3 RQ2. Comparison of Fuzzers Using LLM-Based Mutation

To answer **RQ2**, we conducted 24-hour experiments with Fuzz4All, a state-of-the-art LLM-based fuzzing technique using mutation by prompt applicable to a compiler. While TitanFuzz [12] and FuzzGPT [13] both employ LLM-based mutations, their use of handcrafted annotations, prompts, and mutation patterns is specifically designed for deep learning libraries. The specialization makes them challenging to easily adopt for JavaScript interpreters, which is why they were not included in our experimental subjects.

Table 5 presents the results of comparing the coverage and error rate between CovRL-Fuzz and Fuzz4All [65]. While the coverage and error rates of Fuzz4All and CovRL-Fuzz did not show significant differences, it is difficult to assert that the coverage improvement and error rate achieved by Fuzz4All have significantly contributed to finding bugs. Table 6 shows the bugs found by both Fuzz4All and CovRL-Fuzz. The bugs identified by Fuzz4All were only a few of the subsets of those discovered by CovRL-Fuzz. These results mean that our method, which combines coverage-guided fuzzing and LLM-based mutation through CovRL, is more useful in bug detection than the state-of-the-art LLM-based fuzzing technique.

5.4 RQ3. Ablation Study

To answer **RQ3**, we conducted an ablation study on the two key components of CovRL-Fuzz, CovRL and CWR, based on coverage-guided fuzzing, with a 5-hour timeout. Table 7 shows the error rates and coverage according to different CovRL and CWR variants.

Impact of CovRL. To evaluate the impact of CovRL, we conducted a comparison with w/o LLM (TokenAFL [50]), which uses token-level heuristic mutation, LLM w/o CovRL, which simply applies LLM-based mutation to coverage-guided fuzzing, and LLM w/-CovRL, which represents our CovRL-Fuzz. The experimental results indicated that while the LLM w/o CovRL successfully decreased the error rate in comparison to w/o LLM, it did not significantly improve or even slightly decreased coverage. In contrast, the LLM w/CovRL demonstrated coverage improvement for all targets. These findings suggest that applying LLM-based mutation with CovRL in coverage-guided fuzzing is effective.

Impact of CWR. To evaluate the impact of CWR used in CovRL-compared to other rewards, we included w/o RL, and two additional reward processes in our experiments. For rewarding, we conducted a comparison between two types of rewards: Coverage Reward (CR) and Coverage-Rate Reward (CRR). CR is a simple binary rewarding process we designed, where a reward of 1 is given to test cases that find new coverage, and a penalty of 0 is assigned to those that do not. CRR, on the other hand, is a reward used in traditional RL-based fuzzing techniques [5, 32, 33], calculated as the ratio of current coverage to the total cumulative coverage.

In the experimental results, w/CR and w/CRR showed little to no increase in coverage compared to w/o RL. However, CovRL-Fuzz achieved the highest coverage in both valid and total, and exhibited a low error rate. These results suggest that the coverage-guided fuzzing technique using CWR effectively contributes to improving coverage with LLM-based mutation.

5.5 RQ4. Real-World Bugs

To answer **RQ4**, we evaluated the ability of CovRL-Fuzz to find real-world bugs during a specific period of fuzzing. Specifically, we investigated how many real-world bugs CovRL-Fuzz can find and whether it can discover previously unknown bugs. Thus, we evaluated whether CovRL-Fuzz can find real-world bugs for 2 weeks for each target. We tested the latest version of each target interpreter as of January 2023. Table 8 summarizes the bugs found by CovRL-Fuzz. A total of 58 bugs were identified, of which 50 were previously unknown bugs (15 have been registered as CVEs). Out of the discovered bugs, 45 were confirmed by developers, and 18 have been fixed. The CVEs we identified have an average risk score of 7.5 according to CVSS v3.1, with some reaching as high as 9.8.

Table 7: The ablation study with each variants. The Improv (%) refers to the improvement ratio compared to the baseline.

Target	V8				JavaScriptCore				ChakraCore				JerryScript			
Variants	Error (%)	Coverage Valid	Coverage Total	Improv (%)	Error (%)	Coverage Valid	Coverage Total	Improv (%)	Error (%)	Coverage Valid	Coverage Total	Improv (%)	Error (%)	Coverage Valid	Coverage Total	Improv (%)
Impact of CovRL																
w/o LLM	88.79%	44,705	53,936	-	87.45%	35,406	37,461	-	78.98%	81,393	83,785	-	87.39%	12,312	14,795	-
LLM w/o CovRL	62.68%	55,459	56,576	4.89%	55.40%	41,523	42,385	13.14%	**45.25%**	86,043	86,858	3.67%	78.48%	12,833	14,068	-4.91%
LLM w/CovRL	**61.53%**	**71,319**	**74,574**	**38.26%**	**49.60%**	**56,370**	**58,340**	**55.74%**	58.42%	**96,257**	**98,221**	**17.23%**	**58.59%**	**17,481**	**19,855**	**34.20%**
Impact of CWR																
w/o RL	62.68%	55,459	56,576	-	55.40%	41,523	42,385	-	**45.25%**	86,043	86,858	-	78.48%	12,833	14,068	-
CovRL w/CR	71.77%	55,678	57,735	2.05%	53.00%	47,116	49,083	15.80%	67.50%	92,465	94,145	8.39%	73.35%	16,689	18,629	32.42%
CovRL w/CRR	74.15%	57,401	61,331	8.40%	69.57%	37,230	43,369	2.32%	65.47%	91,427	94,785	9.13%	75.34%	16,118	18,584	32.10%
CovRL w/CWR	**61.53%**	**71,319**	**74,574**	**31.81%**	**49.60%**	**56,370**	**58,340**	**37.64%**	58.42%	**96,257**	**98,221**	**13.08%**	**58.59%**	**17,481**	**19,855**	**41.14%**

```
1  function i ( t ) { }
2  async function n ( t ) {
3      if ( t instanceof i ) {
4          let c = await i ( ) ;
5          await c >> i ( n ) ;
6      } else {
7          var c = await n ( ) ;
8      }
9  }
10 n ( true ) ;
```

a) The test case that triggers an out-of-bounds read on ChakraCore 1.13.0.0-beta (#13).

```
1  class s extends WeakMap {
2      static {} ;
3      }
4  function f ( )
```

b) The test case that triggers a heap buffer overflow on JerryScript 3.0.0 (#24).

Figure 6: On bugs found by CovRL-Fuzz

Table 8: The summary of bugs found by CovRL-Fuzz.

Target	Version	Total	Confirmed	Unknown (Fixed)	CVE
V8	11.4.73	2	2	0 (0)	0
JavaScriptCore	2.38.1	3	3	1 (1)	0
ChakraCore	1.13.0.0-beta	9	8	7 (0)	0
JerryScript	3.0.0	14	2	12 (0)	10
QuickJS	2021-03-27	2	2	2 (1)	1
Jsish	3.5.0	4	4	4 (0)	2
escargot	bd95de3c	21	21	21 (14)	0
Espruino	2v20	2	2	2 (2)	2
hermes	0.12.0	1	1	1 (0)	0
Total		58	45	50 (18)	15

CovRL-Fuzz found a variety of bugs including undefined behaviors like assertion failures as well as memory bugs such as buffer overflow and use after free. Please refer to Appendix for a detailed list of bugs found by CovRL-Fuzz. Note that the experiment was carried out using only 3 cores and for a relatively short duration. In contrast, other fuzzing techniques have utilized an average of around 30 cores and conducted their experiments over a whole month [29, 50, 61, 67]. Despite these significant constraints, CovRL-Fuzz was still able to find a substantial number of unknown bugs. This suggests that CovRL-Fuzz demonstrated effectiveness in finding real-world bugs within JavaScript interpreters.

Case Study. Figure 6a represents a minimized test case generated by CovRL-Fuzz. The code triggered an out-of-bounds read bug in the ChakraCore 1.13.0, causing an abnormal termination of the JavaScript interpreter. The original seed does not assign `await` to `var c`. CovRL-Fuzz changed it to `var c=await n();` and added the `await` statement on line 5, and also changed the condition of the `if` conditional. This caused the logic to call `await n();` repeatedly, which ultimately led to the bug.

Figure 6b represents a minimized test case generated by CovRL-Fuzz, causing a heap buffer overflow in the release version of JerryScript 3.0.0. The bug occurs when a function declaration comes on the line following the declaration of a static initialization block in a class. When the parser read the statement, it didn't correctly distinguish the static initialization block range. As a result, memory corruption occurred when parsing the function statement. In contrast to other fuzzing tools, CovRL-Fuzz is grammatically somewhat free and allows for context-aware mutation. This feature led to the discovery of the bug. Our case study further demonstrates the effectiveness of CovRL-Fuzz in detecting real-world bugs.

6 Related Work

LLM-based Fuzzing. Although not related to JavaScript interpreter fuzzing, there have been proposals to test python deep learning libraries using LLMs [12, 13]. TitanFuzz [12] utilized Codex [8] for seed generation and Incoder [17] for mutation by mask. It also aimed to select interesting seeds, not by using internal program information such as coverage, but by utilizing the static analysis information of the seeds. FuzzGPT [13] was designed to guide mutations around buggy functions by either training on GitHub bug report data or utilizing snippets from such reports.

RL-based Fuzzing. RL-based fuzzing approaches [5, 32, 33] seek to enhance performance by incorporating code coverage feedback into deep learning models, such as deep neural networks (DNNs) and recurrent neural networks (RNNs), rather than relying solely on coverage for seed selection. They provide feedback using each coverage as a reward. For the purpose, they process the data into a quantified reward, which is the ratio of the current coverage relative to the total cumulative coverage. We previously referred to this reward as the Coverage-Rate Reward (CRR).

Neural Network-based coverage guidance. Similar to our approach, there have been attempts to use incremental learning for

coverage guidance, such as NEUZZ [54] and MTFuzz [53]. However, unlike our method, they use neural networks to identify mutation positions that influence branching behaviors. Consequently, NEUZZ and MTFuzz face the same limitations as traditional fuzzing when applied to software requiring strict grammar adherence, as they do not directly control the mutations themselves. In contrast, our approach leverages an LLM to mutate the seeds directly, effectively addressing these constraints. By allowing the LLM to handle mutations, we ensure that the generated inputs adhere to grammatical rules to a significant extent.

7 Discussion

We discuss three properties of CovRL-Fuzz in the following:

Time spent between fuzzing and finetuning. As mentioned in the experimental setup, we calculated the fuzzing time to ensure fairness in fuzzing, including the time spent on finetuning in the experimental results. On average, finetuning occurs for 10 minutes every 2.5 hours of fuzzing. Despite including the finetuning time in the experiment, CovRL-Fuzz achieved high coverage while also decreasing the error rate.

Performance bottleneck. In our experimental environment, we observed that CovRL-Fuzz experienced approximately twice the delay in comparison to Token-Level AFL. While the degree of the discrepancy may vary depending on the target software, our results were generally consistent. Specifically, we found that 73% of the time required to generate a single test case was consumed by the LLM-based mutation process, potentially creating a performance bottleneck. Despite this, our experimental results demonstrated that even though the mutation speed was significantly slowed by the LLM-based approach, it still achieved higher performance. It indicates the efficiency of our methodology, suggesting that the performance improvements brought by the LLM-based mutations outweigh the drawbacks of the increased processing time.

Catastrophic forgetting for finetuning. We mitigated catastrophic forgetting by applying two simple strategies: finetuning with a small learning rate and using some of the original seed data for finetuning. These measures have prevented forgetting in our experiments. However, we cannot entirely rule out the possibility of future occurrences. This issue is not unique to our work but is a general problem associated with finetuning for LLMs. We believe that using the latest prevention techniques [26] could more effectively address this issue in the future.

Supporting other targets. Through finetuning, the core idea of guiding coverage information directly with the LLM-based mutator is actually language-agnostic, which suggests its applicability to other language interpreters or compilers. However, our focus was more on analyzing the suitability of our idea to existing techniques than supporting various languages. Therefore, we conducted experiments only on JavaScript interpreters, which we deemed to have the most impact. Extending to other targets is left as future work.

Application to other LLMs. CovRL-Fuzz is not limited to a specific LLM but is a general strategy applicable to other open-source LLMs as well. Among the base models listed in Table 1, we chose CodeT5+ [63] for our experiments because it showed the most promising results despite being the smallest in size. Due to computational resource constraints, we conducted experiments using LLMs with a maximum size of 1B. Even with these constraints, our technique is designed to be model-agnostic, and we believe it can be effectively applied to larger LLMs. We leave this exploration for future work.

8 Conclusion

We introduced CovRL-Fuzz, a novel LLM-based coverage-guided fuzzing framework that integrates coverage-guided reinforcement learning for the first time. Our approach directly integrates coverage feedback into LLM-based mutation, enhancing coverage-guided fuzzing to reduce syntax limitations while enabling effective testing to achieve broader and hidden path exploration of the JavaScript interpreter. Our evaluation results affirmed the superior efficacy of the CovRL-Fuzz methodology in comparison to existing fuzzing strategies. Impressively, it discovered 58 real-world security-related bugs with 15 CVEs in JavaScript interpreters — among these, 50 were previously unknown bugs. We believe that our methodology paves the way for future studies focused on harnessing LLMs with coverage feedback for software testing.

Acknowledgments

We deeply thank the anonymous reviewers for their helpful comments and suggestions. This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2024-00439762, Developing Techniques for Analyzing and Assessing Vulnerabilities and Tools for Confidentiality Evaluation in Generative AI Models).

References

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *Proceedings 2019 Network and Distributed System Security Symposium*. 15 pages. https://doi.org/10.14722/ndss.2019.23412

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. arXiv:2108.07732 [cs.PL]

[3] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-picking: Differential fuzzing of JavaScript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 351–364.

[4] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. 2019. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. 1985–2002.

[5] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. 2018. Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 116–122.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in neural information processing systems*, Vol. 33. 1877–1901.

[7] Charlie Miller. 2008. Fuzz by number. https://www.ise.io/wp-content/uploads/2019/11/cmiller_cansecwest2008.pdf. Accessed: 2024-01-12.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374 [cs.LG]

[9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. arXiv:2204.02311 [cs.CL]

[10] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. arXiv:2110.14168 [cs.LG]

[11] 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*. 95–105.

[12] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*.

[13] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. arXiv:2304.02014 [cs.SE]

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805 [cs.CL]

[15] ECMA International. 1997. ECMAScript language speicification. https://www.ecma-international.org/ecma-262/. Accessed: 2023-08-15.

[16] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.

[17] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*.

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

[19] Google. 2017. Chrominum Issue 729991. https://bugs.chromium.org/p/chromium/issues/detail?id=729991. Accessed: 2023-08-14.

[20] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities.. In *NDSS*.

[21] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.. In *Proceedings 2019 Network and Distributed System Security Symposium*. 15 pages. https://doi.org/10.14722/ndss.2019.23263

[22] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, et al. 2021. Sofi: Reflection-augmented fuzzing for javascript engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2229–2242. https://doi.org/10.1145/3460120.3484823

[23] hoongwoo Han. 2010. js-vuln-db. https://github.com/tunz/js-vuln-db. Accessed: 2023-08-15.

[24] JSHint. 2013. JSHint: A JavaScript Code Quality Tool. https://jshint.com/. Accessed: 2023-08-15.

[25] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. ACM, 2123–2138. https://doi.org/10.1145/3243734.3243804

[26] Suhas Kotha, Jacob Mitchell Springer, and Aditi Raghunathan. [n. d.]. Understanding Catastrophic Forgetting in Language Models via Implicit Inference. In *The Twelfth International Conference on Learning Representations*.

[27] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems* 35 (2022), 21314–21328.

[28] Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Lu, Thomas Mesnard, Colton Bishop, Victor Carbune, and Abhinav Rastogi. 2023. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. arXiv:2309.00267 [cs.CL]

[29] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language {Model-Guided}{JavaScript} Engine Fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2613–2630. https://www.usenix.org/system/files/sec20-lee-suyoung.pdf

[30] 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*. 475–485.

[31] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[32] Xiaoting Li, Xiao Liu, Lingwei Chen, Rupesh Prajapati, and Dinghao Wu. 2022. ALPHAPROG: reinforcement generation of valid programs for compiler fuzzing. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 12559–12565.

[33] Xiaoting Li, Xiao Liu, Lingwei Chen, Rupesh Prajapati, and Dinghao Wu. 2022. FuzzBoost: Reinforcement Compiler Fuzzing. In *Information and Communications Security: 24th International Conference, ICICS 2022, Canterbury, UK, September 5–8, 2022, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 359–375.

[34] Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023. RLTF: Reinforcement Learning from Unit Test Feedback. arXiv:2307.04349 [cs.AI]

[35] 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*. 1044–1051. https://doi.org/10.1609/aaai.v33i01.33011044

[36] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[37] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[38] Jasiel Spelman Matt Molinyawe, Adul-Aziz Hariri. 2016. $hell on Earth: From Browser to System Compromise. In *Black Hat USA*.

[39] Michal Zalewski. 2013. AFL: American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. Accessed: 2023-08-15.

[40] Mihai Bazon. 2010. uglifyjs. https://github.com/mishoo/UglifyJS. Accessed: 2023-08-14.

[41] Barton P Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279

[42] OpenAI. 2024. gpt4. https://openai.com/gpt-4. Accessed: 2024-03-22.

[43] OpenAI. 2024. OpenAI API. https://openai.com/index/openai-api. Accessed: 2024-07-12.

[44] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[45] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1629–1642. https://doi.org/10.1109/SP40000.2020.00067

[46] Jibesh Patra and Michael Pradel. 2016. *Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data*. Technical Report. TU Darmstadt, Department of Computer Science.

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

[48] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (Jan. 2020), 5485–5551. https://dl.acm.org/doi/abs/10.5555/3455716.3455856

[49] Paul Roit, Johan Ferret, Lior Shani, Roee Aharoni, Geoffrey Cideron, Robert Dadashi, Matthieu Geist, Sertan Girgin, Léonard Hussenot, Orgad Keller, et al. 2023. Factually Consistent Summarization via Reinforcement Learning with Textual Entailment Feedback. arXiv:2306.00186 [cs.CL]

[50] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. 2021. {Token-Level} Fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2795–2809. https://www.usenix.org/system/files/sec21-salls.pdf

[51] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. arXiv:1707.06347 [cs.LG]

[52] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318.

[53] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. 2020. MTFuzz: fuzzing with a multi-task neural network. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 737–749.

[54] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 803–817. https://doi.org/10.1109/SP.2019.00052

[55] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. arXiv:2301.13816 [cs.LG]

[56] Karen Sparck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* 28, 1 (1972), 11–21.

[57] Technical Committee 39 ECMA International. 2010. Test262. https://github.com/tc39/test262. Accessed: 2023-08-15.

[58] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21*. Springer, Cham, 581–601. https://doi.org/10.1007/978-3-319-45744-4_29

[59] W3Techs. 2024. Usage statistics of JavaScript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript. Accessed: 2024-01-17.

[60] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594. https://doi.org/10.1109/SP.2017.23

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

[62] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. {FuzzJIT}:{Oracle-Enhanced} Fuzzing for {JavaScript} Engine {JIT} Compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1865–1882.

[63] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. arXiv:2305.07922 [cs.CL]

[64] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned Language Models are Zero-Shot Learners. In *International Conference on Learning Representations*.

[65] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748* (2023).

[66] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.

[67] 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*. ACM, 435–450. https://doi.org/10.1145/3453483.3454054

[68] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 21–29.

Received 2024-04-12; accepted 2024-07-03