LspFuzz: Hunting Bugs in Language Servers

Hengcheng Zhu*, Songqiang Chen*, Valerio Terragni†, Lili Wei‡
Yepang Liu§, Jiarong Wu*, and Shing-Chi Cheung*
*The Hong Kong University of Science and Technology, Hong Kong SAR
†University of Auckland, Auckland, New Zealand, ‡McGill University, Montreal, Canada
§Southern University of Science and Technology, Shenzhen, China
Emails: *{hzhuaq, i9s.chen}@connect.ust.hk, {jwubf, scc}@cse.ust.hk
†v.terragni@auckland.ac.nz, ‡liil.wei@mcgill.ca, §liuyp1@sustech.edu.cn

***Abstract*—The Language Server Protocol (LSP) has revolutionized the integration of code intelligence in modern software development. There are approximately 300 LSP server implementations for various languages and 50 editors offering LSP integration. However, the reliability of LSP servers is a growing concern, as crashes can disable all code intelligence features and significantly impact productivity, while vulnerabilities can put developers at risk even when editing untrusted source code. Despite the widespread adoption of LSP, no existing techniques specifically target LSP server testing. To bridge this gap, we present LspFuzz, a grey-box hybrid fuzzer for systematic LSP server testing. Our key insight is that effective LSP server testing requires holistic mutation of source code and editor operations, as bugs often manifest from their combinations. To satisfy the sophisticated constraints of LSP and effectively explore the input space, we employ a two-stage mutation pipeline: syntax-aware mutations to source code, followed by context-aware dispatching of editor operations. We evaluated LspFuzz on four widely used LSP servers. LspFuzz demonstrated superior performance compared to baseline fuzzers, and uncovered previously unknown bugs in real-world LSP servers. Of the 51 bugs we reported, 42 have been confirmed, 26 have been fixed by developers, and two have been assigned CVE numbers. Our work advances the quality assurance of LSP servers, providing both a practical tool and foundational insights for future research in this domain.**

***Index Terms*—Language Server Protocol, Fuzzing, Software Testing, Developer Tools**

I. Introduction

Modern software development increasingly relies on code intelligence features such as real-time error detection, autocompletion, and refactoring. The Language Server Protocol (LSP) [1] has revolutionized the delivery of these capabilities. At its core, LSP defines a standardized communication protocol in which code editors act as *LSP clients* and language-specific tools serve as *LSP servers* [2], fostering a robust and interoperable ecosystem of development tools. LSP is widely adopted, with approximately 300 LSP server implementations supporting a wide range of programming languages and around 50 editors (e.g., VSCode, Neovim, Cursor, Zed) offering LSP support. Given this widespread adoption, an increasing number of developers have incorporated LSP into their workflows. For example, LSP-related VSCode extensions have attracted millions of installs [3].

*Shing-Chi Cheung is the corresponding author of this paper. §Yepang Liu is affiliated with both the Research Institute of Trustworthy Autonomous Systems and the Department of Computer Science and Engineering.

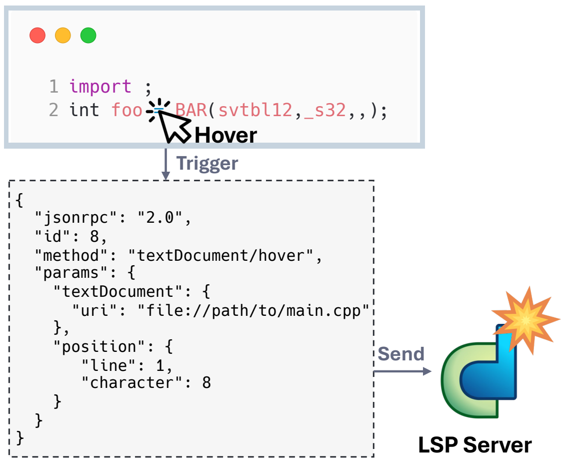

```
import ;
int foo = BAR(svtbl12,_s32,,);
```
```
{
  "jsonrpc": "2.0",
  "id": 8,
  "method": "textDocument/hover",
  "params": {
    "textDocument": {
      "uri": "file://path/to/main.cpp"
    },
    "position": {
      "line": 1,
      "character": 8
    }
  }
}
```

Fig. 1. Hovering over the Equal Sign Causes a Crash in clangd 20.1.4

However, the reliability of LSP servers is a growing concern. Crashes in LSP servers can strip off all code intelligence features, leaving developers unproductive and facing unexpected disruptions. For instance, Fig. 1 illustrates a crash in `clangd` (a C/C++ LSP server by LLVM) [4]. When developers write a variable initialization statement and hover the mouse cursor over the equal sign to view information about the assignment operator, `clangd` unexpectedly crashes. As a result, developers not only fail to obtain the desired information but are also confused by the sudden loss of all code intelligence features. Bugs in LSP servers are prevalent, with `clangd` alone having over 500 issues reporting crashes. Developers express frustration at the disruption caused by such crashes in LSP servers, with some complaining that it *crashed 5 times in the last 5 minutes*[1], while others report the exasperating experience of *losing all its parsing and jump-to-definition functionalities*[2]. Moreover, we found that some memory corruptions in LSP servers may lead to security vulnerabilities, such as remote code execution when processing malicious source files.

Despite the importance of assuring LSP server quality, to the best of our knowledge, no existing techniques specifically

[1]https://www.reddit.com/r/neovim/comments/1hxcs1q
[2]https://github.com/clangd/clangd/issues/2070

target LSP server testing. This motivated us to bridge this gap.

Developing an effective LSP server testing technique requires tackling two unique challenges. First, we must be able to generate test cases that satisfy the combinatorial input constraints (e.g., in Fig. 1, the `position` should point to a valid location in the source file). Without systematic awareness of these constraints, existing binary [5] and grammar-based [6]–[9] fuzzers struggle to penetrate beyond input validation layers to exercise the core functionality of LSP servers. Second, effective testing requires exploring diverse combinations across two key dimensions: 1) the source code being analyzed by the LSP server, and 2) the editor operations performed on that code. The crash in Fig. 1 exemplifies this challenge: the bug only manifests through the interaction between specific code content (i.e., an invalid function call) and a particular editor operation (i.e., hovering over the equal sign). This multiplicative relationship between source code and editor operations demands a testing approach that systematically explores their interaction to enhance bug-finding capability.

To tackle these challenges and test LSP servers effectively, we propose LSPFUZZ, a grey-box hybrid fuzzer designed for LSP servers. Our key insight is that *effective LSP server testing requires the holistic mutation of source code and editor operations*, as bugs often manifest from their specific combinations. In LSPFUZZ, our mutation operators are aware of the sophisticated constraints imposed by the LSP, which enables us to produce test cases that can penetrate the input validation layer and reach the core logic of LSP servers. At the core of LSPFUZZ is a two-stage mutation pipeline that performs context-aware mutation of the source code and editor operations. Specifically, the first stage performs syntax-aware mutations to the source code with the TREE-SITTER [10] grammar, producing diverse source code as a basis. The second stage dispatches editor operations based on the mutated source code to trigger various LSP server behaviors. To achieve this, we leverage syntactic characteristics and LSP server responses to identify locations that can lead to distinct and deeper LSP server behaviors. With these strategies, LSPFUZZ can effectively explore the combinatorial space of source code and editor operations to uncover bugs in LSP servers.

In our evaluation, we ran LSPFUZZ on four widely used LSP servers, such as `clangd` in LLVM and `sorbet` developed by Stripe. The experimental results demonstrate that LSPFUZZ can effectively detect bugs in LSP servers. On average, LSPFUZZ detected 54.1 crashes at different crash locations in these LSP servers across 10 repetitions. Moreover, LSPFUZZ achieves higher code coverage compared to baseline fuzzing tools, with improvements ranging from 2.45x to 142.9x over baseline approaches. Our analysis shows that the two-stage mutation pipeline is essential for effectiveness, especially for LSP servers with rich editor operation support. We reported 51 bugs to the respective development teams and received positive feedback. At the time of writing, 42 have been confirmed, 26 have been fixed, and two have been assigned CVE numbers. In addition, LLVM developers proactively responded to our security advisories by disabling `clangd` in their VSCode extensions for untrusted workspaces (Section V-F). A developer also expressed interest in integrating LSPFUZZ into their testing workflows.

In summary, this paper makes the following contributions:

- To the best of our knowledge, we are the first to systematically explore the quality assurance of LSP servers.
- We designed and implemented LSPFUZZ, a novel hybrid fuzzer for LSP servers, with 12,293 lines of Rust code.
- We evaluated LSPFUZZ on four widely used LSP servers, showing that it significantly outperforms baselines, and that the two-stage mutation pipeline is essential.
- We reported 51 previously unknown bugs to the vendors of LSP servers, with 42 confirmed, 26 fixed, and two CVEs granted. We received positive feedback from LSP server developers.
- We made LSPFUZZ and our experimental data public [11] to facilitate future research.

II. BACKGROUND

The Language Server Protocol (LSP) [1], introduced by Microsoft in 2016, provides a standardized interface between code editors and language analysis tools. LSP decouples code editors from language-specific analysis logic, allowing any editor to support languages with LSP servers and any language with an LSP server to provide code intelligence features across all LSP-compatible editors. This architecture has made LSP the de facto standard for code intelligence features [2]. Around 50 code editors (e.g., VSCODE, NEOVIM, ZED) include LSP clients, and there are over 300 LSP servers for various languages, such as `clangd` for C/C++ and `sorbet` for Ruby. This widespread adoption establishes LSP as the standard protocol for editor-language integration.

LSP servers work with code editors in a dynamic, interactive manner, differing from tools like compilers that process static code in a single pass. Typically, LSP servers are launched by the code editor as background processes to analyze the opened files. The content of the file is sent to the LSP server with a `textDocument/didOpen` message. When a developer interacts with the code editor, the editor translates these actions into LSP requests and sends them to the LSP server to request analysis or information. For instance, as illustrated in Fig. 1, when a developer places the mouse cursor on a token, the editor issues a `textDocument/hover` request to the LSP server, including the URI of the source file and the cursor position as parameters. The LSP server then analyzes the code at the specified location and returns information (e.g., type, signature, documentation) about the symbol, which is displayed to the user in a tooltip.

III. PROBLEM FORMULATION AND CHALLENGES

Testing LSP servers presents unique challenges that traditional software testing approaches cannot adequately address. Unlike most code analysis tools, which typically focus on generating static source code as input, LSP server testing must account for the dynamic and interactive nature of editor-server communication. An input to an LSP server consists of two tightly coupled components:

- **Source Code:** A source code file as displayed and edited in the code editor, written in the language supported by the target LSP server. It is also known as a text document in the protocol [1]. The full content of the source code is sent to the LSP server when the editor opens the file.
- **Editor Operations:** A sequence of user-driven actions performed on the text document, such as auto-completion, code navigation, mouse hovering, and formatting. These operations are encoded as LSP requests or notifications [1] and transmitted to the LSP server in real time.

The input space is not limited to source code; it also includes sequences of editor operations that are tightly coupled with the source code. This coupling between input components introduces unique challenges in LSP testing that require specialized approaches for effective bug discovery.

A. Challenge 1: Combinatorial Input Constraints

To reveal bugs in LSP servers, test cases must first pass complex input validation layers to reach the core functionality code. Without satisfying these validations, potential bugs in the server's internal logic remain unexposed [12]. Valid LSP server inputs must satisfy a set of sophisticated and interdependent constraints as specified in the protocol. Moreover, *these constraints exhibit complex combinatorial properties*. They apply not only to each source code or editor operation independently, but also to specific combinations of the two components interacting together. For example, the parameters of an auto-completion operation, a hover operation (as in Fig. 1), or a definition operation must refer to a valid position within the source code.

Without systematic awareness of these constraints, traditional fuzzing approaches such as binary [5] or grammar-based [6]–[9] fuzzers are unlikely to generate valid editor operations that can satisfy the combinatorial constraints and thus reach the core LSP server functionality. For instance, while a grammar-based fuzzer can easily generate an input with all the required fields to bypass the request parsing component, the request may point to a meaningless position and thus be rejected when the LSP server tries to process it.

B. Challenge 2: Two-Dimensional Diversity Requirements

While addressing protocol constraints enables valid test cases, effective bug discovery further requires exploring diverse combinations of inputs. Although diversity is a common requirement for software testing techniques [7], [13], [14], LSP server testing presents a unique two-dimensional diversity requirement for test cases. Its effectiveness depends on the interplay between two distinct dimensions of diversity:

- **Source Code Diversity:** The structural variety within the source code, including both well-formed and incomplete or invalid code fragments.
- **Editor Operation Diversity:** The range of editor operations performed at various code constructs within the source code file.

A key aspect of source code diversity is the frequent presence of incomplete or invalid code during interactive editing. For example, when a developer writes an incomplete variable declaration, the LSP server is expected to offer it as a completion candidate, demonstrating its ability to perform *partial analysis* on ill-formed code. Such scenarios can trigger unique behaviors and error-handling routines, further expanding the exploration space for LSP server testing.

Editor operation diversity is another important dimension, which is linked to the dynamic and interactive nature of LSP servers. An editor operation is *location-sensitive*: it can target a specific code symbol, a selection region, the entire code file, or the whole project. This *referential* relationship between source code and editor operations results in a vast space of possibilities: the triggered behavior depends not only on the operation itself but also on the particular code construct(s) it references. The same operation can exercise entirely different logic paths depending on where it is applied. For instance, a hover operation at a function call typically prompts the LSP server to extract its documentation, whereas the same request at a variable declaration leads the LSP server to infer its type. The bug illustrated in Fig. 1 exemplifies this interplay: it could only be triggered by the specific combination of source code containing a variable initialization with an incomplete function call and a hover request at the equal sign. Neither component alone would have exposed this bug.

Therefore, the exploration space formed by these two dimensions is not merely additive but multiplicative, which spans all the possible combinations of source code, selection targets, and editor operation sequences. Effective LSP server testing must systematically explore combinations of source code and editor operations, including those involving incomplete or invalid code, to efficiently exercise diverse code paths and error-handling routines. This combinatorial challenge necessitates fuzzing approaches specifically designed to navigate the unique input space of LSP servers.

IV. Approach: LspFuzz

The two challenges in LSP testing necessitate a specialized testing technique that systematically addresses both the combinatorial input constraints and the two-dimensional diversity requirements. In this section, we present our approach, LspFuzz, a grey-box mutational fuzzer designed to detect crashes and memory corruptions in LSP server implementations.

Fig. 2 shows an overview of LspFuzz. The fuzzing process begins by initializing the corpus with a set of randomly generated source code using Tree-Sitter [10] grammar rules of the target language. For each fuzzing iteration, the scheduler selects a seed from the corpus, which then undergoes a two-stage mutation pipeline. The mutated test case is run on the target LSP server, and its runtime behavior is monitored. If the test case triggers a previously unexplored control-flow edge, it is added to the corpus for further mutation. When execution results in a crash with a unique stack trace, the test case is recorded as a potential bug. This process continues iteratively until the allocated time budget is exhausted.

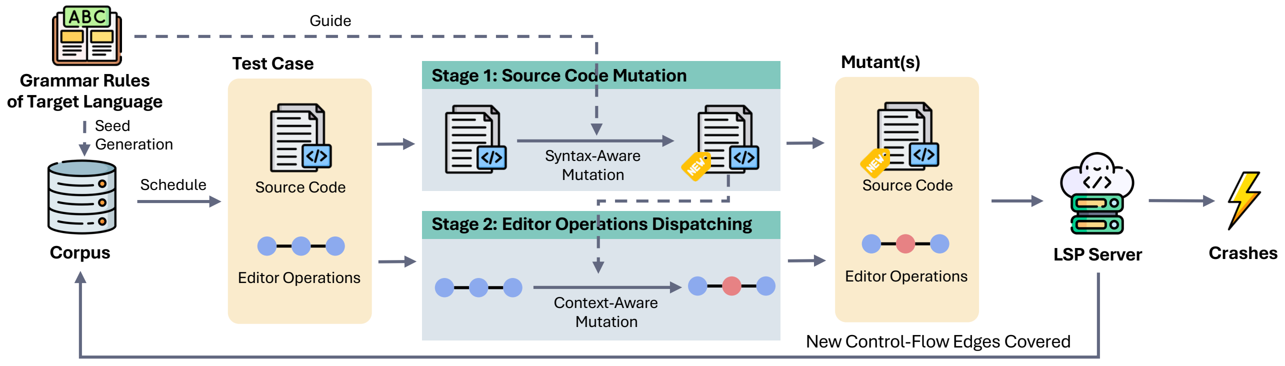


Fig. 2. Overview of LSPFUZZ

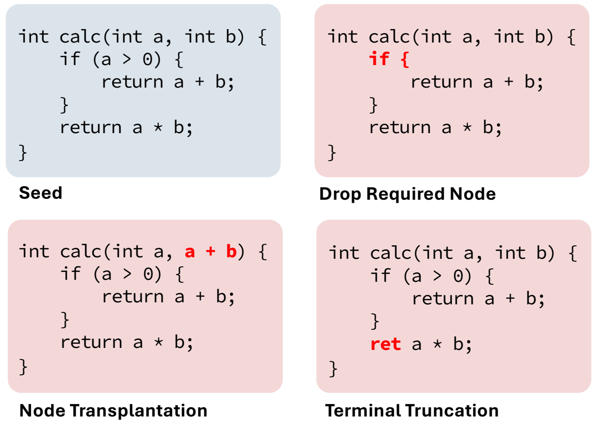
At the core of LSPFUZZ is a two-stage mutation pipeline, which enables the generation of test cases with two interconnected parts (i.e., source code and editor operations). This two-stage pipeline addresses the sophisticated constraints of LSP. The first stage performs syntax-aware mutation to produce source code with diverse syntactic combinations, triggering the code analysis logic for various types of code. The second stage generates editor operations targeting the mutated source code produced by the first stage, applying various strategies to dispatch editor operations that are likely to trigger deeper code analysis behaviors in LSP servers. Together, these strategies exercise various functionalities of LSP servers by producing diverse inputs. The following subsections detail this pipeline.

A. Stage I: Source Code Mutation

The first stage in the mutation pipeline aims to produce source code with diverse syntactic features. Such code serves as a foundation for exploring LSP server behaviors. We combine formal grammar-based generation with real-world code patterns to achieve both structural and semantic diversity.

1) Grammar-Based Mutation: For source code mutation, we leverage the random tree mutation strategy that has been proven effective in many grammar-based fuzzers [6], [7], [9]. Specifically, we randomly select a non-terminal node in the derivation tree of the source code and replace it with a new sub-tree rooted at the current non-terminal node type. To generate the replacement node, we randomly pick a production rule for the current non-terminal and expand it recursively. This ensures that the generated code adheres to the grammar rules of the target language and allows us to explore a wide range of possible syntactic combinations in that language.

While leveraging grammar rules ensures syntactic correctness, the generated code often lacks the semantic richness and complexity of real-world code [15]. To enrich the generated code and trigger various code analysis logic in the LSP server, we incorporate real-world code as ingredients for source code mutation, adopting an idea similar to LANGFUZZ [16]. Specifically, when generating the replacement node, we randomly choose a sub-tree from a code fragment pool (Section IV-C) with a compatible node type.


```
int calc(int a, int b) {
    if (a > 0) {
        return a + b;
    }
    return a * b;
}
```
Seed

```
int calc(int a, int b) {
    if {
        return a + b;
    }
    return a * b;
}
```
Drop Required Node

```
int calc(int a, a + b) {
    if (a > 0) {
        return a + b;
    }
    return a * b;
}
```
Node Transplantation

```
int calc(int a, int b) {
    if (a > 0) {
        return a + b;
    }
    ret a * b;
}
```
Terminal Truncation

Fig. 3. Examples of Mutation Operations Leading to Invalid Code

These two strategies enable us to produce diverse uses of the source code in the target language, which serves as a strong basis for exploring LSP server behaviors.

2) Invalid Code Injection: LSP servers must frequently process invalid code, as developers often produce incomplete or incorrect code during editing. Supporting partial analysis on such code is essential for LSP servers and exposes unique behaviors relevant for testing. For example, an LSP server should recognize an incomplete variable declaration and offer it as a completion candidate. Thus, generating invalid code is necessary to explore the input space and ensure the robustness of LSP servers.

To enable LSPFUZZ to explore this region in the input space, we produce invalid code at a controlled frequency. The generation of invalid code must be carefully controlled to achieve two competing objectives: 1) The code should differ from normal valid code sufficiently to trigger the server's partial analysis mechanisms, and 2) It should not be so malformed that it is immediately rejected without further processing. To balance these objectives, we designed three mutation operators (illustrated in Fig. 3) that produce code with a generally well-formed structure but with localized errors or misalignments.

In addition, they emulate common scenarios where developers produce intermediate code during programming.

- **Drop Required Nodes**: Removes required child nodes from a non-terminal in the parse tree. For example, omitting the condition of an `if` statement creates syntactically incomplete yet structurally meaningful code. It simulates scenarios where developers are creating a code construct (e.g., classes, functions, etc.). The resulting code can trigger the partial analysis mechanism, whereby LSP servers recognize and process incomplete code constructs.
- **Node Transplantation**: Replaces a node with another of a different type, such as substituting a parameter declaration with an expression node. This simulates scenarios where developers are refactoring or reorganizing code. This operator helps evaluate whether LSP servers can robustly process code with localized structural inconsistencies that may arise during such changes.
- **Terminal Truncation**: Truncates terminal tokens, such as cutting `return` to `ret` or leaving a string literal without a closing quote. This simulates scenarios where developers are typing or deleting a code symbol, testing LSP servers' ability to process partial symbols.

The generation of invalid code complements valid code produced by random tree mutation and enables LSPFUZZ to explore both the well-formed and ill-formed regions of the input space, mirroring the spectrum of source code states encountered during real-world software development.

Our approach to source code mutation produces diverse test inputs by generating structurally varied code through grammar-based techniques, incorporating real-world patterns and semantics, including both valid and invalid source code. The resulting rich testing space, when combined with our editor operation dispatching (Section IV-B), enables us to effectively explore the input space of LSP servers.

B. Stage II: Editor Operations Dispatching

Given the source code produced in the first stage as context, the second stage of the mutation pipeline dispatches editor operations targeting various constructs in the source code. Specifically, we randomly select an operation from all the available LSP operations, generate its parameters, and insert it into the editor operation sequence. Among all the parameters, the code construct targeted by the operation is the predominant factor for triggering behaviors in the LSP server. Therefore, the key to editor operation dispatching is to find the target construct for the editor operation. We consider two strategies that focus on the syntactic features of the source code and the responses produced by LSP servers.

1) Syntactic-Category-Level Randomization: In LSP, the code context of an editor operation is specified by line and column, referring to a specific position or range in the source code. However, trivial randomization may not effectively yield editor operations that trigger diverse LSP server behaviors. For example, character-level randomization may frequently select positions within the same token (especially for longer ones) or with the same node types (especially for common node types), which typically trigger identical or similar LSP server behaviors. Instead, we should perform random selection in a space where each element potentially links to distinct LSP server behaviors.

To achieve this goal, we analyze the behaviors of LSP servers given various code and operations to understand how they process these editor operation requests. We observed that the code construct targeted by an editor operation can affect the behavior of LSP servers when processing it. Specifically, different syntactic categories of code constructs (e.g., function, variable) can lead to significantly different behaviors. For example, a `textDocument/hover` at a function can lead to documentation extraction, while the same operation at a variable can lead to type inference. Thus, we should project the locations in the source code to a space that reflects the syntactic category of their corresponding code constructs.

Inspired by this observation, we introduce the *syntactic signature* for each node in the parse tree. Specifically, a syntactic signature of level n refers to a sequence $\langle t_0, t_1, \ldots, t_n \rangle$ where t_0 is the type of the node, and t_n is the type of its n^{th} parent node. A syntactic signature captures not only the syntactic characteristic of the node itself, but also its contextual information. This is crucial because nodes with the same type can have different semantics depending on where they appear in the syntax tree. For example, in C, a function name and a variable name both have the type `identifier`, but their level-1 syntactic signatures differ: ⟨`identifier`, `function_declarator`⟩ vs. ⟨`identifier`, `init_declarator`⟩. They can lead to different LSP server behaviors when targeted by editor operations (e.g., hover) as mentioned earlier. Syntactic signature works at the grammar level and does not rely on language-specific knowledge, which allows us to categorize code constructs in an arbitrary language.

When identifying target code constructs to dispatch editor operations, we perform random selection at the syntactic-category level. Specifically, we group the locations in the source code based on the syntactic signature of their corresponding nodes. Each group contains the locations with parse tree nodes in a specific syntactic category. Then we dispatch editor operations by randomly choosing code constructs from each group. Such syntactic-category-level randomization enables us to distribute exploration among the potential trigger behaviors of LSP servers and reduces the chance of generating redundant test cases.

2) Editor Operations Enrichment: Certain symbols in the source code lead to deeper code analysis, such as the semantic-level symbols recognized by LSP servers and code with diagnostic information. However, this kind of information cannot be derived from syntax-level analysis alone. To better trigger diverse behaviors linked to such symbols, we design two strategies to identify these symbols and prioritize dispatching editor operations that target them.

Semantic Symbols. Semantic symbols are a set of predefined code constructs (e.g., functions, types, and fields) recognized by the target LSP servers. These are regarded as well-formed code constructs because they pass basic validation and are present in the symbol table of the LSP server. Dispatching editor operations on them is more likely to trigger deeper code analysis within the LSP server. We identify them by capturing the responses generated by the LSP server for `textDocument/documentSymbol` and `workspace/symbol`, which contain the locations of these semantic symbols in the source code. During mutation, we prioritize dispatching editor operations to these locations.

Diagnostic Information. Code constructs containing localized errors, whether syntactic or semantic, can trigger partial analysis logic in LSP servers (Section III-B). Apart from syntax errors we inject during source code mutation, semantic errors are an important counterpart that can lead to partial analysis, and we leverage the diagnostic information generated by LSP servers to identify them. Specifically, the LSP server can emit detected errors in the source code by sending `textDocument/publishDiagnostics` requests to the code editor. We capture these requests and prioritize dispatching editor operations targeting the nodes that contain a child with a diagnostic. These nodes contain localized errors and are likely to trigger more partial analysis logic in the LSP servers.

Inter-Operation Dependencies. In LSP, several editor operations depend on the outcome of other operations. For example, a `callHierarchy/outgoingCalls` operation requires a `CallHierarchyItem` in its parameter, which is generated by the `textDocument/prepareCallHierarchy` operation. For such editor operations, we save LSP server responses for each seed in the corpus and use the associated response data to construct the parameters of these operations during mutation.

These two strategies work together to enrich our editor operations, directing fuzzing efforts toward code constructs that are more likely to exercise the deeper analysis logic of LSP servers and uncover potential bugs in their implementation.

C. Implementation

Overall Framework. To facilitate future extension and enable seamless interoperability with different fuzzing strategies, we implemented LspFuzz based on LibAFL [17] with 12,293 lines of Rust code, leveraging its modular architecture. We implemented custom mutators for our two-stage mutation pipeline while utilizing LibAFL's built-in coverage tracking and scheduling modules.

Source Code Processing. For source code parsing, mutation, and code context identification, we integrated Tree-Sitter [10], a parsing system that supports over 100 programming languages. This integration provides LspFuzz with the capability to test LSP servers across diverse programming languages without requiring language-specific tooling.

Code Fragment Pool. Before the fuzzing campaign, we construct the code fragment pool (Section IV-A1) by parsing example files and test cases in the code repository of the target LSP server, which contains code in the target language of the LSP server. Users of LspFuzz may optionally enrich it with additional source files (e.g., mined from GitHub). The probability of selecting a real-world code fragment during mutation is controlled by a configurable hyperparameter, which is set to 0.2 by default.

TABLE I
Subject LSP Servers for Evaluation

Name	Vendor	Language	Version	Popularity*
`clangd`	LLVM	C/C++	v20.1.4	1.8M
`sorbet`	Stripe Inc.	Ruby	v0.5.11031	901k
`verible`	CHIPS Alliance	Verilog	v0.0.3157	1.1M
`solc`	Ethereum	Solidity	v0.8.29	1.6M

*Number of installs of the associated VSCode extension at the time of writing

Fuzz Targets. LspFuzz is compatible with the fuzz targets for AFL++ [5] and libFuzzer [18]. To prepare a fuzz target, users should modify the entry point (i.e., the `main` function) of the target LSP server to read inputs from an in-memory byte array instead of `stdin` (i.e., shared-memory fuzzing). The fuzz target should be instrumented for collecting coverage feedback. Optionally, users can remove code unrelated to LSP functionality (e.g., CLI option parsing, configuration file loading) to achieve higher fuzzing throughput [19].

V. Evaluation

We evaluate LspFuzz through four research questions to assess its effectiveness and usefulness. Specifically, we aim to answer the following research questions.

- **RQ1:** Is LspFuzz effective in detecting crashes in LSP servers?
- **RQ2:** Can LspFuzz outperform baselines in code coverage and crash detection?
- **RQ3:** Is the two-stage mutation pipeline essential for the effectiveness of LspFuzz?
- **RQ4:** Is LspFuzz useful in finding previously unknown bugs in real-world LSP servers?

These research questions assess the effectiveness and practical utility of LspFuzz. RQ1 focuses on whether LspFuzz is effective at detecting crashes in LSP servers, using standard fuzzing metrics such as the number of detected crashes and code coverage. RQ2 compares LspFuzz against baseline fuzzing approaches to determine if it can achieve higher code coverage and detect more crashes. RQ3 investigates the necessity and impact of the two-stage mutation pipeline by comparing LspFuzz to a variant without this feature. RQ4 evaluates the practical usefulness of LspFuzz by examining its ability to uncover previously unknown bugs in production LSP servers and the responses from development teams.

A. Subject LSP Servers

We evaluated LspFuzz by selecting LSP servers from the top 200 most installed VSCode programming language extensions using three criteria: 1) they are actively maintained (i.e., have had a release in the past three months), 2) they are open-source, allowing us to modify the source code to prepare fuzz targets, and 3) they are compatible with AFL++ [5] instrumentation. Table I lists the selected LSP servers, which are the top-4 most popular LSP servers from the candidates. They are designed for diverse languages and ecosystems:

`clangd` is part of LLVM, providing language services for C/C++; the Solidity compiler (`solc`) includes LSP support for Ethereum's smart contracts; `sorbet` is Stripe's LSP server for Ruby; and `verible` supports System Verilog development. These LSP servers are large-scale, production-grade, mature software. They share similar complexity with compilers. Their associated VSCode extensions have attracted thousands to millions of installs, accounting for 96% of the total installations among the candidates.

We prepared fuzz targets following the approach presented in Section IV-C. To achieve higher throughput, we removed the code for CLI option parsing and configuration file loading, as they are irrelevant to LSP. We enabled `AddressSanitizer` [20] during compilation to detect memory corruptions. All four LSP servers are written in C/C++, and we instrumented the fuzz targets with the LLVM LTO mode provided by AFL++ [5].

B. Baselines and Experiment Setup

To the best of our knowledge, no existing technique specifically targets LSP server testing. Therefore, we implemented three baseline approaches using LibAFL [17].

LspBin. To understand the advantage of LspFuzz over current practice, we created a baseline, LspBin, using the binary mutators from LibAFL [17], which share the same algorithm as AFL++ [5]. LspBin is inspired by the practice that `clangd` and `sorbet` use binary fuzzers in their testing workflows, as evidenced in their public repositories. As a result, LspBin represents the current state of practice in fuzzing LSP servers and provides a comparison point to demonstrate the advantage of LspFuzz over current practice.

LspGram. We argued in Section III that a grammar-based fuzzer with LSP schema would be ineffective without considering the complex semantic relationships between source code and LSP operations. To validate this argument, we designed a baseline, LspGram, that implements a pure grammar-based approach. It leverages the grammar mutators built into LibAFL [17] and operates with a grammar constructed from the LSP schema, which specifies the syntactic structure of each LSP message type. Comparing LspFuzz against LspGram helps empirically demonstrate the advantage of maintaining semantic consistency in LSP server testing. In addition, ten randomly sampled inputs generated by LspGram are used as initial seeds for LspBin.

Lsp2D. As discussed in Section III, LSP server inputs consist of two dimensions: source code and editor operations. To capture this structure, we adapted a two-dimensional baseline from the `MultipartInput` in LibAFL. The adaptation was done following the LSP specifications [1]. Specifically, this baseline generates code and editor operations using the built-in mutation operators in LibAFL. The generated code is first sent to the LSP server with a `textDocument/didOpen` message, followed by a sequence of editor operations. Lsp2D generates inputs that are guaranteed to contain both code and editor operations. Comparing LspFuzz with Lsp2D helps to illustrate the benefits of our systematic mutation strategy tailored for LSP server testing.

Compared with LspFuzz these baselines either ignore the structural and semantic relationships in LSP inputs or fail to model the interaction between source code and editor operations, resulting in test cases that lack meaningful correspondence to real LSP usage.

Although we considered LLM-based approaches, they do not support LSP servers and adapting them is non-trivial. For example, ChatAFL [21] fails to start when LSP is specified as the protocol. We also considered existing multi-dimensional fuzzing approaches. However, they either 1) do not intent to address the challenges in LSP server testing, or 2) cannot be adapted to LSP due to their reliance on target-specific properties. Thus, we did not include them as baselines. We discuss these approaches in Section VI.

All experiments were conducted on a dual AMD EPYC 7773X machine (128 cores, 1TB RAM, AlmaLinux 9.5). Each fuzzing process used one core for 24 hours, repeated ten times, totaling 200 CPU-days.

C. RQ1: Effectiveness of LspFuzz *in Crash Detection*

RQ1 evaluates the effectiveness of LspFuzz by the number of detected crashes, which is a typical evaluation metric for fuzzers [19]. We deduplicated the crashes with stack hashes following the practice of existing work [19], and report the results in Table II. In our subject LSP servers, LspFuzz detected a total of 991 crashes in 10 runs of 24 hours, with an average of 181.1 per run.

To better understand how these crashes are triggered, we categorized them into two types based on the stage at which they occur, reflecting the complexity required to trigger them.

- **Code-Loading Crashes:** These occur when the LSP server receives and analyzes source code from the editor. Loading source code alone can trigger these crashes, typically during code parsing or initial analysis, without requiring specific editor interactions.
- **Operation-Triggered Crashes:** These occur when the LSP server processes specific editor operations on certain source code. They require particular combinations of source code and editor operations, and typically manifest in components handling language features or editor functionalities.

The columns *Code* and *EOP* in Table II show the breakdown of the crashes into these two categories, respectively. The number of code-loading crashes is large, especially for `clangd`, which has a huge amount of code for various C/C++ analyses. This is as expected because code-loading crashes are easier to trigger compared with operation-triggered crashes, as they do not require specific combinations of source code and editor operations. Nevertheless, LspFuzz is effective at discovering operation-triggered crashes, with 44% of the detected crashes being operation-triggered. Such crashes require specific combinations of source code and editor operations.

The results highlight the significant crash detection capabilities of LspFuzz. In the following, we show two cases to illustrate the crashes found by LspFuzz. These two cases highlight different strengths of our approach.

TABLE II
REPRODUCIBLE CRASHES WITH UNIQUE STACK TRACES (RQ1 – RQ3)

Subject	`clangd`				`sorbet`				`verible`				`solc`				**All**			
Approach[1]	**Total**	**Code**	**EOP**	**Avg**	**Total**	**Code**	**EOP**	**Avg**	**Total**	**Code**	**EOP**	**Avg**	**Total**	**Code**	**EOP**	**Avg**	**Total**	**Code**	**EOP**	**Avg**
LspFuzz	499	483	16	70.9	64	27	37	28.5	399	20	379	74.6	29	25	4	7.1	991	555	436	181.1
LspBin	0	-	-	0.0	2	-	-	2.0	2	-	-	2.0	0	-	-	0.0	4	-	-	4.0
LspGram	5	5	0	2.3	2	2	0	2.0	5	5	0	2.2	0	0	0	0.0	12	12	0	6.5
Lsp2D	122	122	0	14.9	7	7	0	2.8	7	7	0	3.1	1	1	0	1.0	137	137	0	21.8
LspFuzz$_{NC}$	437	437	0	64.4	25	25	0	14.6	251	21	230	41.5	21	21	0	2.7	734	504	230	120.2

1. Total are all the crashes, Code are the code-loading crashes, EOP are operation-triggered crashes, Avg are the average number of crashes over the 10 repetitions.
2. Test cases from LspBin may not be valid LSP inputs and cannot always be classified as code-loading or operation-triggered crashes.

Case Study I: Crash in `clangd` on Formatting. This case demonstrates the effectiveness of our invalid code injection mechanism (Section IV-A2). LspFuzz found a crash in the C/C++ LSP server `clangd`, which is triggered by the combination of a code snippet with localized errors and a `textDocument/formatting` operation. The code snippet contains a C++ `decltype` specifier missing its closing parenthesis. This simulates a realistic scenario where a developer is writing a `decltype` specifier but has not yet finished. When processing the formatting operation, the formatter in `clangd` attempted to locate the end of the `decltype` sequence but encountered a `null` pointer, resulting in a segmentation fault. The malformed `decltype` specifier was produced by our mutation operators for injecting invalid code, which revealed this bug when combined with a formatting editor operation. Without invalid code injection, LspFuzz could not produce such a test case. The bug affects users in common programming workflows, as code formatting is a frequently used feature during development. LLVM developers confirmed and fixed this bug, which had remained hidden in `clangd` for two years.

Case Study II: Crash in `sorbet` on Go to Implementation. This case demonstrates the effectiveness of our editor operation dispatching strategies (Section IV-B). LspFuzz discovered a crash in the Ruby LSP server `sorbet` triggered by the combination of a malformed lambda expression `->...{}` and a `textDocument/implementation` operation specifically targeting the arrow symbol. This combination resulted in an assertion violation in `sorbet`, causing the LSP server to abort. This case exemplifies the power of our syntactic signature approach: although the arrow of a lambda expression spans only two characters in the source code, our system recognized its syntactic significance as an `arrow` within a `lambda` in the Ruby grammar rules, and balanced the chance to interact with it and other long units like a lengthy literal. Furthermore, the triple dot operator between the arrow and its body triggered diagnostics, causing it to be prioritized by our editor operation enrichment mechanism. This case highlights how our approach can identify interesting positions within the code for applying editor operations, even for small syntax elements. The `sorbet` development team confirmed and fixed this bug, which had remained undetected in the project for four years.

Last but not least, while stack-trace-based crash deduplication is a common strategy [19], for LSP servers, crashes with different stack traces may not always correspond to distinct bugs. This is because the same underlying fault can be triggered in multiple contexts (e.g., a bug in the type inference module can be triggered by both hover and go-to-type-definition), resulting in different stack traces depending on the execution path. Therefore, we further deduplicated the crashes in Table II by the program locations where they occur, and report the results in Table III.

TABLE III
REPRODUCIBLE CRASHES AT UNIQUE LOCATIONS (RQ1 – RQ4)

Subject	`clangd`		`sorbet`		`verible`		`solc`		**All**	
Approach[1]	**Total**	**Avg**	**Total**	**Avg**	**Total**	**Avg**	**Total**	**Avg**	**Total**	**Avg**
LspFuzz	60	14.3	32	19.8	19	16.1	4	4	115	54.2
LspBin	0	0.0	2	2.0	2	2.0	0	0	4	4.0
LspGram	5	2.3	2	2.0	4	1.8	0	0	11	6.1
Lsp2D	5	2.7	7	2.8	3	2.4	1	1.0	16	8.9
LspFuzz$_{NC}$	34	9.9	15	9.7	13	10.1	1	1	62	30.6

1. Total are all the crashes, Avg are the average number of crashes over the 10 repetitions.

Even with deduplication at this level, LspFuzz can still detect an average of 54.2 crashes across 10 runs, with 14.3 in `clangd`, 19.8 in `sorbet`, 16.1 in `verible`, and 4 in `solc`.

RQ1 Takeaway: LspFuzz is effective in detecting crashes in real-world LSP servers. On average, it detected 181.1 crashes with distinct stack traces and 54.2 crashes at different program locations across four LSP servers over 10 runs. Over 40% of the detected crashes manifest with specific combinations of source and editor operations, which are difficult to trigger.

D. RQ2: Comparison with Baselines

RQ2 aims to investigate whether LspFuzz can outperform baselines in two typical metrics for evaluating fuzzers: edge coverage (i.e., number of control-flow edges covered) and number of crashes detected [19].

Table IV shows the average number of control-flow edges covered by LspFuzz and the baselines, with a 95% confidence interval. Their changes over time are illustrated in Fig. 4. LspFuzz achieved significantly higher edge coverage than the baselines. For LspBin, which is the status quo in `clangd` and `sorbet`, the improvement ranges from 3.2x to 142.9x. For LspGram and Lsp2D, even with the guidance of LSP specification and two dimensions, the improvement is still significant, ranging from 2.2x to 15.1x.

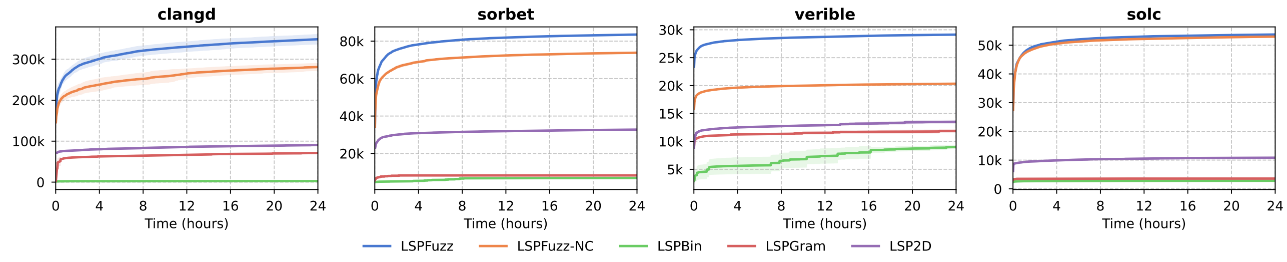


Fig. 4. Average Edge Coverage Over Time with 95% Confidence Interval (RQ2 and RQ3)

TABLE IV
AVERAGE EDGE COVERAGE ACHIEVED IN 24 HOURS (RQ2 AND RQ3)

Subject	Approach	Edge Coverage ± 95% CI	% LSPFUZZ
clangd	LSPFUZZ	348,741 ± 11,402	100%
	LSPBIN	2,503 ± 9	0.7%
	LSPGRAM	71,195 ± 1,915	20.4%
	LSP2D	90,450 ± 1,484	25.9%
	LSPFUZZ$_{NC}$	280,966 ± 8,581	80.6%
sorbet	LSPFUZZ	83,151 ± 127	100%
	LSPBIN	6,932 ± 63	8.3%
	LSPGRAM	8,302 ± 2	9.9%
	LSP2D	32,755 ± 149	39.2%
	LSPFUZZ$_{NC}$	73,812 ± 153	88.4%
verible	LSPFUZZ	29,133 ± 56	100%
	LSPBIN	8,990 ± 332	30.9%
	LSPGRAM	11,874 ± 290	40.8%
	LSP2D	13,504 ± 318	46.4%
	LSPFUZZ$_{NC}$	20,336 ± 31	69.8%
solc	LSPFUZZ	53,720 ± 72	100%
	LSPBIN	2,816 ± 14	5.2%
	LSPGRAM	3,572 ± 7	6.6%
	LSP2D	10,838 ± 154	20.2%
	LSPFUZZ$_{NC}$	52,987 ± 97	98.6%

The improvement in code coverage also enables LSPFUZZ to detect many more crashes compared with the baselines, as shown in Table II. On average, LSPBIN detected 4.0 crashes LSPGRAM detected 6.5 crashes, and LSP2D detected 21.8, which are 2.2%, 3.6%, and 12.0% of LSPFUZZ, respectively. LSPGRAM was only able to find code-loading crashes in all subjects and failed to detect any operation-triggered crashes. This confirms our hypothesis that pure grammar-based approaches struggle with the semantic constraints in LSP. In addition, even with two input dimensions, LSP2D can find only code-loading crashes. This demonstrates the advantage of our mutation strategy tailored for LSP servers. Also, as shown in Table III, LSPFUZZ detected more crashes at unique program locations than the baselines. The average improvement is 13.6x, 8.9x, and 6.1x for LSPBIN, LSPGRAM, and LSP2D, respectively. All four crashes detected by LSPBIN were in the JSON libraries used for message parsing, not in LSP server logic, and are out of scope of this paper.

The significant improvement can be attributed to our LSP-server-specific problem formulation and mutation strategies. In comparison, LSPGRAM lacks a two-dimensional formulation, and its generated test cases may not send the source code to LSP servers. Although LSP2D is built with two-dimensional inputs, it struggles to handle the referential relationship between the code and editor operations, which is brought by the location sensitivity of the editor operations. As a result, it often produces syntactically correct but semantically meaningless inputs. Take the input in Fig. 1 as an example, LSP2D can produce an editor operation that points to nowhere, which violates the location-sensitive constraints of LSP server inputs. With the two-stage semantic-aware mutation pipeline, LSPFUZZ can produce more meaningful inputs and more effectively explore the input space.

RQ2 Takeaway: LSPFUZZ outperforms our baselines by detecting more (6.1x to 13.6x) crashes, especially for those that are difficult to trigger. It also outperforms baselines by a significant margin (2.2x to 142.9x) in edge coverage.

E. RQ3: Effectiveness of the Two-Stage Mutation Pipeline

RQ3 investigates the effectiveness of the holistic mutation strategy in LSPFUZZ, which is the key insight of our approach. To this end, we construct LSPFUZZ$_{NC}$, a variant of LSPFUZZ that mutates the source code and editor operations independently, and run it with the same setup as LSPFUZZ.

As shown in Table IV and Fig. 4, LSPFUZZ$_{NC}$ achieves lower edge coverage than LSPFUZZ. In three of our four subjects, LSPFUZZ$_{NC}$ fails to cover many edges reached by LSPFUZZ. This is because the lack of context awareness makes our mutation strategies for editor operations ineffective, as these strategies rely on the source code as context. As a result, LSPFUZZ$_{NC}$ must blindly explore the large, two-dimensional input space. Such blind exploration often leads to test cases violating LSP constraints (e.g., a hover operation pointing at a non-existent location). Therefore, LSPFUZZ$_{NC}$ fails to cover as much code as LSPFUZZ within the same time budget.

The absence of holistic mutation also leads to a decrease in the number of detected crashes, as shown in Table II. These missed crashes are precisely those that are hard to trigger. As shown in Table II, without context awareness, LSPFUZZ$_{NC}$ fails to detect any operation-triggered crashes in three out of four subjects. This highlights the importance of context awareness in detecting operation-triggered crashes, which are more difficult to trigger compared with code-loading crashes.

An exception is `solc`; as shown in Fig. 4, the margin between LSPFUZZ and LSPFUZZ$_{NC}$ is small. The reason may be that `solc` provides a very thin layer of LSP features – it supports only four types of editor operations. Although most of

TABLE V
STATUS OF THE REPORTED BUGS (RQ4)

LSP Server	Reported	Confirmed	Fixed
clangd	13	13	6
sorbet	20	14	13
verible	14	12	4
solc	4	3	3
Total	51	42	26

the covered code is in the underlying code analyzer, LSPFUZZ can still detect four operation-triggered crashes, demonstrating the effectiveness of our approach even for LSP servers with limited feature sets. In comparison, for the other three LSP servers that provide rich features, our two-stage mutation pipeline can generate test cases to effectively exercise these features, and thus make a significant difference in coverage.

RQ3 Takeaway: The two-stage mutation pipeline in LSPFUZZ is effective. It helps achieve better code coverage and detect more crashes, especially the crashes requiring certain combinations of source code and editor operations.

F. RQ4: Usefulness in Finding Bugs

RQ4 aims to reveal the usefulness of LSPFUZZ in finding bugs in real-world LSP servers. To evaluate this, we report crashes at unique program locations found by LSPFUZZ (Table III) to the LSP server developers for their confirmation.

We followed ethical guidelines when submitting bug reports. For bugs with potential security implications, we contacted development teams privately according to their security policies. To avoid burdening developers [22], we attempted to minimize test cases by removing irrelevant code and editor operations while preserving the bug-triggering behavior. Due to the unique LSP test case format, minimization was performed manually, which was labor-intensive and time-consuming. Therefore, we reported only the crashes detected in the first repetition of our experiments. However, we have validated that all 115 crashes at unique locations are reproducible. They are also available in our artifacts [11] for readers to examine.

Table V shows the status of bugs we reported to LSP developers. In total, we reported 51 previously unknown bugs. At the time of writing, developers have confirmed 42 bugs, of which 26 have been fixed. Among these bug reports, two CVEs have been granted for vulnerabilities in `sorbet` and `verible`.

We received positive feedback from LSP server vendors on our bug reports. Several bugs we reported to `clangd` developers are buffer-overflow and use-after-free, which can potentially lead to remote code execution when editing malicious source files. In response to the security advisories we reported (private disclosure), the LLVM team disabled `clangd` in their VSCODE extension for untrusted workspaces [23].

Parsing untrusted code through clang can result in harmful behavior. Also it isn't considered as a security-sensitive component, hence its on embedders like `vscode-clangd` *to ensure users are aware of such risks.*

Additionally, during our communication with the `sorbet` development team, a developer showed interest in integrating LSPFUZZ into their testing workflow [24].

I'd love to hear more about what your process looks like for that. If there's something repeatable that we could use for our own purposes, I'd love to learn how much effort is involved.

Our results demonstrate that LSPFUZZ is not only effective at uncovering previously unknown bugs in production LSP servers, but also valuable for LSP server developers.

RQ4 Takeaway: LSPFUZZ can find bugs in real-world LSP servers confirmed by developers. We received positive feedback from LSP server developers on our bug reports. In addition, they expressed interest in integrating LSPFUZZ into their testing workflows.

G. Threats to Validity

First, we evaluated LSPFUZZ on four LSP servers, which may not represent the full diversity of LSP servers. To mitigate this threat, we selected widely used, large-scale, and actively maintained LSP servers. However, our results may not generalize to all LSP servers, especially those with different implementation strategies. Evaluating LSPFUZZ on more LSP servers remains an important direction for future work.

In addition, the results of our experiments may vary between runs due to the inherent randomness of fuzzing. To mitigate this threat, we repeated each experiment ten times to reduce the impact of randomness.

VI. RELATED WORK

As the first LSP server testing technique, our work on LSPFUZZ draws inspiration from several lines of related work, which we discuss in this section.

Grammar-Based Fuzzing. The first line of related work is grammar-based fuzzers [6], [7], [13], [25], [26], which generate structurally sound inputs using formal specifications and significantly improve testing effectiveness. However, they often fall short because real-world applications like LSP servers require inputs that satisfy complex constraints. To address this, ISLA [8] introduces a specification language for expressing semantic constraints in grammar-based test generation, and FANDANGO [9] further combines this with search-based techniques for more efficient constraint solving. Although these techniques can generate inputs with semantic constraints, the lack of a comprehensive LSP constraint set limits their applicability to LSP servers. In comparison, in LSPFUZZ, we encode the LSP constraints into the mutation pipeline and perform context-aware mutations, which always lead to test cases conforming to the protocol.

Network Protocol Fuzzing. The second line of related work focuses on network protocol fuzzing, as LSP is also a protocol despite operating locally. Several studies have advanced stateful fuzzing by leveraging protocol state identification and efficient state space exploration [27]–[31]. Other approaches focus on optimizing throughput and discovering memory issues in specialized domains (e.g., SNAPFUZZ [32], IOTFUZZER [33]). More recently, CHATAFL [21] leverages LLMs for mutation to escape the coverage plateau. While these approaches handle protocol state transitions and message sequencing well, they lack awareness of source code, which is essential for LSP servers. We address this by generating both source code and editor operations, thus covering both protocol and content aspects in our two-stage mutation pipeline.

Compilers and Code Analysis Tools Testing. The third line of related work concerns testing compilers and code analysis tools, which, like LSP servers, process and analyze source code. Tools such as CSMITH [13], LANGFUZZ [16], YARPGEN [34], and CUDASMITH [35] generate diverse code to test compilers, while NSSMITH [36] and HIRGEN [37] target deep learning compilers. Recent work, including GRAYC [38], COVRL-FUZZ [39], and FUZZ4ALL [40], applies grey-box fuzzing and LLMs to compiler testing. While these methods excel at code generation, they overlook the interactive aspect of LSP servers, where bugs may arise from specific combinations of code and editor actions. Nevertheless, incorporating advanced code generation strategies from these techniques could further enhance LSPFUZZ.

Multi-Dimensional Fuzzing. The fourth line of related work examines fuzzing techniques for applications that accept multi-dimensional inputs. FALCON [41] targets SMT solvers by leveraging the co-existence relation between formula types and configuration options to guide test generation. However, their approach is too coarse-grained to capture the location-sensitive relationship between code and editor operations in LSP (see Section III-B). KEXTFUZZ [42] focuses on macOS kernel extensions, employing novel instrumentation, in-kernel interactions, and input format inference to exercise diverse behaviors. DISTFUZZ [43] addresses distributed systems by introducing richer event types and pruning communication patterns based on symmetry. These approaches are either highly specialized for their respective domains or are not intended to address the challenges in LSP server testing (see Section III). Therefore, although LSP servers accept multi-dimensional inputs, these techniques cannot be transferred or adapted for LSP server testing.

LLM-Based Fuzzing. Large language models are increasingly being adopted in software testing research. In addition to the previously discussed work [21], [39], [40], many studies have explored leveraging LLMs for fuzzing. CKG-FUZZER [44] introduces a knowledge graph-driven approach that utilizes LLMs to automate fuzz driver generation and input seed refinement. Asmita et al. [45] propose two LLM-based techniques to improve fuzz testing of BUSYBOX. FUZZGPT [46] employs LLMs to generate edge-case programs for deep learning library testing. PROMPTFUZZ [47] uses LLMs for fuzz driver generation. These studies utilize the empirical knowledge encoded in LLMs to address various challenges in fuzzing. Given the widespread integration of LLMs in modern code editors, an intriguing direction for future work is to leverage LLMs to generate complex code editing actions, thereby further enhancing LSP server fuzzing.

VII. CONCLUSION AND FUTURE WORK

LSP has become the de facto standard for enabling code intelligence features in code editors. While the reliability of LSP servers is a growing concern, no existing techniques specifically target LSP server testing. In this paper, we propose LSPFUZZ to bridge this gap. LSPFUZZ outperforms baselines and is effective at detecting bugs and vulnerabilities in real-world LSP servers. Our bug reports led to prompt actions by LSP server developers to address the issues and received positive feedback.

LSPFUZZ is not only the first technique of its kind, but also serves as a foundation for further enhancements. We outline several limitations that present promising future work.

Multi- or Evolving Source File Scenarios. Our current problem formulation does not target scenarios where LSP servers operate on multiple or evolving source files. Therefore, LSPFUZZ cannot detect bugs that are triggered in these scenarios. Investigating the testing of LSP servers in multi-file workspaces or with dynamically changing source files could uncover bugs related to cross-file dependencies and incremental analysis. This is challenging due to the wide variety of possible file states and connections, which requires specifically tailored mutation operators, making it an interesting area for future research.

LSP-Specific Test Oracles. LSPFUZZ focuses on bugs that lead to crashes or memory corruptions. However, many functional bugs could be exposed by designing more sophisticated test oracles (e.g., go-to-definition requests from multiple references to the same symbol should return the same location). Developing and exploring richer test oracles for LSP servers represents an important direction for future work.

Semantic-Aware Source Code Mutation. Our current approach uses only syntactic mutations, which may not generate sufficiently complex and valid programs as part of the test inputs. Thus, LSPFUZZ may miss bugs caused by semantic characteristics (e.g., def-use or caller-callee relations). Incorporating semantic-aware mutations, inspired by strategies from compiler testing techniques, could help uncover deeper bugs in LSP servers.

VIII. DATA AVAILABILITY

Our artifact is available at the following URL:

https://doi.org/10.5281/zenodo.17052142

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. This work is supported by the Hong Kong Research Grants Council General Research Fund (Grant No. 16206524, for authors at HKUST), Natural Sciences and Engineering Research Council of Canada Discovery Grant (Grant No. RGCPIN-2022-03744 and DGECR-2022-00378, for authors at McGill), Fonds de recherche du Québec-secteur Nature et technologies (Grant No. 363482 [48], for authors at McGill), and the National Natural Science Foundation of China (Grant No. 62372219, for authors at SUSTech).

References

[1] M. Corporation. (2016) Language server protocol. [Online]. Available: https://microsoft.github.io/language-server-protocol/

[2] N. Gunasinghe and N. Marcus, *Language Server Protocol and Implementation*, 1st ed. Berkeley, CA: Apress, 2022.

[3] Visual studio code marketplace. [Online]. Available: https://marketplace.visualstudio.com/vscode

[4] LLVM. Issue 138096, llvm-project. [Online]. Available: https://github.com/llvm/llvm-project/issues/138096

[5] A. Fioraldi, D. C. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Y. Yarom and S. Zennou, Eds. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[6] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A. Sadeghi, and D. Teuchert, "NAUTILUS: fishing for deep bugs with grammars," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[7] P. Srivastava and M. Payer, "Gramatron: effective grammar-aware fuzzing," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 244–256. [Online]. Available: https://doi.org/10.1145/3460319.3464814

[8] D. Steinhöfel and A. Zeller, "Input invariants," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 583–594. [Online]. Available: https://doi.org/10.1145/3540250.3549139

[9] J. A. Z. Amaya, M. Smytzek, and A. Zeller, "FANDANGO: evolving language-based testing," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, pp. 894–916, 2025. [Online]. Available: https://doi.org/10.1145/3728915

[10] M. Brunsfeld, A. Qureshi, A. Hlynskyi, P. Thomson, ObserverOfTime, W. Lillis, J. Vera, dundargoc, P. Turnbull, T. Clem, D. Creager, A. Helwer, R. Rix, D. Kavolis, H. van Antwerpen, M. Davis, C. Clason, Ika, A. Ya, R. Bruins, T.-A. Nguygn, S. Brunk, M. Massicotte, bfredl, N. Hasabnis, M. Dong, S. Moelius, S. Kalt, and Kolja. (2025, Mar.) tree-sitter/tree-sitter: v0.25.3. [Online]. Available: https://doi.org/10.5281/zenodo.14969376

[11] H. Zhu, S. Chen, V. Terragni, L. Wei, Y. Liu, J. Wu, and S.-C. Cheung, "Lspfuzz: Hunting bugs in language servers (experimental data)." [Online]. Available: https://doi.org/10.5281/zenodo.17052142

[12] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 579–594. [Online]. Available: https://doi.org/10.1109/SP.2017.23

[13] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds. ACM, 2011, pp. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[14] R. Dutra, R. Gopinath, and A. Zeller, "Formatfuzzer: Effective fuzzing of binary file formats," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, pp. 53:1–53:29, 2024. [Online]. Available: https://doi.org/10.1145/3628157

[15] S. Li, T. Theodoridis, and Z. Su, "Boosting compiler testing by injecting real-world code," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, pp. 223–245, 2024. [Online]. Available: https://doi.org/10.1145/3656386

[16] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, T. Kohno, Ed. USENIX Association, 2012, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[17] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "Libafl: A framework to build modular and reusable fuzzers," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 1051–1065. [Online]. Available: https://doi.org/10.1145/3548606.3560602

[18] LLVM. libfuzzer – a library for coverage-guided fuzz testing. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[19] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[20] Address sanitizer. [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizer

[21] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/large-language-model-guided-protocol-fuzzing/

[22] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: syntax-guided program reduction," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 361–371. [Online]. Available: https://doi.org/10.1145/3180155.3180236

[23] LLVM. Issue 812, vscode-clangd. [Online]. Available: https://github.com/clangd/vscode-clangd/pull/812

[24] S. Inc. Issue 8903, sorbet. [Online]. Available: https://github.com/sorbet/sorbet/issues/8903#issuecomment-2902958689

[25] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, R. Gupta and S. P. Amarasinghe, Eds. ACM, 2008, pp. 206–215. [Online]. Available: https://doi.org/10.1145/1375581.1375607

[26] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "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*, ser. Lecture Notes in Computer Science, I. G. Askoxylakis, S. Ioannidis, S. K. Katsikas, and C. Meadows, Eds., vol. 9878. Springer, 2016, pp. 581–601. [Online]. Available: https://doi.org/10.1007/978-3-319-45744-4_29

[27] V. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox fuzzer for network protocols," in *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 460–465. [Online]. Available: https://doi.org/10.1109/ICST46399.2020.00062

[28] R. Meng, V. Pham, M. Böhme, and A. Roychoudhury, "Aflnet five years later: On coverage-guided protocol fuzzing," *IEEE Trans. Software Eng.*, vol. 51, no. 4, pp. 960–974, 2025. [Online]. Available: https://doi.org/10.1109/TSE.2025.3535925

[29] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *Empir. Softw. Eng.*, vol. 27, no. 7, p. 191, 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10233-3

[30] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 3255–3272. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/ba

[31] S. Qin, F. Hu, Z. Ma, B. Zhao, T. Yin, and C. Zhang, "Nsfuzz: Towards efficient and state-aware network service fuzzing," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 6, pp. 160:1–160:26, 2023. [Online]. Available: https://doi.org/10.1145/3580598

[32] A. Andronidis and C. Cadar, "Snapfuzz: high-throughput fuzzing of network applications," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 340–351. [Online]. Available: https://doi.org/10.1145/3533767.3534376

[33] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet

Society, 2018. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-1_Chen_paper.pdf

[34] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for C and C++ compilers with yarpgen," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 196:1–196:25, 2020. [Online]. Available: https://doi.org/10.1145/3428264

[35] B. Jiang, X. Wang, W. K. Chan, T. H. Tse, N. Li, Y. Yin, and Z. Zhang, "Cudasmith: A fuzzer for CUDA compilers," in *44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, July 13-17, 2020*. IEEE, 2020, pp. 861–871. [Online]. Available: https://doi.org/10.1109/COMPSAC48688.2020.0-156

[36] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds. ACM, 2023, pp. 530–543. [Online]. Available: https://doi.org/10.1145/3575693.3575707

[37] H. Ma, Q. Shen, Y. Tian, J. Chen, and S. Cheung, "Fuzzing deep learning compilers with hirgen," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 248–260. [Online]. Available: https://doi.org/10.1145/3597926.3598053

[38] K. Even-Mendoza, A. Sharma, A. F. Donaldson, and C. Cadar, "Grayc: Greybox fuzzing of compilers and analysers for C," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 1219–1231. [Online]. Available: https://doi.org/10.1145/3597926.3598130

[39] J. Eom, S. Jeong, and T. Kwon, "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 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 1656–1668. [Online]. Available: https://doi.org/10.1145/3650212.3680389

[40] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 126:1–126:13. [Online]. Available: https://doi.org/10.1145/3597503.3639121

[41] P. Yao, H. Huang, W. Tang, Q. Shi, R. Wu, and C. Zhang, "Fuzzing SMT solvers via two-dimensional input space exploration," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 322–335. [Online]. Available: https://doi.org/10.1145/3460319.3464803

[42] T. Yin, Z. Gao, Z. Xiao, Z. Ma, M. Zheng, and C. Zhang, "Kextfuzz: A practical fuzzer for macos kernel extensions on apple silicon," *IEEE Trans. Dependable Secur. Comput.*, vol. 21, no. 4, pp. 3453–3468, 2024. [Online]. Available: https://doi.org/10.1109/TDSC.2023.3330852

[43] Y. Zou, J. Bai, Z. Jiang, M. Zhao, and D. Zhou, "Blackbox fuzzing of distributed systems with multi-dimensional inputs and symmetry-based feedback pruning," in *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[44] H. Xu, W. Ma, T. Zhou, Y. Zhao, K. Chen, Q. Hu, Y. Liu, and H. Wang, "Ckgfuzzer: Llm-based fuzz driver generation enhanced by code knowledge graph," in *ICSE 2025*.

[45] Asmita, Y. Oliinyk, M. Scott, R. Tsang, C. Fang, and H. Homayoun, "Fuzzing busybox: Leveraging LLM and crash reuse for embedded bug unearthing," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/asmita

[46] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 70:1–70:13. [Online]. Available: https://doi.org/10.1145/3597503.3623343

[47] Y. Lyu, Y. Xie, P. Chen, and H. Chen, "Prompt fuzzing for fuzz driver generation," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, B. Luo, X. Liao, J. Xu, E. Kirda, and D. Lie, Eds. ACM, 2024, pp. 3793–3807. [Online]. Available: https://doi.org/10.1145/3658644.3670396

[48] "Combiner l'analyse statique et la génération de tests pour une assurance qualité améliorée : Une exploration des problèmes de compatibilité android," 2025.