IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 23 September 2024, accepted 16 October 2024, date of publication 23 October 2024, date of current version 1 November 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3484947

RESEARCH ARTICLE

Evaluating Large Language Models for Enhanced Fuzzing: An Analysis Framework for LLM-Driven Seed Generation

GAVIN BLACK[1], **(Member, IEEE), VARGHESE MATHEW VAIDYAN**[1], **AND GURCAN COMERT**[2]
[1]Department of Computer and Cyber Sciences, Dakota State University, Madison, SD 57042, USA
[2]Benedict College, Columbia, SC 29204, USA

Corresponding author: Gavin Black (Gavin.Black@trojans.dsu.edu)

This work was supported in part by the National Center for Transportation Cybersecurity and Resiliency (TraCR), USA, headquartered at Clemson University, Clemson, SC, USA; in part by the Cyber-Ag-Law Research Collaborations with the Department of Energy Minority Serving Institutions Partnership Program (MSIPP) managed by the Savannah River National Laboratory under Battelle Savannah River Alliance (BSRA) under Contract TOA 0000525174 CN1; in part by the Minority Science and Engineering Improvement Program (MSEIP) II Cyber, USA, under Grant P120A190061, Grant P120A210048, and Grant FM-MHP-0678-22-01-00; in part by the National Science Foundation (NSF), USA, under Grant 1954532, Grant 2131080, Grant 2200457, Grant OIA-2242812, Grant 2234920, and Grant 2305470.

ABSTRACT Fuzzing is a crucial technique for detecting software defects by dynamically generating and testing program inputs. This study introduces a framework designed to assess the application of Large Language Models (LLMs) to automate the generation of effective seed inputs for fuzzing, particularly in the Python programming environment where traditional approaches are less effective. Utilizing the Atheris fuzzing framework, we created over 38,000 seed inputs from LLMs targeted at 50 Python functions from widely-used libraries. Our findings underscore the critical role of LLM selection in seed effectiveness. In certain cases, seeds generated by LLMs rivaled or surpassed traditional fuzzing campaigns, with a corpus of fewer than 100 LLM-generated entries outperforming over 100,000 conventionally produced inputs. These seeds significantly improved code coverage and instruction count during fuzzing sessions, illustrating the efficacy of our framework in facilitating an automated, scalable approach to evaluating LLM effectiveness. The results, validated through linear regression analysis, demonstrate that selecting the appropriate LLM based on its training and capabilities is essential for optimizing fuzzing efficiency and facilitates the testing of future LLM versions.

INDEX TERMS Fuzzing, machine learning, large language models, python.

I. INTRODUCTION

Fuzzing is a widely recognized technique for uncovering defects in software by profiling program behavior through dynamic analysis [1]. This process systematically generates a series of inputs for specific functions and observes the outcomes, seeking to discover unexpected behaviors or crashes [2]. The effectiveness of fuzzing largely hinges on its ability to explore new logical paths and maximize coverage of the program's instruction set. To facilitate this exploration, a collection of initial seed inputs is often employed. These seeds are designed to guide the fuzzing process into areas of the code that might not be typically reached by random input generation, enhancing the likelihood of finding significant vulnerabilities [3].

However, the generation and selection of effective seeds pose significant challenges, especially when automating the fuzzing of numerous functions at scale. Traditional methods often require seeds to be manually crafted for each specific function, demanding a deep understanding of the expected input formats. This labor-intensive process limits the scalability and efficiency of fuzzing. In response to these challenges, recent advances in machine learning—particularly the development of large language models (LLMs)—offer promising

The associate editor coordinating the review of this manuscript and approving it for publication was Michele Magno.

solutions [4], [5], [6]. These sophisticated models, capable of understanding and generating human-like text, are now being adapted to assist in various computational tasks, including the enhancement of software security through enhanced fuzzing [7].

We specifically investigate the application of LLMs to improve seed generation for fuzzing programs written in Python—a language chosen for its widespread use and a notable deficiency in robust fuzzing tools. The Atheris [8] Python fuzzing engine is employed to evaluate the efficacy of seeds generated by several leading commercial LLMs. Our experimental setup involves multiple unique functions drawn from commonly used Python libraries, providing a broad spectrum of test cases. By comparing the code coverage and instruction counts achieved with LLM-generated seeds against those obtained with traditional seed inputs, we demonstrate that LLMs can significantly enhance the initial phases of fuzzing. Notably, the performance of these seeds varies depending on the specific function under test and the LLM used, suggesting a nuanced relationship between seed effectiveness and the characteristics of both the target function and the model.

This paper makes the following contributions:

- Introduces a publicly-available framework to LLMs for enhancing program behavior fuzzing.[1] This framework systematically evaluates LLM effects on fuzzing coverage across software systems, incorporating extensive tests on 50 key Python libraries and generating corresponding datasets (38,000 seeds in this study). It significantly expands Python fuzzing data and identifies library types that benefit most from LLM-assisted fuzzing, offering key insights into effective model utilization detailed in Section IV-A.
- Highlights the critical role of selecting the right LLMs for generating program inputs in Section IV-B, showing that the appropriate choice of LLM can enhance fuzzing coverage of Python libraries by up to 16% compared to less effective models.
- Demonstrates that variations in large language model (LLM) settings, including temperature selection and prompting style, have a minimal impact on the quality of the resulting fuzzing corpus. This consistency highlights the robustness of LLMs in generating effective fuzzing inputs, supporting streamlined testing with fewer prompting permutations. Further details are discussed in Section IV-C.
- Provides validation of the findings through linear regression testing, as detailed in Section IV-D. The analysis involves fitting several regression models using the collected data to demonstrate the significant predictive ability of code coverage based on settings and LLM selection. This approach yields a closed-form linear formula, presented in Equation 2, capable of estimating code coverage across different models and functions.

[1] https://github.com/gavin-black-dsu/fuzzing_seeds

In this study, the `yaml.load(data)` function is consistently utilized as a representative example among the 50 functions examined. This specific function allows demonstration of the fuzzing process with a clear and concrete set of inputs and outputs. We will detail the harness used for fuzzing as shown in Listing 1, describe a prompt provided to the LLMs in Listing 4, and present samples of the outputs returned in Listings 6 and 7. Finally, the resulting coverage is analyzed as illustrated in Fig. 2 and provided in Table 3.

II. BACKGROUND

Fuzzing is a method in software testing where mutated data is fed into programs to detect security vulnerabilities and bugs [9]. While traditional fuzzing relies heavily on random or manually crafted inputs, leveraging LLMs promises a more automated generation of data that could uncover more complex vulnerabilities. These modern approaches, as highlighted in the survey by Huang et al. [10], seek to address the broader historical challenges on fuzzing outlined by Chen et al. [11], allowing for significant enhancements in the field.

Current LLMs leverage self-attention introduced by Vaswani et al. [12] to learn distributions over large amounts of gathered data. These mechanisms help understand and generate contextually relevant text for a wide range of tasks as discussed by Voita et al. [13]. This ability is crucial for generating meaningful test inputs that are not just random but are structured in a way that is likely to trigger meaningful software interactions.

Controlling the generation process of LLMs is critical, especially in generating structured data. The generation's temperature, a parameter influencing the randomness of output, plays a significant role in balancing between novelty and relevance of the generated seeds [14], [15]. Prompting strategies also impact the effectiveness of these models in fuzzing scenarios [16], guiding the LLMs to produce outputs that are more likely to expose software flaws.

Applications like WhiteFox and InputBlaster showcase the versatility of LLMs in generating test cases and unconventional inputs, thereby broadening the scope of automated testing [17], [18]. These advancements not only enhance the efficiency of testing but also help in understanding and documenting software behavior [19], [20], and even in developing specialized fuzzing drivers for test programs [21].

Several efforts have explored related uses of LLMs for enhanced fuzzing. Projects like Fuzz4All, TitanFuzz, and CHATAFL utilize LLMs to generate or mutate inputs in specific software environments and protocols [22], [23], [24]. For instance, CHATFUZZ leverages LLMs to generate variations of seeds from a greybox fuzzer's pool, aiming to produce high-quality, format-conforming seeds that are more likely to expose hidden vulnerabilities [25].

Our study explores the conditions under which fuzzing techniques such as Fuzz4All and TitanFuzz are most effective, particularly across a broad spectrum of Python functionalities. Unlike previous research, which utilize narrow fuzzing targets like compilers and deep learning libraries, our work extends to a wider range of common libraries with a particular focus on web-facing and input-parsing features. Additionally, multiple commercial LLMs are compared under various settings to underscore the significance of model selection over other factors.

III. METHODOLOGY

To investigate the efficacy of LLM-generated fuzzing seeds, we developed a comprehensive framework designed for the automated collection of results at scale. Initially, for each function listed in Section III-A, corresponding fuzzing drivers were constructed, as elaborated in Section III-B. Subsequently, a selection of commercial LLMs, whose specifics are provided in Section III-C, were employed to generate appropriate fuzzing seeds. These seeds were created following the prompt template outlined in Section III-D and subsequently utilized in various combinations to evaluate their coverage capabilities, as described in Section III-E. The workflow of this methodology are illustrated in Fig. 1, highlighting integration of these components.

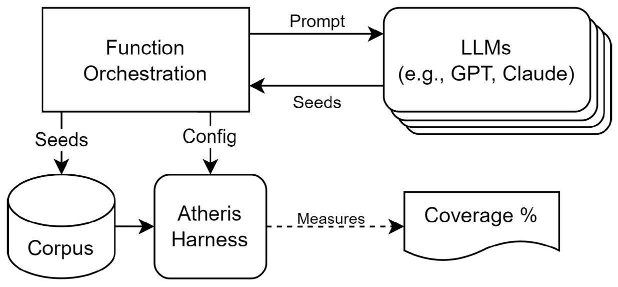


FIGURE 1. Testing pipeline for generating coverage metrics. A prompt is sent to the target language model to generate seeds suitable for fuzzing a specified function. These seeds are then input into an Atheris driver, which runs the function with coverage tracking enabled, thereby producing a percentage of the module's code that is executed during fuzzing.

A. FUNCTION SELECTION

The functions chosen for this study are representative of various layers of application logic—from low-level binary file handling to high-level network requests and data parsing. Each group of functions was selected based on its usage patterns and the type of data it processes, including functions that deserialize data, parse user-supplied inputs, handle multimedia files, and manage network operations. These functions are of particular interest because they inherently interact with external entities and control critical information flows within applications.

The subsequent subsubsections detail the chosen categories of functionality, the specific functions selected, and their brief descriptions as outlined in Table 1. This discussion aims to underscore the diversity of these functions and their importance in general security practices and fuzzing.

1) CONFIGURATION PARSING

This category contains functions that parse standard textual and configuration data formats such as JSON, YAML, XML, and CSV. These formats are ubiquitous in software applications, serving as the backbone for configuration management and data interchange [26]. The importance of rigorously testing these functions is crucial, as vulnerabilities in parsing can lead to critical security flaws like unauthorized data access, injection attacks, and execution of arbitrary code. Since these data formats are commonly used to handle user or system-generated input, their secure parsing is essential for maintaining the integrity and confidentiality of the data they manipulate.

2) BINARY FORMAT PARSING

Functions that deal with binary and complex media formats, including images, audio files, and documents, are categorized here. Parsing such data is inherently riskier due to the possibility of underlying memory issues, improper handling of headers, or execution of embedded malicious code within seemingly innocuous files [27]. These functions are frequent targets in fuzzing because they directly interact with file input/output operations that are susceptible to attacks using specially crafted inputs.

3) NETWORK COMMUNICATION

This category includes functions that facilitate network communications or manage network-related data. Functions such as FTP, SSH, SMTP, and HTTP request handlers are integral to many applications' operations, handling everything from data transfer to remote server management. The security of these functions is critical as vulnerabilities can lead to man-in-the-middle attacks, data leaks, or unauthorized system access [28]. Given the diverse range of protocols and data formats these functions handle, they present a complex attack surface that requires thorough testing to secure against external threats.

4) NETWORK FORMAT HANDLING

This category contains functions that handle various network protocols and complex data formats, particularly those used in web environments such as HTML, CSS, and related data formats. These functions often parse and generate outputs that are directly rendered or executed by client devices. Incorrect parsing or handling of these formats can lead to cross-site scripting, cross-site request forgery, and other web-based attacks, necessitating robust testing [29].

5) TEXT PROCESSING UTILITIES

Functions in this category are focused on processing user inputs and other forms of arbitrary text data. They perform tasks such as data decoding, string splitting, and syntax tokenization. Due to their potential for direct interaction with user-provided data, these functions are susceptible to various forms of input validation vulnerabilities, including SQL injection, command injection, and buffer overflow attacks. Ensuring that these functions handle inputs securely is crucial

TABLE 1. Evaluated functions.

Baseline (%)	Function	Description
	Configuration Parsing	
4.36	`json.loads(data)`	Parses JSON formatted string into a dictionary.
23.30	`yaml.load(data, Loader=yaml.FullLoader)`	Loads YAML formatted data into Python objects.
23.33	`yaml.safe_load(data)`	Safely parses YAML formatted data.
16.42	`django.core.deserialize(data)`	Translate Django objects to configurations.
5.87	`pandas.read_csv(io.StringIO(data))`	Reads CSV formatted data into a DataFrame.
27.20	`geojson.loads(data)`	Parses GeoJSON string into data structures.
7.61	`configparser.read_string(data)`	Parses configuration from a string in INI format.
16.70	`plistlib.dumps(data)`	Serializes Python objects into plist format.
14.11	`toml.loads(data)`	Parses TOML formatted string into a dictionary.
8.59	`simplejson.loads(data)`	Parses JSON data with extendabile options.
12.87	`tablib.import_set(data, format='csv')`	Imports CSV data into a Tablib Dataset.
29.85	`pyexcel.get_sheet(file_content=data)`	Loads data into a PyExcel Sheet object.
	Binary Format Parsing	
17.69	`PIL.Image.open(io.BytesIO(data))`	Opens and identifies an image file as a stream.
27.00	`wave.open(io.BytesIO(data), 'rb')`	Opens a WAV audio file stream for reading.
26.89	`sunau.open(io.BytesIO(data), 'rb')`	Opens an AU audio file stream for reading.
17.12	`pydub.AudioSegment.from_file(data)`	Creates an audio segment from a file stream.
23.25	`mido.MidiFile(file=io.BytesIO(data))`	Parses MIDI data from a file stream.
27.84	`rarfile.RarFile(io.BytesIO(data))`	Opens a RAR archive file from a stream.
14.02	`xlrd.open_workbook(file_contents=data)`	Opens a workbook from an XLS data stream.
38.31	`ics.Calendar(data)`	Parses iCalendar formatted data into objects.
24.74	`iptcinfo3.IPTCInfo(io.BytesIO(data))`	Extracts IPTC metadata from image files.
2.92	`scipy.optimize.minimize(data, 'BFGS')`	Minimizes a function using the BFGS algorithm.
	Network Communication	
19.13	`ftplib.FTP(data)`	Initiates an FTP connection to an address.
20.48	`paramiko.SSHClient().connect(data)`	Establishes an SSH connection.
21.79	`smtplib.SMTP(data)`	Initializes an SMTP client session.
31.46	`requests.get(data)`	Sends an HTTP GET request to the URL.
0.72	`urllib.parse.parse_qs(data)`	Parses a query string storing key-value pairs.
0.56	`urllib.parse.parse_qsl(data)`	Parses a query and stores tuples.
	Network Format Handling	
16.48	`cgi.parse_header(data)`	Parses a MIME header into a dictionary.
30.49	`cgi.parse_multipart(data, pdict)`	Parses multipart/form-data from a stream.
11.74	`email.message_from_string(data)`	Parses an email message from a string.
11.76	`email.parser.parsebytes(data)`	Parses an email message from a byte stream.
11.70	`email.parser.Parser().parsestr(data)`	Parses an email message from a string.
4.16	`email.utils.parseaddr(data)`	Parses a single email address into its components.
4.03	`email.utils.parsedate(data)`	Parses a date string based on e-mail rules.
15.60	`bs4.BeautifulSoup(data, 'html')`	Parses HTML data into a soup object.
23.03	`html.parser.HTMLParser().feed(data)`	Feeds HTML content into a sequential parser.
21.28	`html5lib.parse(data)`	Parses HTML5 content into a tree.
16.35	`markdown.markdown(data)`	Converts Markdown text to HTML.
25.89	`markdown2.markdown(data)`	Alternative Markdown to HTML conversion.
	Text Processing Utilities	
0.78	`ast.literal_eval(data)`	Evaluates and parses an expression from a literal.
40.24	`chardet.detect(data)`	Detects the character encoding used.
30.10	`construct.Struct.parse(data)`	Interprets bytes as structured data.
13.43	`exrex.getone(data)`	Generates a random string that matches a regex.
24.30	`fnmatch.filter([files], data)`	Filters filenames that match a Unix pattern.
13.21	`glob.glob(data)`	Finds pathnames matching a specified pattern.
24.88	`phonenumbers.parse(data, 'US')`	Parses strings representing phone numbers.
3.82	`pygments.lex(data, lexer`	Tokenizes source code for syntax highlighting.
10.84	`quopri.decodestring(data)`	Decodes a string with quoted-printable encoding.
23.57	`shlex.split(data)`	Splits a string using shell-like syntax.

to preventing attackers from exploiting input validation errors to gain unauthorized access or disrupt service [30].

B. FUZZING DRIVERS

To conduct fuzzing effectively, it is important to develop a driver for the specific functionality being tested. These drivers are small programs that provide necessary instrumentation and trigger the execution of the program logic under test [31]. For each function listed in Section III-A, a corresponding driver was created.

Building upon this setup, we selected the Atheris coverage-guided fuzzer, specifically designed for Python, to enhance our testing process. Atheris is actively maintained by Google, ensuring it remains current with evolving software practices. Its integration with libFuzzer allows the tool to leverage byte-level mutations, which are guided by detailed code coverage

```
 1 import atheris
 2 import yaml
 3 import sys
 4
 5 def fuzz_test(data):
 6   try:
 7     yaml.load(data, Loader=yaml.FullLoader)
 8   except yaml.YAMLError as e:
 9     pass
10
11 def main():
12   atheris.Setup(sys.argv, fuzz_test)
13   atheris.Fuzz()
14
15 if __name__ == "__main__":
16   main()
```

LISTING 1. Atheris fuzzing driver for yaml.load.

data, to uncover hard-to-detect bugs in both Python code and native extensions [8], [32]. This capability makes Atheris particularly effective for our project, where comprehensive testing of Python libraries is crucial. The fuzzer's focus on unexplored code paths and its ease of use also contribute to its selection, providing an efficient method to enhance code reliability and security.

An example of the harness used for `yaml.load` testing is provided in Listing 1. The setup begins with the inclusion of the Atheris library on line 1 via `import`, alongside the `yaml` library on line 2, which contains the function under test. The configuration for testing is specified on line 12, based on command-line arguments (`sys.argv`). This configuration includes specifying the number of fuzzing runs, the maximum input size, and an optional seeding corpus for the current test. The `fuzz_test` function, which directs the fuzzing process, starts on line 5. A straightforward `try/except` block within this function aims to capture expected errors known as `yaml.YAMLError`, with any other unanticipated errors being recorded as a crash. The actual testing of the `yaml.load` function happens on line 7, where data provided by the fuzzer is inputted. Here, `Loader=yaml.FullLoader` is used to impose stricter input validations and support advanced Yaml specifications [33].

To facilitate command-line operation, lines 15 and 16 include a handler for the `main` function, allowing the harness to integrate with `coverage` tooling, as detailed in Section III-E. This integration also includes a feature to monitor the `yaml` module within `coverage`, tracking the execution of lines in the `yaml` codebase. The command-line invocation below demonstrates how to run the harness with a *simple* corpus generated by GPT-3.5, specifying the number of runs, maximum input length, and the corpus file:

```
coverage run --source=yaml \
yaml.load_driver.py  \
./gpt3.5_simple_corpus \
-max_len=500 \
-atheris_runs=10000
```

Note that a fixed configuration is used, consisting of 500 bytes maximum input size and 100,000 fuzzing mutations. This configuration is designed to show early coverage gains by incorporating seeds generated from LLMs. The choice of 500 bytes balances complexity with efficiency, allowing for quick test cycles while adequately encapsulating meaningful data structures. The 100,000 runs allow us to conduct multiple repeated trials across various configurations within a reasonable timeframe, even for functions that execute more slowly. These choice of constants represent a trade-off between performance and effectiveness to allow focusing on multiple LLM model selections and settings for validating our framework.

The sole deviation from this standard occurs in tests where only the seeds are used, as illustrated for the `LLM Only` strategy in Fig. 4 and 2. In such instances, the number of runs is adjusted to equal the size of the corpus, ensuring that the test executes only the seeding samples without any further mutations. This approach provides a clear measure of the coverage achievable by the LLM-generated samples in isolation, focusing solely on their potential to increase code coverage.

C. MODEL SELECTION

In the context of software testing, LLMs are increasingly being utilized not only as augmentations but, in some cases, as potential replacements for traditional fuzzing techniques [22], [23]. Our study specifically explores this emerging application, assessing the efficacy of general LLM models in generating effective early inputs. While specialized, format-specific LLM models might offer benefits for specific data types, their development and implementation are not well established at this time. Training such models must address the scarcity of tailored datasets and the complexity of model training for specific formats. Additionally, training our own models would necessitate attention to the diversity of data and justification for the selected transformer architectures.

Furthermore, the few existing format-specific fuzzers tend to be very targeted and are not readily applicable to our broad set of tests [34], [35]. These limitations underscore the practicality of using existing LLMs for software security testing, while highlighting the need for future studies. Such studies should compare format-aware fuzzing against the learned format structures inherent in LLMs, as discussed in Section V-A.

The LLMs selected for our experiments are chosen for their recent development and widespread use across various applications, ensuring their relevance to current trends. Models from leading companies are included: OpenAI's GPT-3.5 and GPT-4, Anthropic's Claude-Instant and Claude-Opus, and Google's Gemini-1.0 [36], [37], [38]. These models were chosen based on their robust performance metrics, extensive usage in the industry, and their integration into popular tools. For example, GPT-4 powers GitHub's Copilot, a widely used AI pair programmer [39], demonstrating its practical

```
Generate several corpus samples for
fuzzing the Python function
"{func_name}"

The 'data' variable indicates the
location for fuzzing.

[The samples should be diverse and have
varying complexity.]

Provide the samples in a Python
array of strings with no commentary.

Ensure that your reply fits in a single
response.
```

LISTING 2. Prompt template for requesting fuzzing seeds.

utility and impact. This selection allows us to explore the capabilities and differences of models that are not only theoretically advanced but also practically proven in real-world applications.

Temperature is a parameter in language models that influences the randomness of the generated outputs [40]. A lower temperature (e.g., 0.0) results in more deterministic and predictable outputs, favoring the most likely pattern of tokens. Conversely, a higher temperature (e.g., 1.0) increases randomness, leading to more diverse and less predictable responses. In our study, we selected temperature values of 0.0, 0.5, and 1.0 to comprehensively assess how this parameter affects the quality and variety of the seeds generated by the models. This range allows us to evaluate the models' performance across a spectrum from conservative to creative outputs, enabling analysis on their practical applications under varying degrees of randomness.

D. PROMPTING

The prompts provided to the large language models are deliberately straightforward, yet follow best practices from multiple sources [41], [42], [43]. Each prompt was crafted to provide the LLMs with a clear context of the task and the expected outputs, along with a specified output format to facilitate automated processing of the results. An optional command was included to elicit samples that are *diverse and of varying complexity*, which provides the distinction between the `simple` and `complex` corpora. The template used for the prompts is provided in Listing 2.

Note that the final directive `Ensure that your reply fits in a single response` is essential due to our testing at low temperatures. It was frequently observed that at a temperature setting of 0.0, outputs often looped indefinitely. A common pattern in many binary samples, for instance, involved repeated null padding, as shown in Listing 3. Such infinite loops would prevent the response from completing, causing the test to fail. This behavior and its implications are further discussed in Section IV-C1. In cases

```
b"\x00\x00\x00\x00\x00\x00\x00\x00\x00\
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00 ...
```

LISTING 3. Repeat padding seen with temperature 0.0.

```
Generate several corpus samples for
fuzzing the Python function
"yaml.load(data)"

The 'data' variable indicates the
location for fuzzing.

Provide the samples in a Python
array of strings with no commentary.

Ensure that your reply fits in a single
response.
```

LISTING 4. yaml.load prompt for a 'simple' corpus.

where corpus generation was compromised by such repetitive patterns, any samples generated prior to the issue were retained. Additionally, due to this behavior, all comparisons other than temperature are conducted with the temperature setting set to 1.0. This approach prevents the skewing of results from truncation issues that predominantly occur at lower temperatures.

The `simple` prompt for yaml.load shown in Listing 4 demonstrates the process. The input string `yaml.load (data)` uses data as the parameter for which samples are provided. This setup, referred to as the `simple` corpus, does not request additional complexity. The output from this prompt is a Python array containing suitable YAML samples in string format, prepared for integration into the testing process.

E. COVERAGE

In our study, we measure the coverage of the Python module being tested as a percentage. This standardizes results across modules regardless of their size or complexity. For example, while a small utility module may have only a few dozen statements, a larger framework could contain thousands. Using percentage coverage rather than absolute statement counts prevents the results from being skewed by the module's size, enabling more meaningful comparisons across different tests.

1) RELATION TO LINES OF CODE

Table 2 lists the total source lines of code (SLOC) for each module, showing that even small percentage increases in coverage can represent hundreds of additional lines of code covered. Additionally, because smaller libraries often rely on their dependencies, expanding the tested branches can significantly increase the total logic covered.

TABLE 2. SLOC by module comparison.

Module	SLOC	Module	SLOC
PIL	14406	pandas	247795
ast	1029	paramiko	8712
bs4	5531	phonenumber	4136
cgi	528	plistlib	533
chardet	1568	pydub	1852
configparser	696	pyexcel	2700
construct	3572	pygments	14226
email	5067	quopri	166
django	69998	rarfile	2094
fnmatch	107	requests	2149
ftplib	549	scipy	166316
geojson	353	shlex	263
glob	106	simplejson	2258
html	330	smtplib	514
html5lib	5756	sunau	331
ics	1827	tablib	1919
iptcinfo3	477	toml	1063
json	597	urllib	2501
markdown	3235	wave	300
markdown2	1603	xlrd	4858
mido	2288	yaml	3609

By expressing coverage as a percentage of the total statements, we provide a scale-independent measure that reflects the proportion of the module's functionality exercised during testing. This metric helps in assessing the effectiveness of the generated seed inputs relative to the entire scope of the module. A significant increase in coverage can be demonstrated when comparing initial runs without LLM-generated inputs to subsequent tests, although it is not expected for a single function to execute every statement in a module.

Our experiments consistently utilize the `coverage` tool to measure code coverage [44]. This tool calculates the number of executed statements compared to the total number of executable statements within a Python module. Its analysis is strictly confined to the Python code, providing a precise measure of which parts of the module are being utilized during tests.

It is important to note that `coverage` does not include calls to external supporting libraries. These are interactions that occur outside the module being tested and are not part of the package itself. Consequently, these external calls are excluded from the coverage statistics. This limitation can result in incomplete representation of the application's behavior, especially in cases involving complex interactions with external systems.

2) BASELINE COVERAGE

Our research concentrates on evaluating the enhancements in code coverage achieved through the use of different fuzzing corpora. To accurately measure the impact of these configurations, it is essential to establish a baseline code coverage level. This baseline accounts for the coverage generated by any initialization processes or common lines of code that are executed regardless of the input provided. Such lines might include setup tasks, default configurations, and error handling routines that are triggered by the absence of inputs [45].

To determine this baseline, we utilize the Atheris driver for the specific function under test using a single empty input. This approach ensures that the function is invoked in its simplest form, without any data that might influence its behavior and inadvertently increase code coverage. Subsequently, this baseline percentage, as detailed in Table 1, is subtracted from the coverage results of all further tests that employ various fuzzing inputs. This subtraction isolates the coverage attributable purely to the input variations introduced by different fuzzing corpora, thereby providing a clearer insight into the effectiveness of each configuration in exploring new execution paths within the module.

3) COVERAGE EXAMPLE

For a concrete example, consider the `yaml.load(data)` function from the YAML processing library. Initially, a baseline for code coverage is established by executing the function with an empty input, `data=''`, as input to the test setup shown in Listing 1. From this baseline run, a coverage of 23.30% is recorded, as shown in Table 1. This number represents the minimum coverage achievable, accounting for the execution of essential code paths that do not depend on input data.

Subsequent tests are conducted both with and without specialized input datasets to observe variations in code coverage. For instance, running the function using inputs from the GPT-4 simple corpus, the recorded coverage increases to 51.21%. This value is adjusting by subtracting the baseline coverage to isolate the impact of fuzzing from the inherent baseline activity of simply loading the library. Thus, the net increase in coverage due to the corpus is calculated as $51.21\% - 23.30\% = 27.91\%$. This adjusted coverage increment, representing the additional code paths explored due to the input from the GPT-4 corpus, is then documented in Table 3.

F. VALIDATION

To enhance the understanding of coverage behaviors, a series of standard linear regression models were trained. Linear regression is a statistical method used to model the relationship between a dependent variable and one or more independent variables by fitting a linear equation to observed data [46]. The primary goal is to find the best-fit line that minimizes the differences between predicted values and actual observations. This approach allows for predicting the dependent variable based on the values of the independent variables, providing insights into how different factors, such as model configurations or operating parameters, influence the overall coverage outcomes in our study.

This analysis was conducted on a per-function basis to accommodate the significant variance observed in coverage metrics. A predictor for each function was created using the mean (μ) of all observed values. Utilizing this method is a common approach for constructing a 'dummy' classifier to compare against other models [47].

The effectiveness of these models was evaluated by measuring the mean squared error (MSE), which was expected to decrease as the correlation to coverage results improved. Additionally, the coefficient of determination (R^2) was monitored, a critical metric for assessing the accuracy of the models in replicating observed outcomes. Higher R^2 values, approaching 1.0, indicate better explanatory power, while values near 0.0 suggest a minimal relationship between the predictors and the outcomes [48].

The models developed serve to test the following null and alternate hypotheses:

H_0 : μ accurately predicts coverage outcomes.

H_a : Linear regression can improve estimation accuracy.

Validation of the alternative hypothesis, H_a, depends on meeting specific statistical criteria. If the MSE for the average-based model, denoted as MSE_{avg}, exceeds that of the linear regression model, MSE_{lnr}, then H_a is considered supported. Additionally, if the coefficient of determination (R^2) for the linear regression model, R^2_{lnr}, surpasses that of the average model, R^2_{avg}, this provides further evidence for H_a. These conditions demonstrate that the linear regression model not only reduces prediction errors but also more effectively captures the variability in the observed data, outperforming a simple mean value predictor.

The variables evaluated in the regression models include Model (M), Temperature (τ), additional fuzzing (Fuzzing), and complexity-focused prompt directives (*Complex*). The Model variable encompasses the distinct LLMs: Claude_I (for Claude Instant), Claude_O (for Claude Opus), GPT_3, GPT_4, and Gemini. These variables are analyzed to determine their correlation with the coverage outcomes observed. Algorithm 1 is employed to provide a structured and comparative analysis of the two competing hypotheses.

To uniformly analyze the coverage deltas across all functions, the data is standardized using z-score normalization as seen in Equation 1. This technique adjusts the scaling to ensure that each function has a mean (μ_f) of 0 and a standard deviation (σ_f) of 1, facilitating comparisons across diverse datasets [49].

$$Coverage_{scaled} = \frac{Coverage_{initial} - \mu_f}{\sigma_f} \tag{1}$$

Importantly, this normalization process is reversible, allowing for the restoration of function-specific values:

$$\begin{aligned}\text{Coverage}_{\text{scaled}} &= (\text{Coverage}_{\text{initial}} - \mu_f)/\sigma_f \\ &\Rightarrow \sigma_f \text{Coverage}_{\text{scaled}} = \text{Coverage}_{\text{initial}} - \mu_f \\ &\Rightarrow \text{Coverage}_{\text{initial}} = \sigma_f \text{Coverage}_{\text{scaled}} + \mu_f\end{aligned}$$

The values derived from the z-score normalization are utilized to formulate a generalized equation that estimates coverage increases based on the study variables, shown in Equation 2. This formula quantifies the relative impact of each variable on the outcome, where the absolute value of the coefficients indicates the strength of each variable's influence. Larger values signify a more substantial impact on coverage changes, regardless of the direction of the effect. This methodological approach enables a comprehensive analysis of how each factor contributes to variations in coverage, providing a nuanced understanding of LLM impacts to fuzzing improvements.

Algorithm 1 Construct and Evaluate Regression Model for Coverage Predictions per Function

```
 1: procedure RegressionModel($D$, $F$)
 2:     Input: Dataset $D$ filtered by Function $F$
 3:     Output: Formula, MSE, $R^2$ statistics
            Data Preparation:
 4:     $D_{\text{encoded}} \leftarrow$ Encode categorical variables in $D$
 5:     $D_{\text{train}}, D_{\text{test}} \leftarrow$ Split $D_{\text{encoded}}$ (20% test data)
            Model Setup:
 6:     $\mu \leftarrow \text{mean}(D_{\text{train}})$
 7:     $\text{avg}(\mu) \leftarrow$ Null hypothesis, mean predictor
 8:     $\text{linear}(D_{\text{train}}) \leftarrow$ Train linear regression Model
            Formula Construction:
 9:     coefficients, intercept $\leftarrow$ Extract from linear
10:     formula $\leftarrow$ intercept
11:     for each coeff in coefficients do
12:         formula $\leftarrow$ formula $+$ coeff $\cdot$ feature
13:     end for
        Null Hypothesis Test:
14:     $\hat{y}_{\text{avg}} \leftarrow \text{avg}(D_{\text{test}})$
15:     $MSE_{\text{avg}} \leftarrow \text{MeanSquaredError}(D_{\text{test}}, \hat{y}_{\text{avg}})$
16:     $R^2_{\text{avg}} \leftarrow R_2\text{Score}(D_{\text{test}}, \hat{y}_{\text{avg}})$
            Model Statistics:
17:     $\hat{y}_{\text{lnr}} \leftarrow \text{linear}(D_{\text{test}})$
18:     $\text{MSE}_{\text{lnr}} \leftarrow \text{MeanSquaredError}(D_{\text{test}}, \hat{y}_{\text{lnr}})$
19:     $R^2_{\text{lnr}} \leftarrow R_2\text{Score}(D_{\text{test}}, \hat{y}_{\text{lnr}})$
            Output:
20:     return formula, $\text{MSE}_{\text{avg}}$, $\text{MSE}_{\text{lnr}}$, $R^2_{\text{avg}}$, $R^2_{\text{lnr}}$
21: end procedure
```

IV. RESULTS AND ANALYSIS

A comprehensive evaluation of the effectiveness of LLMs in enhancing fuzzing coverage across software systems using our framework is presented following the methodology described in Section III. The aim is to determine how different models and their generated corpora contribute to improving software testing outcomes, with the full set of result averages and 95% confidence intervals captured in Table 3.

These findings not only validate the established assessment framework but also shed light on the optimal strategies for employing LLMs in targeted fuzzing applications. The following analysis will explore the implications of these results, focusing on how variations in LLM settings, corpus characteristics, and other operational factors influence the overall effectiveness of the fuzzing process.

A. FRAMEWORK OUTPUTS

Exploring the relationship between LLM configurations and the characteristics of the resulting fuzzing seeds highlights the framework's effectiveness. It assesses how well LLMs can generate valuable fuzzing inputs. Through detailed, function-specific analyses, we show how changes in seed composition and size are influenced by the choice of LLM and the specific prompts used. Our objective is to ensure that the framework

TABLE 3. Average coverage delta per model and function.

Function	Claude-Instant	Claude-Opus	GPT-3	GPT-4	Gemini-1.0	none
PIL.Image.open	1.08 ± 0.54	1.19 ± 0.46	1.31 ± 0.24	3.94 ± 0.47	1.60 ± 0.41	1.32 ± 0.18
ast.literal_eval	2.45 ± 0.38	2.42 ± 0.39	2.41 ± 0.49	2.46 ± 0.23	2.55 ± 0.67	2.25 ± 0.07
bs4.BeautifulSoup	0.73 ± 0.07	0.74 ± 0.06	0.74 ± 0.12	0.72 ± 0.08	0.75 ± 0.05	0.74 ± 0.03
cgi.parse_header	1.39 ± 0.18	1.11 ± 0.63	1.31 ± 0.36	1.48 ± 0.13	1.27 ± 0.62	0.38 ± 0.00
cgi.parse_multipart	4.80 ± 0.18	4.92 ± 0.00	4.84 ± 0.65	4.93 ± 0.05	4.58 ± 1.15	4.62 ± 0.15
chardet.detect	45.50 ± 2.92	46.23 ± 2.31	43.36 ± 4.00	43.61 ± 2.46	45.81 ± 4.70	46.47 ± 0.15
configparser.read_string	9.67 ± 0.32	8.04 ± 2.75	6.71 ± 1.42	13.19 ± 1.04	10.43 ± 1.14	4.89 ± 1.44
construct.Struct.parse	1.43 ± 0.00	1.43 ± 0.00	1.43 ± 0.00	1.43 ± 0.00	1.43 ± 0.00	1.43 ± 0.00
django.core.deserialize	0.02 ± 0.01	0.01 ± 0.00	0.05 ± 0.07	0.38 ± 0.61	0.02 ± 0.01	0.01 ± 0.00
email.message_from_string	0.94 ± 0.30	1.06 ± 0.23	1.22 ± 0.10	2.25 ± 1.35	2.16 ± 1.45	1.19 ± 0.07
email.parser.Parser.parsestr	0.88 ± 0.32	1.05 ± 0.23	1.20 ± 0.07	1.33 ± 0.18	1.85 ± 1.02	1.22 ± 0.00
email.parser.parsebytes	1.92 ± 1.00	4.00 ± 0.22	1.29 ± 0.16	3.37 ± 1.18	1.21 ± 0.09	1.23 ± 0.02
email.utils.parseaddr	2.65 ± 0.28	2.94 ± 0.14	2.89 ± 0.14	3.14 ± 0.11	3.00 ± 0.10	2.80 ± 0.06
email.utils.parsedate	1.32 ± 0.12	1.36 ± 0.08	1.15 ± 0.11	1.36 ± 0.08	1.23 ± 0.11	0.53 ± 0.00
exrex.getone	15.72 ± 1.61	13.68 ± 0.73	10.16 ± 1.69	15.74 ± 1.41	17.30 ± 3.99	7.83 ± 0.80
fnmatch.filter	37.63 ± 2.27	42.79 ± 0.38	39.36 ± 4.16	42.70 ± 0.61	37.77 ± 4.47	36.64 ± 3.98
ftplib.FTP	1.64 ± 0.00	1.64 ± 0.00	1.64 ± 0.00	1.64 ± 0.00	1.09 ± 0.78	1.64 ± 0.00
geojson.loads	14.71 ± 2.13	16.99 ± 0.64	14.51 ± 1.92	16.74 ± 1.32	16.43 ± 0.80	0.00 ± 0.00
glob.glob	29.70 ± 1.40	32.83 ± 3.01	35.36 ± 3.34	35.24 ± 2.84	29.54 ± 4.67	23.58 ± 2.31
html.parser.HTMLParser.feed	6.26 ± 3.64	6.77 ± 4.29	12.15 ± 3.48	12.06 ± 3.67	8.77 ± 4.76	13.09 ± 3.67
html5lib.parse	10.52 ± 1.36	12.53 ± 1.17	7.45 ± 0.79	17.34 ± 1.40	7.79 ± 0.69	2.93 ± 0.08
ics.Calendar	8.91 ± 3.24	8.18 ± 4.17	2.36 ± 0.24	12.49 ± 2.53	7.27 ± 3.98	2.42 ± 0.02
iptcinfo3.IPTCInfo	2.30 ± 1.68	3.07 ± 0.10	3.09 ± 0.65	13.79 ± 2.45	3.05 ± 0.73	3.14 ± 0.00
json.loads	1.55 ± 0.45	1.57 ± 0.41	2.25 ± 0.24	2.27 ± 0.27	1.76 ± 0.56	2.31 ± 0.07
markdown.markdown	13.62 ± 1.39	9.95 ± 4.35	14.70 ± 0.97	12.92 ± 1.22	13.32 ± 2.42	12.09 ± 0.48
markdown2.markdown	12.14 ± 2.92	12.22 ± 4.65	11.22 ± 3.22	10.82 ± 2.16	8.42 ± 2.00	8.56 ± 0.46
mido.MidiFile	0.10 ± 0.05	1.22 ± 0.40	0.13 ± 0.00	4.26 ± 1.89	0.36 ± 0.16	0.13 ± 0.00
pandas.read_csv	0.29 ± 0.03	0.29 ± 0.02	0.31 ± 0.04	0.35 ± 0.07	0.29 ± 0.03	0.32 ± 0.05
paramiko.SSHClient.connect	1.15 ± 3.64	8.92 ± 5.84	7.01 ± 6.26	12.81 ± 0.06	2.97 ± 5.38	10.26 ± 5.03
phonenumbers.parse	4.47 ± 0.24	5.15 ± 0.10	5.00 ± 0.18	5.60 ± 0.24	5.19 ± 0.24	4.71 ± 0.19
plistlib.dumps	23.51 ± 0.90	17.49 ± 11.57	23.20 ± 2.22	25.95 ± 3.22	17.15 ± 11.35	18.80 ± 2.15
pydub.AudioSegment.from_file	1.57 ± 0.71	1.55 ± 0.70	1.87 ± 0.18	1.87 ± 0.10	1.25 ± 0.67	1.88 ± 0.09
pyexcel.get_sheet	1.92 ± 0.02	1.92 ± 0.02	1.92 ± 0.02	1.93 ± 0.00	1.92 ± 0.02	1.93 ± 0.00
pygments.lex	0.04 ± 0.02	0.04 ± 0.02	0.05 ± 0.02	0.05 ± 0.02	0.05 ± 0.01	0.06 ± 0.00
quopri.decodestring	1.20 ± 0.00	1.20 ± 0.00	1.20 ± 0.00	1.20 ± 0.00	1.20 ± 0.00	1.20 ± 0.00
rarfile.RarFile	3.38 ± 2.22	0.29 ± 0.00	0.29 ± 0.00	5.29 ± 1.87	0.29 ± 0.00	0.29 ± 0.00
requests.get	4.71 ± 0.04	15.49 ± 0.97	4.74 ± 0.02	17.28 ± 2.50	14.58 ± 0.18	4.75 ± 0.00
scipy.optimize.minimize	0.08 ± 0.06	0.08 ± 0.06	0.12 ± 0.03	0.12 ± 0.03	0.11 ± 0.04	0.13 ± 0.00
shlex.split	21.16 ± 2.41	21.41 ± 1.52	21.84 ± 1.95	22.16 ± 0.68	22.22 ± 1.24	22.28 ± 0.46
simplejson.loads	0.47 ± 0.11	0.47 ± 0.11	0.51 ± 0.06	0.52 ± 0.04	0.53 ± 0.01	0.53 ± 0.00
smtplib.SMTP	3.18 ± 0.32	2.33 ± 0.00	2.33 ± 0.00	2.46 ± 0.24	2.99 ± 0.46	2.72 ± 0.48
sunau.open	1.21 ± 0.00	2.01 ± 0.57	1.21 ± 0.00	4.05 ± 3.09	1.21 ± 0.00	1.21 ± 0.00
tablib.import_set	1.28 ± 0.27	1.28 ± 0.26	1.40 ± 0.16	1.60 ± 0.04	1.38 ± 0.17	1.43 ± 0.03
toml.loads	28.33 ± 3.87	32.88 ± 3.99	23.12 ± 6.40	35.84 ± 13.55	19.91 ± 3.10	13.08 ± 4.47
urllib.parse.parse_qs	1.60 ± 0.34	1.80 ± 0.05	1.29 ± 0.49	1.82 ± 0.05	1.47 ± 0.52	0.96 ± 0.40
urllib.parse.parse_qsl	0.91 ± 0.42	1.68 ± 0.05	1.21 ± 0.49	1.71 ± 0.04	0.79 ± 0.32	0.88 ± 0.40
wave.open	0.67 ± 0.00	7.38 ± 4.41	0.67 ± 0.00	11.62 ± 1.90	0.67 ± 0.00	0.67 ± 0.00
xlrd.open_workbook	0.74 ± 0.90	1.92 ± 0.04	1.99 ± 0.10	1.48 ± 1.04	1.96 ± 0.05	1.98 ± 0.03
yaml.load	19.75 ± 6.32	23.64 ± 1.55	23.67 ± 3.04	27.91 ± 1.74	22.70 ± 4.27	23.84 ± 0.46
yaml.safe_load	22.27 ± 4.05	23.39 ± 3.63	22.36 ± 3.73	27.28 ± 2.00	22.54 ± 4.28	23.43 ± 0.30

adjusts to various testing environments and consistently offers insights into the practical benefits of using LLMs to improve program testing.

Key artifacts supporting this analysis include Table 3, which presents average fuzzing coverage for all tested LLMs, offering a comparative performance overview. Additionally, Figure 2 details improvements in fuzzing effectiveness for each function, categorized by strategy. Sample outputs from the models illustrate the diversity and relevance of the generated seeds and their correlation with the observed coverage improvements. Moreover, Figure 3 displays graphs that show the number of samples each LLM produces per prompt type, highlighting the impact of different prompt strategies on seed generation.

1) FUNCTION TYPES

The effectiveness of these seeds varied significantly among different functions, as detailed in Fig. 2. Three scenarios were analyzed: `No Corpus`, using only the Atheris fuzzing tool; `LLM + Corpus`, combining Atheris with LLM-generated seeds; and `LLM Only`, employing seeds independently without additional mutations.

The `LLM + Fuzzing` approach was the most effective, improving coverage in 70% of the tested functions (35 out of 50). However, some libraries like `bs4`, `smtplib`, and `xlrd` saw better results with fuzzing alone, often due to the poor quality of LLM-generated seeds. For example, seeds for the `xlrd.open_workbook` function generated by GPT-3.5 were ineffective, leading

```
xlrd.open_workbook(b'important')
xlrd.open_workbook(b'fuzzing')
xlrd.open_workbook(b'random')
xlrd.open_workbook(b'fuzzdata1')
xlrd.open_workbook(b'fuzzdata2')
```

LISTING 5. GPT-3.5 seed samples for xlrd.

to wasted cycles on non-functional inputs as seen in Listing 5.

Further analysis showed that using LLM-generated seeds alone provided better average coverage for libraries like `exrex`, `requests`, and `rarfile`. Performance was comparable to `LLM + Fuzzing` in libraries such as `ics`, attributed to the presence of fewer, higher-coverage seeds. In contrast, traditional fuzzing with 100, 000 mutations tended to lower the overall average by exploring with low-coverage samples in these cases.

Certain libraries, including `construct`, `pyexcel`, and `quopri`, showed no coverage improvement with LLM-generated seeds, with results remaining consistent across different input scenarios. This lack of variation, marked by an absence of confidence intervals, indicates the same level of coverage in every test.

It is important to note that none of the tests achieved complete 100% coverage. The `chardet` library recorded the highest total test coverage, reaching a maximum of 78.66% when utilizing the `LLM + Fuzzing` approach. This indicates that there is potential for further improvement through additional fuzzing techniques.

These findings highlight the variable efficacy of LLM-generated seeds in enhancing fuzzing coverage. While beneficial in most contexts, their impact is limited in others, suggesting a need for a strategic blend of LLM-generated seeds and traditional mutation fuzzing to fully explore program behaviors. This approach demonstrates how to baseline and test LLMs for improvements. It is readily apparent where the models failed to provide meaningful, and sometimes detrimental seeds by analyzing the resulting impact to coverage.

2) SAMPLES GENERATED

The total number of samples generated for each program varied significantly depending on the LLM used, as shown in Fig. 3. Although instructions were given to limit outputs to a single reply, most models produced far fewer than the maximum allowable tokens. The exception was `Claude Opus`, which often reached or exceeded the token limit and required truncation. Importantly, as discussed in Section II, the size of the generated corpora did not correlate with the quality of fuzzing – larger datasets do not necessarily translate to better fuzzing outcomes. This is a critical insight for optimizing fuzz testing strategies, as it suggests a shift in focus from quantity to the quality and relevance of the data generated from LLMs.

The complexity of the directions given, whether simple or complex, had minimal impact on the size of the corpora but significantly influenced the content of the seeds. For instance, complex directives typically resulted in longer samples with more variation. An example from `GPT-4` using our `yaml.load` function shows that the most detailed sample from the `simple` corpus includes type enforcement and sequences, as seen in Listing 6. Conversely, the most elaborate `complex` sample includes shape types, namespace tagging, and comments, as detailed in Listing 7.

```
a:
  b: !!set
    ? !!seq [a]
    ? !!seq [b]
```

LISTING 6. GPT-4 yaml.load simple corpus sample.

```
%TAG ! tag:sample.domain,2002:
---
!shape
  # Use the ! handle for presenting
  x: 1
  y: 1
  width: 10
```

LISTING 7. GPT-4 yaml.load complex corpus sample.

B. MODEL SELECTION

The selection of LLM was the most crucial factor influencing the enhancement of early coverage metrics during fuzzing processes, as depicted in Fig. 4. Among the various models tested, `GPT-3.5` demonstrated the least effectiveness; its samples not only failed to improve coverage but also performed worse than direct fuzzing executed in isolation. This indicates a significant limitation in the `GPT-3.5` model's ability to generate useful fuzzing inputs compared to other models.

On the other hand, `GPT-4` stood out by consistently surpassing all other models in performance, yielding higher averages across the majority of the evaluated functions, as shown in Table 3. This superior performance suggests that `GPT-4`'s advanced training and broader knowledge base likely contribute to generating more effective and diverse test cases, which translates to early gains in fuzzing coverage.

The performance of `GPT-4` was not notably enhanced by the use of the Atheris fuzzing tool. The generated mutations did not substantially enhance the coverage beyond the original samples generated by `GPT-4`. This observation shows that the coverage achieved with an average of 37 samples (see Fig. 3) from `GPT-4` was comparable to that of direct fuzzing using the same set of samples over 100, 000 steps. This could suggest that the initial samples from `GPT-4` were already well-suited for improving initial fuzzing steps, leaving little room for early coverage gains through mutation.

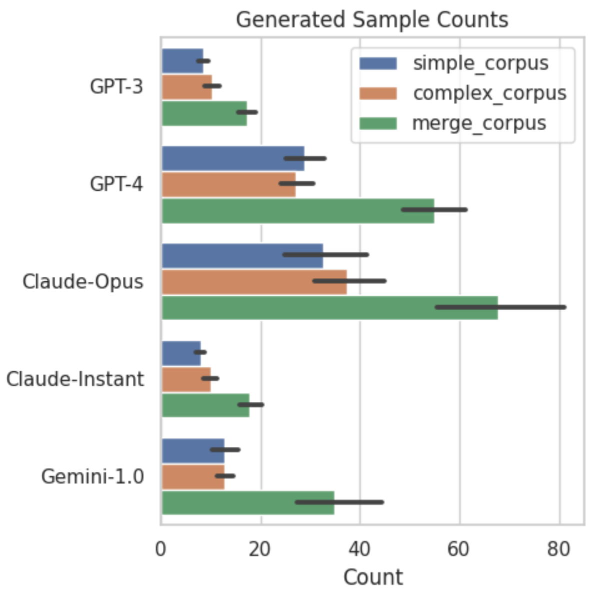


FIGURE 2. Comparison of coverage across the 50 Python functions tested using three strategies: 'No Corpus' (Atheris fuzzer without seeds), 'LLM + Fuzzing' (combines LLM outputs with Atheris), and 'LLM Only' (solely LLM-generated seeds). 95% CI lines on each bar highlight reliability. 'LLM + Fuzzing' consistently shows the best performance, with 'LLM Only' as a close competitor, demonstrating the ability to achieve reasonable coverage gains with a minimal set of targeted samples.

In contrast to `GPT-4`, all other tested models benefited from mutation-based fuzzing applied to their generated seeds. The application of mutations introduced new variations and complexities into the input data, which then exposed more edge cases not covered by the original samples. This improvement underscores the potential value of combining LLM-generated seeds with traditional fuzzing techniques to maximize early coverage and uncover more subtle software bugs.

This conclusion is expected, as transformer-based LLMs are designed to identify patterns and dependencies in sequences of data, constructing probability distributions to generate outputs based on their training data [50]. These distributions are influenced by the characteristics of the datasets and are further refined by techniques such as reward-based feedback [51]. Consequently, a model with more comprehensive distributions that cover a broader array of input types and that has enhancements for structural validity will inherently outperform a model with less effective distributions. Increasing only the model size and the number of parameters, without optimizing these factors, will not necessarily lead to improvements in task-specific performance.

The use of safety systems can introduce notable limitations for LLM-assisted fuzzing, as evidenced by challenges encountered with the `Gemini` model. For example, when using the `ftplib.FTP` function, `Gemini` frequently produced errors citing 'dangerous content', as detailed in Listing 8. Similar issues arose with nearly all network-focused libraries, such as those for ssh and e-mail. The safety mechanisms often activated unexpectedly, necessitating

```
safety_ratings {
  category:
    HARM_CATEGORY_DANGEROUS_CONTENT
  probability: HIGH
}
```

LISTING 8. Gemini safety blocking for FTP testing.

multiple attempts to obtain usable results. Additionally, a recurrent 'recitation' error, where no output was produced because the response was too similar to the training data, required further repetitive testing [52]. Despite these challenges having minimal impact on scoring due to repeated trials, they compromise the model's capability for automated code coverage testing, thus undermining its reliability and effectiveness in many practical scenarios.

C. LLM SETTINGS IMPACT

As mentioned in Section IV-B, the quality of fuzzing seeds is mainly determined by the models' learned distributions. Our experiments show that changing how we sample from these models, such as adjusting the temperature, changing prompts, or varying the sizes of the datasets, has little impact on the results. This indicates that the success of LLMs in fuzzing tasks depends mainly on their training and fine-tuning, rather than on how they are accessed through the API.

1) TEMPERATURE SETTING

Adjusting the temperature parameter (τ), which affects the token choice probability distribution, showed negligible effects on the quality of the generated fuzzing seeds. Setting the temperature to 0.0 led to slightly worse outcomes for both `GPT` models, though these differences were not statistically significant. The relationship between temperature settings and fuzzing coverage is depicted in Fig. 5 and detailed results across both models are provided in Table 4.

To further understand the differences in behavior based on temperature settings, an analysis of the GPT-4 outputs for the `yaml.load` example is shown. At a temperature setting of 0.0, all generated samples exhibit a high degree of similarity in structure, as documented in Listing 9. Specifically, 42 of the 44 samples begin with `a:`, and all but one include the string `!!python`.

Conversely, when the temperature is set to 1.0–as shown in Listing 10–the samples display significantly greater diversity. This includes a lack of repetitive patterns, suggesting that these samples are more likely to cover a wider range of program behaviors. Ultimately, these enhanced samples demonstrated minimal impact on the average final coverage scores.

2) PROMPT DIRECTIVES

The adjustments made to the prompting directives also exhibited minimal impact. Requests for increased complexity and diversity in the samples, as outlined in Listing 2, as well as the combination of outputs from different prompts within

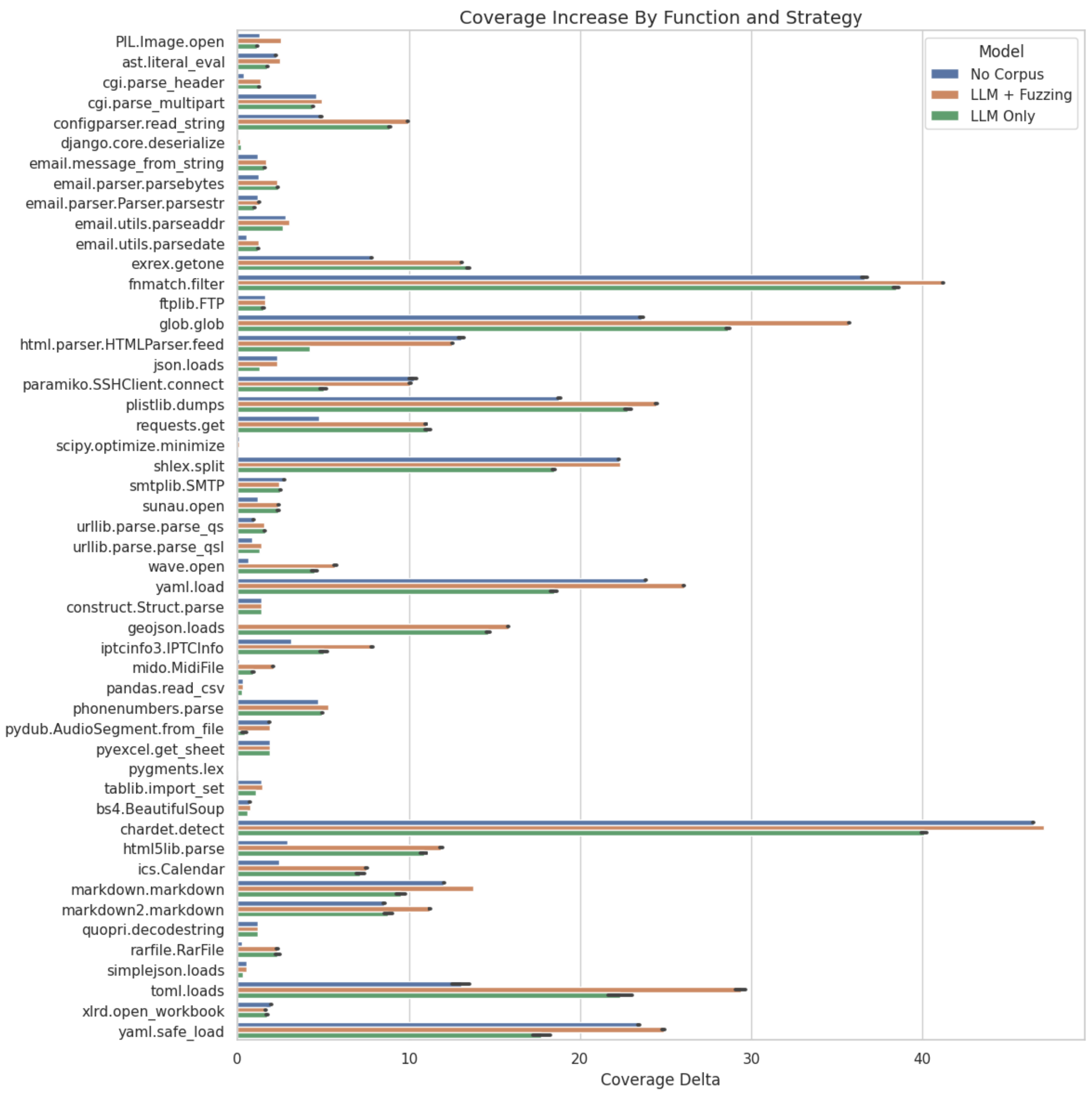


FIGURE 3. Comparison of generated sample counts across three corpora: simple, complex, and their combined samples (merge), annotated with 95% confidence intervals to highlight variability. The results from each LLM tested are shown, emphasizing the influence of model selection on sample volume. This graph underscores the differences between LLM offerings and demonstrates minimal impact on the final corpus size when prompting for additional complexity.

the same models, showed little effect. As depicted in Fig. 6, only the `merge` corpus demonstrated a slight advantage over the corpora generated from `complex` or `simple` prompts. However, on average, the use of LLM-generated corpora resulted in notably better outcomes, especially when compared to fuzzing without seeds, as indicated by the `None` entry.

3) NUMBER OF SAMPLES

No significant correlation was observed between the number of samples generated by a model and the resultant coverage, as illustrated in Fig. 7 where comparisons are made across all models and individual functions. Models that generated larger corpora, such as `Claude Opus`, did not consistently demonstrate a corresponding increase in coverage. However, they occasionally influenced the variation observed in the confidence intervals.

For clarity, the `yaml.load` function is examined in isolation. This analysis is depicted in Fig. 8, showing that the size of the corpus has minimal impact on the average coverage, although the confidence intervals do exhibit fluctuations in certain instances, notably for `Gemini` and `Claude`

TABLE 4. Temperature (τ) averages for GPT-3.5 and GPT-4.

Function	$\tau = 0.0$	$\tau = 0.5$	$\tau = 1.0$
ast.literal_eval	2.44 ± 0.26	2.54 ± 0.28	2.35 ± 0.50
bs4.BeautifulSoup	0.76 ± 0.04	0.76 ± 0.04	0.69 ± 0.13
cgi.parse_header	1.45 ± 0.14	1.33 ± 0.43	1.39 ± 0.19
chardet.detect	47.05 ± 0.27	46.61 ± 0.34	41.45 ± 2.90
construct.Struct.parse	1.43 ± 0.00	1.43 ± 0.00	1.43 ± 0.00
email.message_from_string	1.24 ± 0.02	1.22 ± 0.03	2.60 ± 1.41
email.parser.Parser.parsestr	1.20 ± 0.06	1.21 ± 0.04	1.37 ± 0.20
email.utils.parseaddr	3.01 ± 0.12	3.09 ± 0.18	2.96 ± 0.19
email.utils.parsedate	1.23 ± 0.14	1.22 ± 0.10	1.29 ± 0.17
exrex.getone	13.09 ± 3.08	12.79 ± 2.39	12.91 ± 3.82
fnmatch.filter	41.90 ± 0.87	40.95 ± 3.53	40.32 ± 4.39
ftplib.FTP	1.64 ± 0.00	1.64 ± 0.00	1.64 ± 0.00
geojson.loads	15.90 ± 1.01	15.85 ± 1.01	15.16 ± 2.90
glob.glob	36.03 ± 2.82	36.29 ± 1.23	33.83 ± 3.81
html.parser.HTMLParser.feed	12.71 ± 3.21	13.31 ± 3.06	10.56 ± 3.73
html5lib.parse	11.75 ± 4.81	11.96 ± 4.85	13.01 ± 5.39
ics.Calendar	6.72 ± 5.19	8.03 ± 5.86	7.15 ± 4.99
json.loads	2.30 ± 0.09	2.32 ± 0.07	2.16 ± 0.39
markdown.markdown	13.04 ± 1.06	14.58 ± 1.33	13.90 ± 1.38
markdown2.markdown	8.77 ± 1.46	11.67 ± 2.66	12.48 ± 2.46
paramiko.SSHClient.connect	9.82 ± 5.34	10.67 ± 4.71	9.25 ± 5.66
phonenumbers.parse	5.19 ± 0.33	5.42 ± 0.47	5.26 ± 0.26
plistlib.dumps	22.60 ± 3.80	24.24 ± 1.41	26.50 ± 2.12
pyexcel.get_sheet	1.93 ± 0.00	1.93 ± 0.00	1.91 ± 0.03
quopri.decodestring	1.20 ± 0.00	1.20 ± 0.00	1.20 ± 0.00
requests.get	9.89 ± 5.20	10.18 ± 5.51	12.47 ± 7.86
scipy.optimize.minimize	0.13 ± 0.00	0.13 ± 0.00	0.11 ± 0.05
shlex.split	22.65 ± 0.48	22.07 ± 0.37	21.39 ± 2.19
simplejson.loads	0.53 ± 0.00	0.53 ± 0.00	0.50 ± 0.09
smtplib.SMTP	2.33 ± 0.00	2.53 ± 0.27	2.33 ± 0.00
tablib.import_set	1.51 ± 0.10	1.52 ± 0.09	1.46 ± 0.21
toml.loads	31.16 ± 11.40	33.48 ± 10.81	23.00 ± 11.33
urllib.parse.parse_qs	1.34 ± 0.51	1.50 ± 0.49	1.77 ± 0.07
urllib.parse.parse_qsl	1.53 ± 0.38	1.27 ± 0.49	1.55 ± 0.36
xlrd.open_workbook	1.18 ± 0.95	1.96 ± 0.20	2.16 ± 0.28
yaml.load	25.87 ± 2.44	25.63 ± 2.09	28.95 ± 4.29
yaml.safe_load	25.29 ± 2.11	24.95 ± 2.32	21.70 ± 5.22

```
a: !!python/apply:__builtins__.open ...
a: !!python/apply:__builtins__.eval ...
a: !!python/apply:__import__('os')  ...
a: !!python/apply:__import__('subpr ...
a: !!python/apply:__import__('shuti ...
```

LISTING 9. yaml.load: τ=0.0 samples (One per line).

```
- step: !!omap\n  - id: test1\n  -nam...
receipt: Oz-Ware Purchase Invoice\n d...
- !!python/name:module.submodule.Clas...
a: 'single quoted string with special...
? - Detroit Tigers\n  - Chicago cubs
```

LISTING 10. yaml.load: τ=1.0 samples (One per line).

`Instant`. Despite these variations, the areas of uncertainty are not substantial enough to result in statistically significant overlap with the highest performing model, `GPT-4`.

D. LINEAR REGRESSION TESTING

Multiple linear regression models were developed for each function as detailed in Section III-F. We recorded the MSE and the coefficient of determination (R^2) for the average-based (*avg*) and linear regression (*linear*) models. These correspond to the null hypothesis (H_0) and the alternative hypothesis (H_a), respectively. Metrics and their differences are displayed for each function in Table 5. Functions without notable coverage changes, as specified in Section IV-A1, are excluded from the table. Furthermore, all R^2_{avg} values were consistently near 0.0 and are omitted due to their minimal impact.

In all instances, a significant reduction in prediction error was observed for the linear regression model. This was evident from the comparison of MSE values between the linear regression (MSE_{lnr}) and average-based (MSE_{avg}) models, with a mean difference of 8.34. Even more indicative of the model's effectiveness are the R^2_{lnr} values, which were consistently high, with the majority exceeding 0.5. These results demonstrate a strong correlation between coverage predictions and the variables associated with different models and prompt settings. Considering the specific model, prompt directives, and temperature settings, it is often possible to achieve reasonably accurate predictions of code coverage.

We further explored these correlations across all functions using z-score normalization, as shown in Equation 1. This process yields the coefficients for each variable, presented in Equation 2. The variable with the greatest impact is Fuzzing

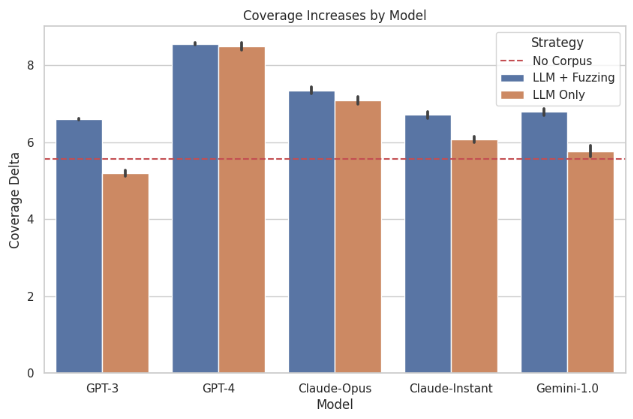


FIGURE 4. Observed coverage increases across different models using 'LLM + Fuzzing' and 'LLM Only' strategies, compared against a baseline represented by a dashed red line for 'No Corpus'. While model selection notably impacts coverage, differences between 'LLM + Fuzzing' and 'LLM Only' are minor, indicating that both strategies effectively enhance coverage beyond the baseline at a similar rate in most cases. These findings underscore the role of LLM model choice in optimizing test coverage.

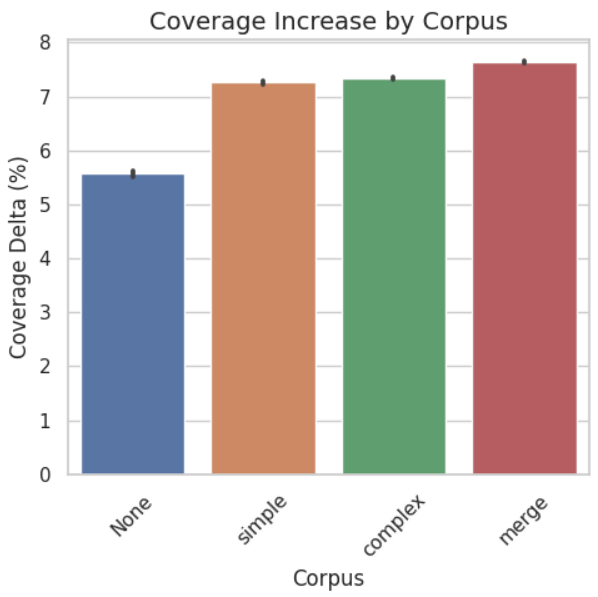


FIGURE 6. Coverage improvements across different corpora types with data from all trials. A marked improvement is seen between the 'None' baseline, which shows significantly lower coverage, and the enhanced results using LLM-generated seeds. While all corpora types improve upon the baseline, the 'Merge' corpus exhibits a slight additional increase. This finding highlights the contributions of the LLM-generated seeds in providing early coverage improvements to fuzzing.

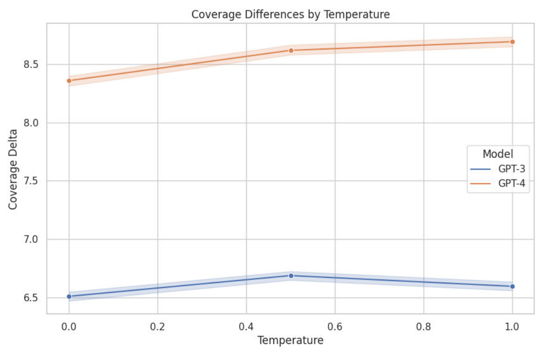


FIGURE 5. Assessment of temperature impact on observed coverage delta between GPT models. The grey areas represent the 95% confidence intervals over all tests. The results show minimal variation in coverage as temperature changes within each model, showing limited influence of the temperature setting. However, the models themselves show notable differences, highlighting the importance of LLM selection over temperature adjustments in optimizing fuzzing strategies.

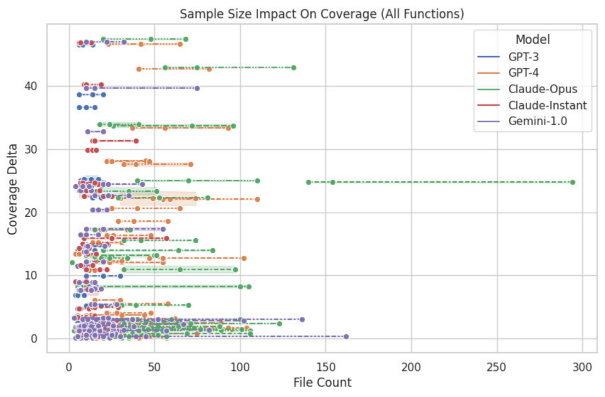


FIGURE 7. Coverage delta as a function of the number of generated samples (File Count), each LLM model is differentiated by line color. Despite varying sample sizes, each set of coverage improvements is consistent for each permutation of model and function. This indicates minimal correlation between the number of samples a given LLM creates and coverage improvements. This consistency shows that early coverage gains are predominantly influenced by the choice of LLM model, consistent with the other findings in the paper.

at 1.06, which indicates an improvement in coverage when using the Atheris fuzzer in conjunction with LLM-generated seeds. Following this, the selection of the model M_{GPT_4}, with a coefficient of 0.89, confirms earlier findings that GPT-4 consistently outperforms other LLMs. In contrast, the use of M_{GPT_3} shows the least improvement, with a coefficient of only 0.01. Overall, the choice of model and the integration of seed generation with fuzzing significantly influences outcomes.

$$\begin{aligned}\text{Coverage}_{\text{scaled}} &= -1.49\\ &+ 0.23 \cdot M_{\text{Claude}_I} + 0.43 \cdot M_{\text{Claude}_O}\\ &+ 0.01 \cdot M_{\text{GPT}_3} + 0.89 \cdot M_{\text{GPT}_4}\\ &+ 0.21 \cdot M_{\text{Gemini}} + 1.06 \cdot \text{Fuzzing}\\ &+ 0.09 \cdot \text{Complex} + 0.19 \cdot \tau \end{aligned} \tag{2}$$

The MSE values and R^2 scores for the unified formula were recorded and showed positive improvements over the average-based predictor:

$$\begin{aligned}&\text{MSE}_{\text{avg}} - \text{MSE}_{\text{lnr}}\\ &\ = 1.0 - 0.71 = 0.29 R^2_{\text{lnr}} - R^2_{\text{avg}} = 0.31 - 0.0 = 0.31\end{aligned}$$

The R^2 value of 0.31 indicates that 31% of the variability in the dependent variable is accounted for by the predictors.

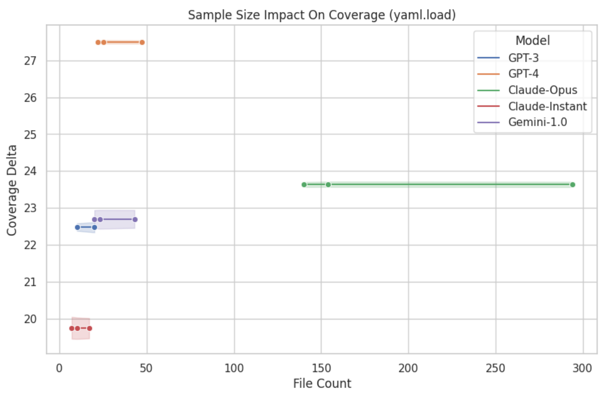


FIGURE 8. **Early fuzzing results comparison for** `yaml.load` **across model types, illustrating the relationship between fuzzing coverage and corpus size. Each model's performance is depicted by a unique color with the 95% confidence intervals shaded around each line. In this case GPT-4 achieves substantial early coverage, a trend that was seen in most other tests. Changes in the number of samples do not significantly impact coverage results, reflecting the similarity of generated constructs in samples sourced from the same LLM.**

This suggests that our all-function model has captured a significant portion of the available explanatory power. However, it does not match the accuracy of the function-specific models, which achieved an average R^2 score of 65%. This outcome is expected, as targeted regression models tend to fit more closely to the underlying dataset.

Overall, the results from the application of linear regression models reveal a notable ability to predict coverage changes, as evidenced by the decrease in MSE and the increase in R^2 values for the linear models compared to average-based predictors. This demonstrates the effectiveness of incorporating specific variables such as model type, fuzzing practices, and temperature settings, which are closely correlated with coverage enhancements.

V. DISCUSSION

This study investigated the role of LLMs in enhancing the effectiveness of fuzzing tasks, particularly in achieving early coverage gains during mutation-based exploration. Our findings reveal that LLM-generated fuzzing seeds readily boost initial coverage in our test scenarios. In certain cases, the use of LLM-generated samples alone can yield results comparable to those of short-duration coverage-guided fuzzing techniques. This demonstrates the potential for LLMs to partially replace traditional methods in specific contexts while using far fewer samples.

The impact of generated samples is strongly influenced by the choice of model, which proved to be the best indicator of early gains. Temperature settings and prompt modifications showed minimal impacts and should be secondary concerns when performing LLM-assisted testing. Despite these promising results, no testing scenario achieved complete code coverage, indicating room for improvement in the application of LLMs for software testing. For instance, the highest total coverage observed was with the `chardet` library, which reached 78.66%. These outcomes highlight the necessity for further refinement and development of LLM strategies to maximize their potential in comprehensive software testing environments.

TABLE 5. **Summary of validation performance metrics.**

Function	$\downarrow$MSE$_{lnr}$	MSE$_{avg}$	$\uparrow R^2_{lnr}$
PIL.Image.open	0.06	1.40	0.95
ast.literal_eval	0.09	0.20	0.52
cgi.parse_header	0.12	0.14	0.18
cgi.parse_multipart	0.26	0.33	0.23
configparser.read_string	1.45	11.85	0.88
django.core.deserialize	0.10	0.15	0.29
email.message_from_string	0.44	1.53	0.71
email.parser.parsebytes	0.41	1.77	0.77
email.parser.Parser.parsestr	0.01	0.06	0.76
email.utils.parseaddr	0.01	0.05	0.79
email.utils.parsedate	0.01	0.02	0.74
exrex.getone	3.10	11.54	0.73
fnmatch.filter	10.26	19.50	0.47
ftplib.FTP	0.00	0.00	0.34
glob.glob	6.24	16.13	0.61
html.parser.HTMLParser.feed	7.48	17.92	0.58
json.loads	0.01	0.18	0.94
paramiko.SSHClient.connect	19.09	34.53	0.45
plistlib.dumps	12.89	24.84	0.48
requests.get	1.45	42.29	0.97
shlex.split	1.22	4.07	0.70
smtplib.SMTP	0.02	0.13	0.83
sunau.open	2.58	4.76	0.46
urllib.parse.parse_qs	0.07	0.13	0.45
urllib.parse.parse_qsl	0.09	0.23	0.59
wave.open	3.38	29.27	0.88
yaml.load	5.05	20.96	0.76
geojson.loads	2.63	6.39	0.59
iptcinfo3.IPTCInfo	2.47	29.94	0.92
mido.MidiFile	1.15	3.57	0.68
pandas.read_csv	0.00	0.00	0.19
phonenumbers.parse	0.03	0.14	0.79
pydub.AudioSegment.from_file	0.01	0.03	0.68
tablib.import_set	0.02	0.06	0.70
bs4.BeautifulSoup	0.00	0.01	0.47
chardet.detect	2.55	14.40	0.82
html5lib.parse	0.45	18.72	0.98
ics.Calendar	7.15	22.35	0.68
markdown.markdown	1.64	3.79	0.57
markdown2.markdown	5.75	11.03	0.48
rarfile.RarFile	1.25	6.54	0.81
simplejson.loads	0.00	0.01	0.88
toml.loads	64.50	110.78	0.42
xlrd.open_workbook	0.26	0.45	0.41
yaml.safe_load	2.71	19.16	0.86
Average	**3.83**	**11.17**	**0.65**

A. FUTURE WORK

Future research will broaden the scope of our framework by incorporating diverse fuzzing harnesses, extended mutation steps, and multiple programming languages, enabling an evaluation of our methods in varied software environments. This will enhance the general applicability and validity of our findings, currently focused on input parsing, binary processing, and network technologies.

In addition to expanding fuzzing harnesses and language diversity, future studies will specifically address the effectiveness of our fuzzing approach in identifying faults. We plan to integrate mutation analysis into our evaluation framework for cases when real-world defects are not available, allowing for a systematic assessment of how well LLM-generated fuzzing seeds can aid in vulnerability discovery. This focus on defect detection will enhance our understanding of the practical impacts of fuzzing, evaluating reliability and security beyond coverage metrics.

Future work will also explore the potential of format-specific LLM models to address the variability in performance attributed to insufficient training on specialized data formats. Recognizing the challenges outlined in Model Selection (Section III-C), such as corpus generation and minimal established work, our research will investigate the feasibility of employing tailored transformer-based models for distinct data types. These efforts will extend the current understanding of LLM capabilities for analyzing software security and their potential limitations.

CONCLUSION

Our study makes multiple contributions to the emerging body of research surrounding the use of LLMs to assist fuzzing and software testing tasks. The key achievements of our research are summarized as follows:

- **Publicly Available Fuzzing Framework**: We introduce a publicly available framework that advances the evaluation of LLMs for fuzzing tasks, systematically assessing their impact across various software systems. This framework has produced an extensive dataset with over 38,000 samples from 50 Python security-relevant libraries, significantly refining fuzzing techniques and identifying libraries that benefit substantially from LLM-enhanced approaches.
- **Optimal LLM Selection**: Our findings confirm that the strategic selection of LLMs can boost fuzzing coverage by up to 16%. This emphasizes the critical role of choosing the proper models for generating program inputs, which significantly enhances the effectiveness of fuzzing coverage across Python libraries.
- **Robustness of LLM Settings**: The study demonstrates that variations in LLM settings, such as temperature and prompting styles, have minimal impact on the quality of the fuzzing corpus. This finding suggests that evaluations can be conducted effectively with fewer samples due to the limited variability.
- **Predictive Strength Testing**: Our regression analysis highlights the predictive strength of our framework, with R^2 scores for linear models surpassing those of average-based predictors. This not only shows the effectiveness of our approach but also offers a reliable method for estimating coverage improvements across different models and functions. By providing a formulaic approach to predict outcomes, our framework assists developers in making informed decisions about LLM integration for fuzzing.

These contributions collectively enhance understanding of LLM effectiveness when integrated into software testing workflows to improve both the efficiency and observed coverage of underlying logic.

ACKNOWLEDGMENT

Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Center for Connected Multimodal Mobility (C2M2) and the official policy or position of the USDOT/OST-R, or any state or other entity. The U.S. Government assumes no liability for the contents or use thereof.

REFERENCES

[1] M. Boehme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Softw.*, vol. 38, no. 3, pp. 79–86, May 2021.

[2] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, *The Fuzzing Book*. Saarbrücken, Germany: CISPA Helmholtz Center for Information Security, 2019.

[3] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. London, U.K.: Pearson Education, 2007.

[4] M. Sallam, "ChatGPT utility in healthcare education, research, and practice: Systematic review on the promising perspectives and valid concerns," *Healthcare*, vol. 11, no. 6, p. 887, Mar. 2023.

[5] S. MacNeil, A. Tran, A. Hellas, J. Kim, S. Sarsa, P. Denny, S. Bernstein, and J. Leinonen, "Experiences from using code explanations generated by large language models in a web software development E-book," in *Proc. 54th ACM Tech. Symp. Comput. Sci. Educ.*, 2023, pp. 931–937.

[6] A. Jungherr, "Using ChatGPT and other large language model (LLM) applications for academic paper assignments," *SocArXiv*, Mar. 2023, doi: 10.31235/osf.io/d84q6.

[7] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," 2023, *arXiv:2307.06435*.

[8] Google. *Atheris: A Coverage-Guided, Native Python Fuzzer*. Accessed: Nov. 22, 2023. [Online]. Available: https://github.com/google/atheris

[9] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021.

[10] L. Huang, P. Zhao, H. Chen, and L. Ma, "Large language models based fuzzing techniques: A survey," 2024, *arXiv:2402.00350*.

[11] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Comput. Secur.*, vol. 75, pp. 118–137, Jun. 2018.

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–11.

[13] E. Voita, D. Talbot, F. Moiseev, R. Sennrich, and I. Titov, "Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned," 2019, *arXiv:1905.09418*.

[14] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proc. 6th ACM SIGPLAN Int. Symp. Mach. Program.*, 2022, pp. 1–10.

[15] M. Renze and E. Guven, "The effect of sampling temperature on problem solving in large language models," 2024, *arXiv:2402.05201*.

[16] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Comput. Surv.*, vol. 55, no. 9, pp. 1–35, Sep. 2023.

[17] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "WhiteFox: White-box compiler fuzzing empowered by large language models," 2023, *arXiv:2310.15991*.

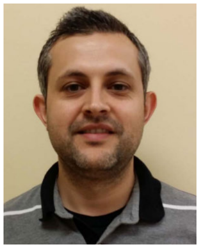
[18] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, Z. Tian, Y. Huang, J. Hu, and Q. Wang, "Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, Apr. 2024, pp. 1–12.

[19] J. Carver, R. Colomo-Palacios, X. Larrucea, and M. Staron, "Automatic program repair," *IEEE Softw.*, vol. 38, no. 4, pp. 122–124, Jul. 2021.

[20] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Oct. 2017, pp. 50–59.
[21] C. Zhang, Y. Zheng, M. Bai, Y. Li, W. Ma, X. Xie, Y. Li, L. Sun, and Y. Liu, "How effective are they? Exploring large language model based fuzz driver generation," 2023, *arXiv:2307.12469*.
[22] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, Apr. 2024, pp. 1–13.
[23] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2023, pp. 423–435.
[24] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proc. 31st Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2024, pp. 1–17.
[25] J. Hu, Q. Zhang, and H. Yin, "Augmenting greybox fuzzing with generative AI," 2023, *arXiv:2306.06782*.
[26] H. Dai, C. Murphy, and G. Kaiser, "Configuration fuzzing for software vulnerability detection," in *Proc. Int. Conf. Availability, Rel. Secur.*, Feb. 2010, pp. 525–530.
[27] T. Zhang, W. H. Lee, M. Gao, and J. Zhou, "File guard: Automatic format-based media file sanitization: A black-box approach against vulnerability exploitation," *Int. J. Inf. Secur.*, vol. 18, no. 6, pp. 701–713, Dec. 2019.
[28] G. Xiong, J. Tong, Y. Xu, H. Yu, and Y. Zhao, "A survey of network attacks based on protocol vulnerabilities," in *Proc. Web Technol. Appl.*, Changsha, China. Cham, Switzerland: Springer, 2014, pp. 246–257.
[29] A. Singh, B. Singh, and H. Joseph, "Vulnerability analysis for mail protocols," in *Vulnerability Analysis and Defense for the Internet*. Cham, Switzerland: Springer, 2008, pp. 47–70.
[30] T. Scholte, D. Balzarotti, and E. Kirda, "Have things changed now? An empirical study on input validation vulnerabilities in web applications," *Comput. Secur.*, vol. 31, no. 3, pp. 344–356, May 2012.
[31] E. Rigtorp. *Fuzzing/Docs/Structure-Aware-Fuzzing.md at Master · Google/Fuzzing*. Accessed: Feb. 15, 2024. [Online]. Available: https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md
[32] *LibFuzzer—A Library for Coverage-Guided Fuzz Testing.—LLVM 19.0.0Git Documentation*. Accessed: Feb. 15, 2024. [Online]. Available: https://llvm.org/docs/LibFuzzer.html
[33] *GitHub—Yaml/Pyyaml: Canonical Source Repository for PyYAML—Github.com*. Accessed: May 21, 2024. [Online]. Available: https://github.com/yaml/pyyaml
[34] R. Dutra, R. Gopinath, and A. Zeller, "FormatFuzzer : Effective fuzzing of binary file formats," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, pp. 1–29, Dec. 2023, doi: 10.1145/3628157.
[35] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: Automatic grey-box fuzzing for structured binary formats," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2020, pp. 1–13.
[36] OpenAI et al., "GPT-4 technical report," 2023, *arXiv:2303.08774*.
[37] *Introducing the Next Generation of Claude*. [Online]. Available: https://www.anthropic.com/news/claude-3-family
[38] G. Team et al., "Gemini: A family of highly capable multimodal models," 2023, *arXiv:2312.11805*.
[39] T. Dohmke. *GitHub Copilot X: The AI-Powered Developer Experience—Github.Blog*. Accessed: May 29, 2024. [Online]. Available: https://github.blog/2023-03-22-github-copilot-x-the-ai-powered/
[40] L. Tunstall, L. Von Werra, and T. Wolf, *Natural Language Processing With Transformers*. Sebastopol, CA, USA: O'Reilly Media, 2022.
[41] J. Shieh. *Best Practices for Prompt Engineering With OpenAI API*. OpenAI. Accessed: Jun. 28, 2024. [Online]. Available: https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering
[42] J. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, "Why Johnny can't prompt: How non-AI experts try (and fail) to design LLM prompts," in *Proc. CHI Conf. Hum. Factors Comput. Syst.* New York, NY, USA: ACM, 2023, pp. 1–21, doi: 10.1145/3544548.3581388.
[43] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with ChatGPT," 2023, *arXiv:2302.11382*.
[44] *Coverage.py—Coverage.py 7.5.1 Documentation*. Accessed: Nov. 22, 2023. [Online]. Available: https://coverage.readthedocs.io/en/7.5.1/
[45] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*. Hoboken, NJ, USA: Wiley, 2008.
[46] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to Linear Regression Analysis*. Hoboken, NJ, USA: Wiley, 2021.
[47] *DummyClassifier—Scikit-Learn.org*. Accessed: Jun. 6, 2024. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html
[48] D. Zhang, "A coefficient of determination for generalized linear models," *Amer. Statistician*, vol. 71, no. 4, pp. 310–316, Oct. 2017.
[49] S. G. K. Patro and K. K. Sahu, "Normalization: A preprocessing stage," 2015, *arXiv:1503.06462*.
[50] S. Chan, A. Santoro, A. Lampinen, J. Wang, A. Singh, P. Richemond, J. McClelland, and F. Hill, "Data distributional properties drive emergent in-context learning in transformers," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 18878–18891.
[51] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, S. Zhong, B. Yin, and X. Hu, "Harnessing the power of LLMs in practice: A survey on ChatGPT and beyond," *ACM Trans. Knowl. Discovery Data*, vol. 18, no. 6, pp. 1–32, Jul. 2024.
[52] *GenerateContentResponse | Google AI for Developers | Google for Developers—AI.Google.dev*. Accessed: May 30, 2024. [Online]. Available: https://ai.google.dev/api/rest/v1/GenerateContentResponse

GAVIN BLACK (Member, IEEE) received the B.S. degree in computer science and mathematics from Purdue University and the M.S. degree in applied and computational mathematics from the University of Massachusetts. He is currently pursuing the Ph.D. degree in computer and cyber sciences with Dakota State University, Madison, SD, USA. He is a Senior Researcher with Leidos Inc., with a focus on machine learning applications for cybersecurity problems. He previously worked as the Technical Lead, a Researcher, and a Subject Matter Expert for various government agencies through the MITRE Federally Funded Research Center.

VARGHESE MATHEW VAIDYAN received the Bachelor of Technology degree from the University of Calicut, India, the M.S. degree from the University of Glasgow, U.K., and the Ph.D. degree from Iowa State University. He is currently an Assistant Professor with the Beacom College of Computer and Cyber Sciences, Dakota State University, Madison, SD, USA. His areas of research interests include the IoT security and machine learning. His publications cover the IoT device security and hybrid quantum architecture-based methods. In addition, he is a regular reviewer of multiple journals and conferences, including several IEEE papers.



GURCAN COMERT received the B.Sc. and M.Sc. degrees in industrial engineering from Fatih University, Türkiye, in 2003 and 2005, respectively, and the Ph.D. degree in civil engineering from the University of South Carolina, Columbia, SC, USA, in 2008. He is currently with Benedict College. He is also the Associate Director of the USDOT Center for Connected Multimodal Mobility (C2M2) and a Researcher with the Information Trust Institute, University of Illinois Urbana–Champaign. His research interest includes applications of probabilistic models to transportation problems.