Large Language Models Based JSON Parser Fuzzing for Bug Discovery and Behavioral Analysis

ZHIYUAN ZHONG, ZHEZHEN CAO, and ZHANWEI ZHANG, SUSTech, China

Fuzzing has been incredibly successful in uncovering bugs and vulnerabilities across diverse software systems. JSON parsers play a vital role in modern software development, and ensuring their reliability is of great importance. This research project focuses on leveraging Large Language Models (LLMs) to enhance JSON parser testing. The primary objectives are to generate test cases and mutants using LLMs for the discovery of potential bugs in open-source JSON parsers and the identification of behavioral diversities among them. We aim to uncover underlying bugs, plus discovering (and overcoming) behavioral diversities.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Large Language Models, JSON, Fuzzing

1 INTRODUCTION

JSON (JavaScript Object Notation) is a crucial data interchange format due to its simplicity and versatility, playing a vital role in modern software development. It's platform-independent, human-readable, and easy to work with, making it ideal for data exchange between different systems and languages. When parsing JSON in Java, potential intricacies may arise. Handling nested structures, ensuring data type consistency, and managing errors in JSON data can be challenging. Developers need to be cautious about potential issues like null values, unexpected data structures, and encoding/escaping problems. Selecting a robust JSON parsing library and understanding its features is crucial for effectively working with JSON data in Java, as it can impact both the efficiency and reliability of data processing in software applications.

Large Language Models (LLMs) has had a transformative impact on downstream software engineering tasks. Its advanced NLP capabilities have accelerated code development and maintenance, enabling developers to write code efficiently. LLMs have aided tasks such as code completion[3], bug detection[11], and automated program repair[5].

This project aims to integrate Large Language Models (LLMs) into JSON parser fuzzing framework, which can bring substantial value to the process of testing and securing software systems. LLMs can automatically generate a wide range of complex and realistic JSON payloads, which are invaluable for uncovering vulnerabilities and edge cases that may be missed with manually crafted test data, ultimately contributing to improved JSON parser reliability.

1.1 Background and Significance

JSON has rapidly become the de facto standard for data interchange on the web due to its simplicity, versatility, and platform independence. However, as JSON's ubiquity has grown, so has the importance of effectively parsing and interpreting it. Incorrect or vulnerable JSON parsers can lead to issues in software applications, ranging from minor inconveniences to severe security vulnerabilities. With the rise of cyber-attacks targeting data structures and communication, ensuring the robustness of JSON parsers is crucial. Given the critical role that JSON parsers play in modern software systems, there is a significant need to ensure their reliability and security.

Historically, research has shown the discovery of vulnerabilities in commonly used software components, such as JSON parsers, can lead to widespread exploitation, as these components are

Authors' address: Zhiyuan Zhong, zhongzy2021@mail.sustech.edu.cn; Zhezhen Cao; Zhanwei Zhang, SUSTech, 1088 Xueyuan Venue, Shenzhen, China.

reused across countless applications. Therefore, improving the robustness of JSON parsers is not just beneficial for individual applications, but for the entire software ecosystem.

1.2 Problem Description

The core problem to be addressed is the potential existence of undiscovered bugs and vulnerabilities in open-source JSON parsers. Such issues might lead to incorrect data interpretation, application crashes, or even potential security threats like code injections.

When parsing JSON, intricacies such as handling nested structures, ensuring data type consistency, and managing errors in JSON data emerge. Challenges like null values, unexpected data structures, and encoding/escaping problems further complicate the task. Given these complexities, it is imperative to develop comprehensive testing and validation techniques to ensure the reliability and security of JSON parsers.

The flexible nature of JSON standard results in the behavioral differences between different implementations of JSON parser, which are not negligible since different software applications might adopt different JSON libraries. The key problem is to uncover the underlying bugs while considering the behavioral diversities among different JSON parser implementations.

The main directions of our exploration can be summarized as the following:

(1) Utilize LLMs to enhance the generation of diverse test cases.
(2) Identify anomalies and bugs that surpass the scope of capturing behavioral differences.

1.3 Evaluation Metrics for Parsers

To measure and evaluate a JSON parser, we will consider the following evaluation metrics:

- Error Handling: Evaluating how the a parser handle error scenarios specified by the JSON specification, such as invalid syntax or unsupported data types.
- Performance: Measuring parsing speed and resource consumption (e.g., memory usage) of each parser to analyze their efficiency.
- Conformance to JSON Specification: Verifying the parser's adherence to the JSON specification in terms of supported features and compliance with standards.
- (In this project) How does one parser differs from other parsers?

2 LITERATURE OVERVIEW

2.1 Fuzzing and Testing

Fuzzing is an automated software testing technique that involves providing a program with a large volume of random or unexpected inputs to discover vulnerabilities and defects. Parser fuzzing, therefore, involves generating and feeding malformed or unexpected data to a parser to assess its robustness and identify vulnerabilities.

- *Mutation-Based*: Similar to traditional fuzzing, mutation-based fuzzing involves generating inputs by slightly modifying existing valid inputs. These modifications can include bit flips, byte substitutions, or other simple transformations.
- *Generation-Based*: In contrast to mutation-based fuzzing, generation-based fuzzing constructs inputs from scratch based on a defined grammar or specification. It aims to generate more structured and valid inputs while still exploring potential vulnerabilities.
- *Grammar-Based*: Grammar-based fuzzing introduces structured inputs based on the expected format of the data being parsed. This approach can discover vulnerabilities that traditional fuzzing may overlook, as it adheres to the expected syntax of the data.
- *Coverage-Guided*: Coverage-guided fuzzing, using techniques like AFL (American Fuzzy Lop), instruments the target parser to monitor code coverage and guide the fuzzer to explore

untested code paths. This has proven to be highly effective in identifying vulnerabilities and increasing the efficiency of parser fuzzing.

- *Differential Fuzzing*: Differential fuzzing involves comparing the parsing results of multiple parsers designed for the same format or protocol. Discrepancies can indicate potential vulnerabilities.

2.2 Large Language Models

Recent advancements in the field of natural language processing (NLP) have led to the widespread adoption of Large Language Models (LLMs) for a wide range of tasks, encompassing both natural language and code-related applications. State-of-the-art LLMs are primarily based on transformer architectures and can be categorized into three main types: decoder-only models (e.g., GPT-3 and StarCoder), encoder-only models (e.g., BERT and CodeBERT), and encoder-decoder models (e.g., BART and CodeT5). More recently, instruction-based LLMs (e.g., ChatGPT and GPT-4) and LLMs fine-tuned using reinforcement learning from human feedback (RLHF) have demonstrated a capacity to comprehend and execute complex instructions effectively.

LLMs are typically customized for specific tasks through fine-tuning or by providing prompts. Fine-tuning involves further training the model on task-specific data, which can become costly and challenging as LLM sizes continue to increase. In contrast, prompting provides the model with a task description and, optionally, a few examples, without explicitly updating model weights. The selection of the input prompt is known as prompt engineering, where users experiment with different instructions.

2.3 Behavioral Diversity

The popularity of JSON has fueled the development and maintenance of multiple libraries that all provide services to process JSON files. While the format is thoroughly specified in RFC 8259, the specification leaves significant room for choice when implementing a specific library to process JSON. Harrand et.al[7] had observed a remarkable behavioral diversity between java json libraries with ill-formed files, or corner cases such as large numbers or duplicate data.

3 EVALUATION

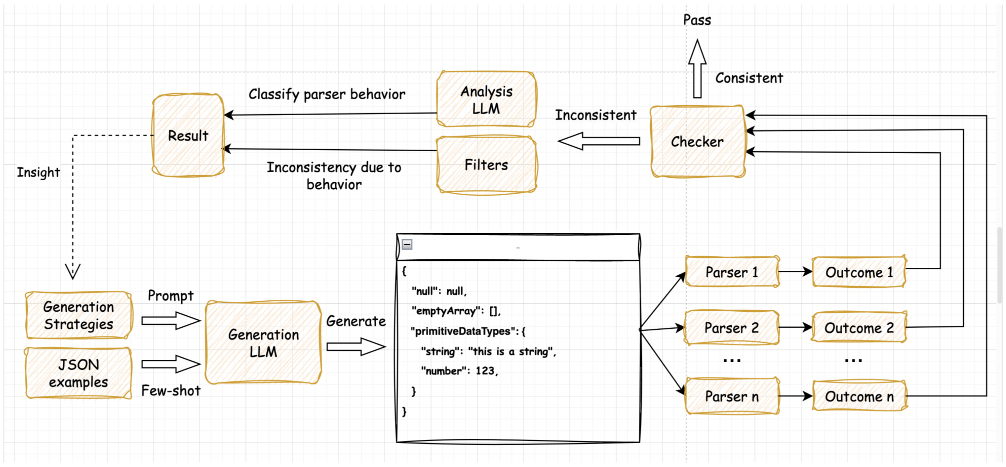
We will use the following metrics to evaluate the findings of our project:

(1) Number and Severity of Bugs Discovered. Enhanced performance is indicated by the detection of a greater number of (unique) bugs.
(2) Code coverage. It indicates the percentage of code that is executed or covered during automated testing. The goal of is to ensure that as much code as possible is exercised during testing, aiming to improve the quality and reliability of the codebase.
(3) Insight of behavioral differences. We aim to summarize the differences (either obvious or potential) between Java JSON parsers, which helps developers gain a comprehensive understanding of how different JSON parsers behave. For example, when choosing a JSON parser for a particular project or use case under the consideration of compatibility, performance, etc.

4 A PROPOSED APPROACH AND PROJECT FRAMEWORK

We presents JFuzz, an approach focuses on leveraging Large Language Models (LLMs) to enhance JSON parser testing. JFuzz is capable of generating JSON test cases and mutants using LLMs for the discovery of potential bugs and the identification of behavioral diversities among parsers.

JFuzz employs innovative prompt-engineering techniques, such as chain-of-thought and few-shot prompting, to guide LLMs in producing a wide array of test cases. The few-shot JSON examples are derived from existing JSON parsing test suites available on the Internet[8], as well as from user-reported instances on platforms like StackOverflow or GitHub. Additionally, JSON specifications documentation serves as supplementary domain knowledge. Constructing prompt patterns involves adapting approaches from prior works[6][10]. Employing differential testing, JFuzz utilizes LLM-generated seed JSON files as inputs across multiple parsers. Subsequently, it evaluates the parser output to ensure compliance with the JSON standard format and assesses the consistency among outputs from all parsers.



```
{
  "null": null,
  "emptyArray": [],
  "primitiveDataTypes": {
    "string": "this is a string",
    "number": 123,
  }
}
```

Fig. 1. Overview of Fuzzer

Due to the loosely defined nature of JSON, implementations of JSON parsers often exhibit variability, even in basic scenarios like handling null values. Consequently, inconsistencies among parsers' outputs primarily stem from their individual features and strategies, rather than specific parser bugs. Behavioral diversities encompass various aspects, including the treatment of null fields, acceptance of trailing commas, and number precision, etc. To discern potential bugs amidst these diversities, we propose a sampling method aimed at reducing noise generated by false positives (inconsistencies attributed to behavioral diversities) in section 4.3. This method enables us to focus on inconsistencies more likely to indicate genuine bugs. JFuzz selectively samples JSON test cases that lead to 'inconsistencies' among parsers, along with their corresponding parsing results for the same JSON input. Employing LLMs with human evaluation, inconsistencies arising from minor differences in parser features are filtered out. This iterative process continues in subsequent fuzzing loops.

Update*: A new idea is proposed in section 4.4.

4.1 Test case Generation

Prompt-Engineering has been demonstrated as a powerful method to improve the quality of LLM's responses in many downstream tasks.[9] Few-shot prompting is one of the most effective one which provides few-shot examples to fine-tune LLM's output as desired. For example, target library and API signature as few-shot examples in fuzzing python libraries[4]. JFuzz utilized the existing JSON test suites from the paper[7], which consists of 206 files that are syntactically correct, according to the JSON grammar specified in RFC 8259, and 267 ill-formed JSON files that include some structural errors.

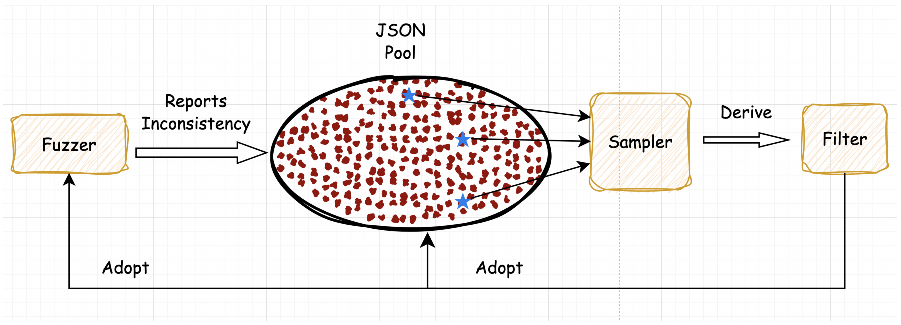


Fig. 2. Overview of Sampler

4.1.1 Method A: Sampling few-shot prompting. Given the JSON test suites S, JFuzz randomly samples k JSON files as examples from S at each fuzzing loop. The k examples may be well-formed or ill-formed, thus when providing the prompt, labeling the example as well-formed or ill-formed is required. Each examples also have its features. For example, the JSON $[-0.00000...00001]$ tests parser's handling of a double very close to zero. Therefore, JFuzz provides the filename as additional information that indicates the feature of the JSON. Then the LLM are asked to generate a new JSON test case based on the given k examples.

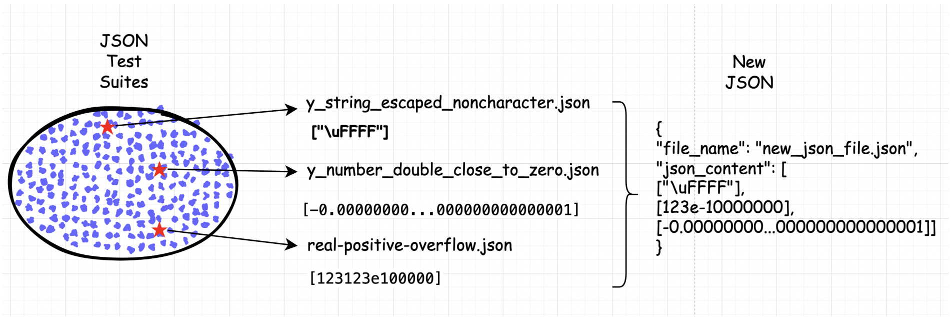


Fig. 3. Few-shot Prompting via sampling

4.1.2 Method B: Sampling + Autoprompting. To be continued.

4.2 Differential Fuzzing Analysis

At each round of parsing, we need to know whether the results are consistent or not. Now we specify the rules for comparison in Section 4.2.1 and Section 4.2.2.

4.2.1 Definition of Equivalence. We adopt the method in the paper [7], according to the following rules: JSON arrays must consist solely of equivalent elements arranged in the same order; JSON objects should contain identical sets of keys, each key corresponding to an equivalent object; strings are considered equivalent if they are strictly identical; numbers are considered equivalent if they match in value and type; and literals are deemed equivalent.

Note*: When judging whether two JSON strings (or files) J_1 and J_2 are equal, we use a third-party JSON Parser P_{check} to parse J_1 and J_2, then compare them using P_{check}'s $.equivalence()$ method.

Another more rigorous and precise way is: sort each JSON object's key set & value set in a certain order (e.g. alphabetical), then serialize the objects and use string comparison.

Assumption: P_{check} is "fair". For 2 JSON string, if they are semantically equivalent, then P_{check} should output the same result. It won't make mistake on 2 equivalent JSON strings.

However, it may make mistake on 2 non-equivalent JSON strings, since itself may accept 2 non-equivalent JSON strings as equivalent (this part needs further discussion).

4.2.2 *Definition of Consistency.*

- For an individual parser, we categorize the outcome of parsing as *pass* or *fail*. A *pass* means the parser accepts the input JSON without throwing any exceptions. And a *fail* means the parser throws unchecked exceptions(e.g. a crash) or catches checked exceptions, that have been anticipated by the developers.
- For all the parsers' output, we categorize the outcome as *consistent* or *inconsistent*.
  - *consistent*: All the parsers reject the input JSON and output a *fail*; Or all of them output a *pass* and the output JSON are equivalent (for now, pass the third-party parser P_{check} check).
  - *inconsistent*: Some parsers reject the input JSON while others accept it, i.e. the output are mixed with *pass* and *fail*; Or all of them are *pass*, but the equivalence check (for now, P_{check}) failed.

For *inconsistent* cases, The test cases are further divided into: *mix* and *check*.

- *mix*: Some parsers pass, some parsers fail on this test case.
- *check*: All parsers pass, but the outputs are different, checked by P_{check}.

4.3 Inconsistency Filtering (Idea 1)

The inconsistency reported by the fuzzer could be numerous, because any small behavioral difference can lead to inconsistent results. Therefore, filtering the results helps us focus on more "valuable" inconsistencies that are more likely to reveal true bugs. The primary workflow is shown in figure 2.

For a JSON pool of n JSON files that have caused inconsistencies, we randomly sample k files from the pool ($k \ll n$) and analyze the similarities between them. For example, for k=20 and n=1000, 15 out of the 20 files contain the value *null*. Further assessment confirms that this is a behavioral feature, as some parsers ignore *null* field by default, while others preserve it. Subsequently a *null* filter is created and applied to the JSON pool. All files with the filtered feature (in the example, *null* field) will be filtered out, reducing the size of n.

The filters are also applied in the fuzzer: 1. Guide the generation LLM to avoid generating test cases of these behavioral features, 2. Identify these features during the *Checker* phase in figure 1, classify them as behavioral features instead of real "inconsistency".

The filters will be updated after each fuzzing experiment in the above manner iteratively, optimizing further fuzzing.

4.4 Inconsistency Analysis (Idea 2, updated)

To filter out behavioral feature by sampling and summarizing (as idea 1) could be difficult. Because of the flexibility of JSON format, identifying the differences between two JSON files requires more than String comparison, but human domain knowledge. For instance, the examples in table 1 are indeed different in String level, but can nearly be viewed as the same in reality.

More difficulties include:

- Hard to summarize the difference as a concrete rule automatically. In table 1, the rule might be *"decimal with trailing zeros and exponential notation does not matter"*, which may be verbose and requires human to derive (LLM may be capable).

- A rule can be hard to be applied in filtering. For example, the rule *"decimal with trailing zeros and exponential notation does not matter"* is hard to be transformed into code for automatic filter purpose. Even it is possible, it can be tedious in programming with lots of conditions. (LLM may be capable via prompting/in-context learning).

Table 1. Example "Inconsistency"

JSON 1	JSON 2	Difference
{"a":100.000}	{"a":100.0}	trailing zeros
{"a":1e2}	{"a":1E+2 }	representation
{"a":1e2}	{"a":100.0}	notation

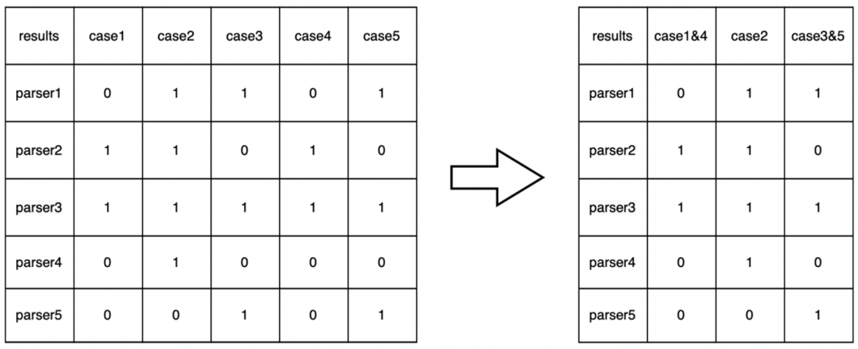
Thus establishing a systematic and automatic mechanism to analyse 2 JSON files inconsistencies is of importance. Our current idea is as follows:

(1) Use difference tools like [1] and [2] to roughly locate the difference parts (Accomplished).
(2) Utilize LLMs to inference/summarize the differences as one category (Exploring).

4.4.1 Inconsistency Classification Based on Grouping. As defined in section 4.2.2, the test cases are divided into: *mix* and *check*. Now we only consider the *mix* case.

During testing, label *pass* as a 1, a *fail* as a 0. For N parsers under test, we can encode the results into a N-dim vector. For example, $(1, 1, 0, 1, 0)$ means parser 1,2,4 pass, parser 3,5 fail. Based on the assumption in section 4.2.1, two same vectors are likely to indicate the same behavior, i.e., two test cases exhibiting similar test outcomes may be attributed to a shared feature present in both test cases.

Subsequently, the organization of test cases into groups according to their respective vectors allows for the extraction of features through group analysis. Furthermore, a higher frequency of a vector (occurrence counts across the test cases), signifies its likelihood as a common behavioral feature.



results	case1	case2	case3	case4	case5
parser1	0	1	1	0	1
parser2	1	1	0	1	0
parser3	1	1	1	1	1
parser4	0	1	0	0	0
parser5	0	0	1	0	1

results	case1&4	case2	case3&5
parser1	0	1	1
parser2	1	1	0
parser3	1	1	1
parser4	0	1	0
parser5	0	0	1

Fig. 4. Grouping illutration

Note*: It it possible that the vectors are the same for 2 test cases, but they actually reveal different behavior. For example, parser p_i fails for test cases with features f_1 and f_2, while others $(p_1, p_2, ..., p_j, ..., p_n (j \neq i))$ all accept them. In this case their behavior vector are identical, which in fact are caused by two different JSON features f_1 and f_2.

It is also possible that the combination of features results in a behavioral vector. For example, test cases that simultaneously possess both f_1 and f_2 may result in a distinctive vector. Thus a unique vector does not necessarily imply an atomic feature.

Besides, a behavioral vector only denotes the parsers' tolerance or approach to a given test case. Among the parsers that accept the test case, divergent behaviors may still occur.

Manual observation is required in these situations. We will figure out the solution.

5 PRELIMINARY RESULT

5.1 Summary of Behavioral Diversity

We have summarized the behavioral diversities among all parsers under test in table 2. The behaviors vary a lot among parsers. Certain parsers exhibit heightened tolerance towards syntactic elements such as comments and trailing commas, while others demonstrate a capacity to accommodate a broader array of string encodings. Furthermore, some parsers display a propensity for accommodating flexible representations of numerical values and wide number range. This summary helps developers to select a JSON parser that fits their project engineering needs.

Table 2. Behavior Summary

category	Test Cases	OrgJson	fastjson	fastjson2	genson	gson	jackson-datab	javax-json	jettison	json-argo	json-io	json-lib	json-simple	jsonij
string/encoding	whitespace_formfeed	yes	yes	yes	yes	no	no	yes	yes	no	no	yes	no	no
	unquoted string	yes	no	yes	no	yes	no	no	yes	no	no	yes	no	no
	single quote string	yes	yes	yes	no	yes	no	no	yes	no	no	yes	no	no
	escaped control character	yes	yes	yes	no	no	no	no	yes	yes	no	yes	yes	no
	escape x: \x00	no	yes	yes	no	no	no	no	yes	no	no	yes	yes	no
	Control character	yes	yes	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	no
	invalid character escape	no	no	no	no	no	no	no	yes	no	no	yes	yes	no
	invalid Unicode surrogate	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no
number	nothing before decimal point	yes	no	no	no	yes	no	no	yes	no	no	yes	no	no
	negative sign before decimal point	yes	yes	yes	yes	yes	no	yes	yes	no	yes	yes	no	no
	exponent/empty after decimal point	yes	yes	yes	yes	yes	no	yes	yes	no	yes	yes	no	no
	hex number	yes	no	no	no	yes	no	no	yes	no	no	yes	no	no
	large integer	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	no	yes
	overflow-exponent exceed 1024	yes	yes	no	yes	yes	yes	no	no	yes	yes	yes	yes	yes
	overflow-exponent within 1024	yes	yes	yes	yes	yes	yes	no	no	yes	yes	yes	yes	yes
	overflow-exponent huge	yes	no	no	yes	yes	yes	no	no	yes	yes	yes	yes	yes
	underflow-exponent exceed 1024	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
	underflow-exponent huge	yes	no	no	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
	overflow-exponent exceed 1024 (in array)	yes	yes	yes	yes	yes	yes	no	yes	yes	yes	yes	yes	yes
	self-increment in array	yes	yes	no	no	yes	no	no	yes	no	no	no	no	no
syntax	comment bewtween key-value	no	yes	no	yes	yes	no	yes	yes	no	no	yes	no	no
	comment inside object	no	yes	yes	yes	yes	no	yes	yes	no	no	yes	no	no
	object trailing comma	yes	yes	yes	no	no	no	no	yes	yes	no	yes	yes	no
	array trailing comma	yes	yes	yes	no	yes	no	no	yes	no	yes	yes	yes	no
	duplicate key	no	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no
unknown	unknown	no	yes	yes	no	no	no	no	no	yes	no	no	yes	no
	unknown	no	yes	yes	yes	yes	no	yes	no	yes	yes	no	yes	no
	unknown	yes	yes	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	no
	unknown	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	yes
	unknown	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	yes
	unknown	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no

5.2 Code Coverage of Parsers

We also test the code coverage rate of two parsers, for exploratory purpose (figure 5).

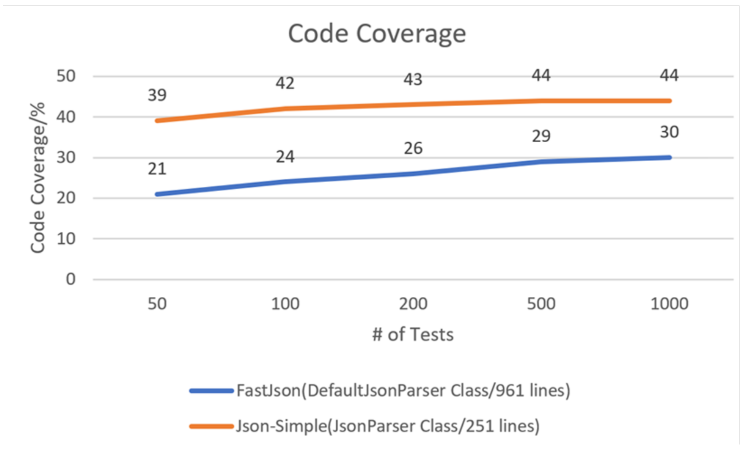


Fig. 5. Code coverage of fastjson and json-simple

6 WORK ACCOMPLISHED AND PROBLEMS

So far, we have finished the following tasks:

(1) Build a LLM-based fuzzing framework that:
  - Supports different LLMs (llama2/CodeLlama).
  - Tests 13 Java JSON parsers.
  - Runs automated fuzzing for a given time budget.

(2) Categorize test cases under the *mix* situation.
(3) Summarize the behaviors of parsers in a relatively coarse-grained manner.
(4) Organize a dataset corresponding to the summary.

Several problems occurred during our exploration: The generation quality of LLM is unstable sometimes such as the issues of duplicated response and irrelevant explanation. The criteria of checking the consistency of parsers output needs further discussion, solely relying on a third-party parser or string comparison is questionable. Give the test cases that caused inconsistency, identifying exact behaviors or potential bugs is hard to automate because of the flexibility of JSON files.

7 FUTURE PLANS

(1) **Primary:** Establish a systematic and (semi) automated mechanism to extract differences between JSON files and categorize them. This is crucial to both discovering behaviors and revealing potential bugs.
(2) Integrate more resource profiling aspects such as parse time and memory consumption into the framework, to measure each parsers performances.
(3) Continue to improve the diversity of test cases.
(4) Monitor code coverage of each JSON library.

REFERENCES

[1] 2023. deepdiff. https://github.com/seperman/deepdiff

[2] 2023. java-diff-utils. https://github.com/java-diff-utils/java-diff-utils

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[4] 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. arXiv:2212.14834 [cs.SE]

[5] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. arXiv:2205.10583 [cs.SE]

[6] Sidong Feng and Chunyang Chen. 2023. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. arXiv:2306.01987 [cs.SE]

[7] Nicolas Harrand, Thomas Durieux, David Broman, and Benoit Baudry. 2021. The Behavioral Diversity of Java JSON Libraries. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. 412–422. https://doi.org/10.1109/ISSRE52982.2021.00050

[8] Nicolas Seriot. 2021. JSONTestSuite. https://github.com/nst/JSONTestSuite Accessed: 2023-10-31.

[9] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. ACM. https://doi.org/10.1145/3540250.3549113

[10] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. arXiv:2302.11382 [cs.SE]

[11] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal Fuzzing via Large Language Models. arXiv:2308.04748 [cs.SE]