HFUZZER: Testing Large Language Models for Package Hallucinations via Phrase-based Fuzzing

Yukai Zhao*†, Menghan Wu†, Xing Hu†‡, Xin Xia†
*School of Software Technology, Zhejiang University, Ningbo, China
†The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China
{yukaizhao2000, menghanwu, xinghu}@zju.edu.cn, xin.xia@acm.org

***Abstract*—Large Language Models (LLMs) are widely used for code generation, but they face critical security risks when applied to practical production due to package hallucinations, in which LLMs recommend non-existent packages. These hallucinations can be exploited in software supply chain attacks, where malicious attackers exploit them to register harmful packages. It is critical to test LLMs for package hallucinations to mitigate package hallucinations and defend against potential attacks. Although researchers have proposed testing frameworks for fact-conflicting hallucinations in natural language generation, there is a lack of research on package hallucinations. To fill this gap, we propose HFUZZER, a novel phrase-based fuzzing framework to test LLMs for package hallucinations. HFUZZER adopts fuzzing technology and guides the model to infer a wider range of reasonable information based on phrases, thereby generating enough and diverse coding tasks. Furthermore, HFUZZER extracts phrases from package information or coding tasks to ensure the relevance of phrases and code, thereby improving the relevance of generated tasks and code. We evaluate HFUZZER on multiple LLMs and find that it triggers package hallucinations across all selected models. Compared to the mutational fuzzing framework, HFUZZER identifies 2.60× more unique hallucinated packages and generates more diverse tasks. Additionally, when testing the model GPT-4o, HFUZZER finds 46 unique hallucinated packages. Further analysis reveals that for GPT-4o, LLMs exhibit package hallucinations not only during code generation but also when assisting with environment configuration.**

***Index Terms*—Large Language Models, Package Hallucination, Fuzzing**

I. INTRODUCTION

Large Language Models (LLMs) have shown significant potential across various domains and are widely used for code generation [1]. However, despite their success in tackling complex tasks, LLMs face critical challenges related to security and privacy [2], [3], [4], [5]. One major issue is hallucination, where LLM-generated outputs may appear credible or authentic but are factually incorrect, self-contradictory, or unrelated to inputs [6], [7]. This issue has been extensively studied in natural language generation (NLG) [8]. Recently, Liu et al. [5] explore hallucinations in code generation and classify code hallucinations into 19 types. One critical hallucination type is *package hallucination*, which is defined as LLMs recommend packages or libraries that do not exist [9], [10].

Compared to other types of code hallucinations, package hallucination poses a higher risk of malicious exploitation, introducing new software supply chain security threats [11], [9]. Such attacks often fall under package obfuscation incidents, where developers are misled into importing packages they do not expect, which is one of the most serious problems in supply chain security [10], [12]. Figure 1 shows an example. A user first prompts the model to generate a Python program that implements a simple HTTP/2 server with some specific requirements. After generating the code, the user then asks how to install the packages used in the generated program. In response, the model recommends two packages to be installed via pip. Upon inspection, it is found that the package "h2" is correct, while "hyper-h2" is a hallucinated package that does not exist in the package repository (PyPI [13]). If an attacker registers this hallucinated package in the repository and embeds malicious code within it, uninformed developers may inadvertently install and execute it, exposing themselves to supply-chain attacks. Moreover, researchers and practitioners have developed various LLM agents for end-to-end software development (such as Devin [14]), which are capable of using tools and executing commands [15]. This further increases the success rate of package hallucination-based attacks, as malicious packages may be downloaded into the development environment without the developers' awareness.

Spracklen et al. [10] and Krishna et al. [9] construct datasets and empirical studies on package hallucinations. While valuable, their studies are limited by the size of the dataset and cannot cover a wide range of code generation scenarios. Although Drowzee [16] is proposed to test LLMs for fact-conflicting hallucinations in NLG, research on code hallucinations—particularly on package hallucinations—remains scarce. Inspired by testing software to help discover failures, we propose to test LLMs for package hallucinations. However, testing LLMs faces the following two key challenges:

• **Challenge 1: How to cover as many code generation scenarios for LLMs as possible?** To cover these scenarios, we need to generate adequate and diverse coding tasks to test LLMs. Although Drowzee [16], MORTAR [17], and MetaQA [18] are proposed to generate natural language questions with single correct answers (e.g., "Did Haruki Murakami and Bob Dylan ever win the same award?"), they cannot complete the generation of coding tasks with multiple correct implementations. Moreover, approaches that mutate existing tasks using predefined mutation rules often result in limited task diversity, as the variations are bounded by the original task structure and mutation rules. Although such strategies may be

‡Corresponding Author

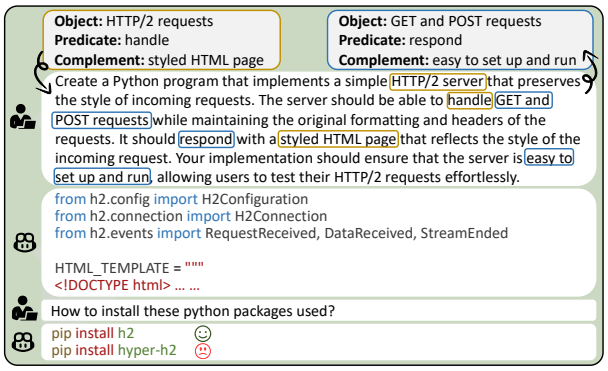

Fig. 1. An Example of Package Hallucination

effective in adversarial attacks like jailbreak attacks, they are inadequate for generating diverse coding tasks.

• **Challenge 2: How to generate code-relevant tasks to test LLMs for package hallucinations?** Non-code-related tasks introduce conflict in the prompt of the model (e.g., asking an LLM to generate code for the task of "Win the Nobel Prize in literature"), which undermines the effectiveness of testing LLMs for package hallucinations. Therefore, automatically generated tasks must be restricted to code-related ones. However, leveraging LLMs or applying simple mutation-based approaches often cannot ensure that the constraint is met, especially when diversity is also desired. This issue may lead to the generation of non-code-related tasks.

To overcome these challenges, we propose HFUZZER, a novel phrase-based fuzzing framework designed to test LLMs for package hallucinations. For **Challenge-1**, HFUZZER adopts fuzzing technology and leverages LLMs as a task generation engine, which enables HFUZZER to generate adequate tasks. To increase the diversity of tasks, HFUZZER guides LLM to generate tasks based on phrases. Since LLM has been pre-trained on a large amount of data, when phrases are input, the model can use this knowledge to infer a wider range of reasonable information, thereby improving the diversity of tasks. For instance, as shown in Figure 1, HFUZZER leverages LLM to generate a coding task based on phrases `HTTP/2 requests`, `handle`, and `styled HTML page`. In the generated task, LLM infers the task "implement an HTTP/2 server" based on the phrase `HTTP/2 requests`. For **Challenge-2**, HFUZZER extracts phrases from package information or coding tasks to ensure that the phrases are relevant to the code, thereby improving the relevance of the generated tasks and code. For instance, the phrase `HTTP/2 requests` is more conducive to the model inferring code-related information than `Nobel Prize`. Inspired by the fact that subject-predicate-object triples can summarize information in three phrases, HFUZZER formulates package information or coding tasks as phrase compositions $\langle Object, Predicate, Complement\rangle$ to obtain richer phrases and avoid redundancy. Among them, $Object$ represents the object processed by the package or the coding task (e.g., `HTTP/2 requests` and `Get and Post Requests` in Figure 1), $Predicate$ represents the method applied to $Object$ (e.g., `handle` and `respond` in Figure 1), and $Complement$ represents additional relevant details (e.g., `styled HTML page` and `easy to set up and run` in Figure 1). Based on the extracted phrases, HFUZZER guides LLM to consider the packages related, and then restricts the tasks to those that can be solved by using packages, thus increasing the likelihood of generating code-related tasks that require calling packages.

To evaluate HFUZZER, we use the descriptions of the top 100 Python packages in libraries.io [19] as the initial input and set the budget of a run as 1000 rounds. We count the number of unique hallucinated packages to evaluate the effectiveness of HFUZZER and cluster the generated tasks to analyze diversity. We compare HFUZZER with GPTFUZZER-A, which is adapted from GPTFuzzer [20], and use nine models as tester and target models to comprehensively assess the generalizability of HFUZZER. The tester model is the model used by HFUZZER and GPTFUZZER-A; the target model is the model tested. Our results show that HFUZZER successfully triggers package hallucinations in all target models and outperforms GPTFUZZER-A across all tester models, finding on average 2.60x more unique hallucinated packages. Tasks generated by HFUZZER are more diverse than those from GPTFUZZER-A. We also find 46 unique hallucinated packages recommended by GPT-4o. Further analysis shows that for GPT-4o, package hallucinations occur not only during code generation but also when assisting with environment configuration.

The contributions of our paper are summarized as follows:

- We design a new phrase-based coding task generation method that leverages the knowledge of the LLM to infer a wider range of reasonable information based on phrases, thereby generating diverse tasks.
- To our knowledge, our framework is the first to introduce the concept of fuzzing into testing LLMs for package hallucinations. The code can be found on our website [21].
- We conduct a comprehensive evaluation by using different LLMs as tester and target models. Results show that HFUZZER successfully triggers package hallucinations in all target models and outperforms GPTFUZZER-A on all tester models. Tasks generated by HFUZZER are also more diverse than those generated by GPTFUZZER-A.

II. METHODOLOGY

Figure 2 provides an overview of HFUZZER, which includes two parts (i.e., Phrase Extraction and Fuzzing Loop) and resembles the fuzzing process (Seed Pool Initialization, Seed Selection, Seed Mutation, and Execution). The tester model LLM_{tester} is the model used by HFUZZER, and the target model LLM_{target} is the tested model. HFUZZER uses package information, which includes package names (e.g., "requests") and their descriptions (e.g., "Python HTTP for Humans"), as input and tests closed-source models solely by accessing their input/output. The fuzzer first executes the program under test with inputs from testers to initialize the seed pool (Seed Pool Initialization). Similarly, in the Phrase Extraction phase, HFUZZER extracts phrase compositions $\langle Object, Predicate, Complement\rangle$ from the package

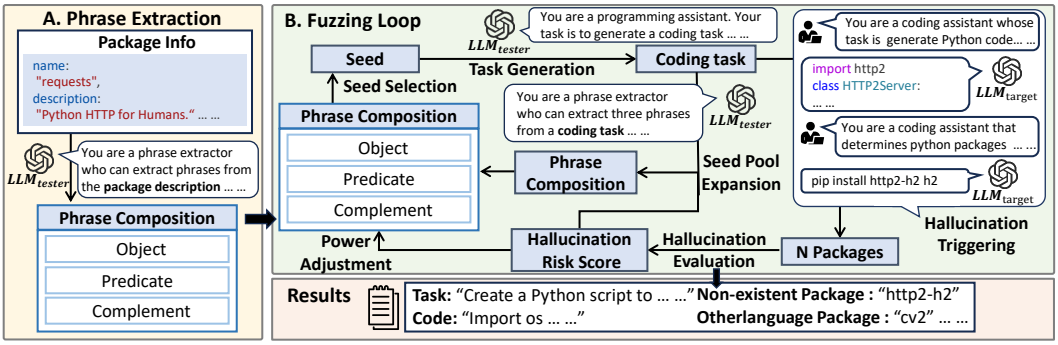

Fig. 2. The overview of HFUZZER. Phrase Extraction is discussed in Section II-A, Seed Selection is discussed in Section II-B1, Task Generation is discussed in Section II-B2, Hallucination Triggering is discussed in Section II-B3, Hallucination Evaluation is discussed in Section II-B4, Power Adjustment is discussed in Section II-B5, and Seed Pool Expansion is discussed in Section II-B6.

information by LLM_{tester} to construct three corresponding phrase pools, which consist of the seed pool (Section II-A). Subsequently, HFUZZER enters the Fuzzing Loop phase. Similar to selecting seeds from the seed pool (Seed Selection), in each round, HFUZZER selects a seed consisting of three phrases from the seed pool according to the power of phrases (Section II-B1). The power is defined as the potential of a phrase, which determines the probability of the phrase being selected. The seed is used by LLM_{tester} to generate a coding task (Section II-B2). Unlike traditional fuzzing generates new inputs by mutating seeds (Seed Mutation), HFUZZER generates new coding tasks by recombining different phrases. In the Hallucination Triggering phase, HFUZZER asks LLM_{target} to generate code for the provided task, and then guides it to recommend packages required to execute code, which corresponds to the Execution of fuzzing. HFUZZER performs hallucination evaluation on packages recommended by LLM_{target} to calculate the Hallucination Score (HS), which is used to measure the package hallucination triggering on LLM_{target} (Section II-B4), and adjusts the power of related phrases according to the HS (Section II-B5), thereby guiding future selection toward under-explored phrases. To expand the seed pool, for tasks that trigger package hallucinations, HFUZZER extracts new phrase compositions from them and adds new phrase compositions to the seed pool (Section II-B6). The process continues until the query budget is exhausted. Finally, we get the generated coding tasks and corresponding model outputs. In the above process, to handle incorrect responses, HFUZZER discards intermediate results if the output format is invalid (e.g., missing tags) and allows LLM_{tester} to respond with "None" when it is unable to generate reasonable output. HFUZZER verifies packages by querying package indices (e.g., PyPI) to avoid false positives due to LLM_{tester} output errors. LLM_{target} is also allowed to refuse unrelated coding tasks. Additionally, compared to traditional parsing methods, LLMs provide stronger semantic understanding and natural language processing, allowing them to handle complex inputs.

A. Phrase Extraction

HFUZZER formulates package information as the phrase composition $\langle Object, Predicate, Complement\rangle$ and extracts the phrase composition from each package's information to construct the seed pool. This seed pool is divided into three corresponding phrase pools with power, each containing phrases of one component type. The three phrases of the composition are defined as follows:

- $Object$ represents the object processed by the package.
- $Predicate$ represents the method applied by the package.
- $Complement$ represents a phrase that captures essential contextual details beyond the $Object$ or $Predicate$, providing additional context when applicable.

To achieve this, HFUZZER leverages LLM to extract phrase compositions from package information. HFUZZER configures the system prompt to define the target of phrase extraction from package information and provides an example to specify the expected response format. In the user prompt, HFUZZER provides the package information to the model to obtain phrases. These phrases are added to their respective phrase pools and assigned an initial power, influencing the selection process described in Section II-B1. To handle cases where the model cannot extract a specific phrase due to insufficient information, the prompt instructs the model to answer "None". This design helps minimize the negative impact of incomplete package descriptions on phrase quality.

Example A. The description of package `pre-commit` is "*A framework for managing and maintaining multi-language pre-commit hooks*". We formalize it as $\langle$*pre-commit hooks, managing and maintaining, multi-language support*$\rangle$. Among them, `pre-commit hooks` represents the objects processed by this package, `managing and maintaining` represents the methods provided by this package, and `multi-language support` supplements the features of this package. HFUZZER extracts such information from package information to construct the seed pool.

B. Fuzzing Loop

Similar to fuzzing, we consider the following process as one round and repeat it until the budget is exhausted.

1) Seed Selection: In directed fuzzing, fuzzers assign scores to the seeds and prioritize those with higher scores for mutation. HFUZZER adopts a similar strategy for seed selection. In this paper, each seed consists of three phrases corresponding to the phrase composition described in Section II-A, to increase the amount of information and provide richer context for LLM. Specifically, HFUZZER applies a weighted random selection algorithm to construct a seed: one phrase is selected from

each phrase pool, where the selection probability of a phrase is proportional to its power relative to the total power of the phrase pool. The power is initialized in Section II-A and continuously adjusted to reduce the likelihood of repeatedly selecting the same phrases. The details of power adjustment are presented in Section II-B5.

Example B1. In Figure 1, HFUZZER selects one phrase from each of three phrase pools to construct a phrase composition: $\langle HTTP/2\ requests,\ handle,\ styled\ HTML\ page \rangle$.

2) Task Generation: HFUZZER exploits LLM to generate a coding task based on a selected seed. Specifically, HFUZZER leverages LLM to identify packages that may be relevant to the provided phrases and to generate a coding task that can be solved using those packages. To enable automatic extraction of the task from the model's response, HFUZZER includes a formatting example in the prompt and requires the LLM to enclose the generated task within a predefined tag. During execution, HFUZZER inserts three phrases of the selected seed into a preset user prompt, queries the model, and parses the tagged output to extract the final coding task.

Example B2. Based on the phrase composition, HFUZZER uses LLM_{tester} to generate a coding task, shown in Figure 1. This task requires implementing an HTTP/2 server (HTTP/2 requests) that can handle GET and POST requests (handle) and respond with a styled HTML page (style HTML page).

3) Hallucination Triggering: To trigger package hallucination in the common LLM usage scenario, we divide the entire triggering process into two stages: Code Generation and Package Recommendation. In Code Generation, HFUZZER guides the target model to generate code that solves the generated task, providing an example to clarify the expected response format. To mitigate the influence of unintended coding tasks (e.g., non-code-related tasks), the prompt explicitly instructs the model to return "None" if it cannot produce code. In Package Recommendation, HFUZZER prompts the target model with the generated code and requires the model to answer the installation commands corresponding to the packages used in the code. Based on the package answered, HFUZZER further performs hallucination evaluation. This design simulates a realistic user scenario (developer requests a code snippet along with the required packages) and reduces false positives from aliases and similar ambiguities compared with rule-based package extraction. Additionally, Package Recommendation tests LLMs on environment configuration, motivated by existing studies that use LLMs for this purpose (e.g., in automatic Docker builds [20]). It also supports analyses of hallucinations from mismatches between imported modules (e.g., cv2) and actual package names (e.g., opencv-python) [10].

Example B3. In Figure 1, HFUZZER first prompts LLM_{target} to generate a Python program for the task, and then queries it to return the corresponding package installation commands (i.e., pip install h2 and pip install hyper-h2).

4) Hallucination Evaluation: In this phase, HFUZZER evaluates the package hallucination based on extracted packages in Section II-B3 and calculates the HS. To achieve fine-grained evaluation, we first classify the extracted packages. Prior studies define hallucinated packages as those recommended by LLMs but either (i) registered after the model's knowledge cutoff date or (ii) not present in the appropriate package repository [9]. Since HFUZZER aims not to test whether LLMs can infer unknown packages but rather to find hallucinations within existing knowledge, it adopts the above definition and further classifies hallucinated packages into two types: nonexistent packages and otherlanguage packages. Additionally, we observe a common scenario in which LLMs mistakenly treat standard libraries (e.g., json) as packages requiring installation. Although these libraries are not found in repositories, they do not strictly qualify as hallucinated packages. Therefore, we exclude them from hallucinated packages. In summary, we classify the packages recommended by the model into four types: "stdPackage", "existPackage", "otherLanguagePackage", and "nonExistentPackage". To formalize these types, let P_{std} be the set of standard libraries, P_{exist} be the set of packages registered in the appropriate package repository before the model's knowledge cutoff date, and P_{lib} be the set of packages in Libraries.io [19]. We define these types as follows:

- Package p is classified as a "stdPackage" if $p \in P_{\text{std}}$.
- Package p is classified as an "existPackage" if $p \in P_{\text{exist}}$.
- Package p is classified as an "otherLanguagePackage" if $p \notin (P_{\text{std}} \cup P_{\text{exist}}) \wedge p \in P_{\text{lib}}$.
- Package p is classified as a "nonExistentPackage" if $p \notin (P_{\text{std}} \cup P_{\text{exist}} \cup P_{\text{lib}})$.

Libraries.io is a cross-language package index that aggregates repositories from multiple programming ecosystems. Following prior studies [10], [9], we use it to verify whether a package originates from a different language.

Based on the above definition, HFUZZER classifies packages recommended by the model and calculates the HS to measure the package hallucination triggering on the target model. Let N_{package} be the total number of packages recommended by the target model, N_{non} be the number of packages classified as "nonExistentPackage", and N_{other} be the number of packages classified as "otherLanguagePackage". The HS is defined as:

$$HS = \frac{\alpha \cdot N_{\text{non}}}{N_{\text{package}}} + \frac{\beta \cdot N_{\text{other}}}{N_{\text{package}}} \tag{1}$$

where $\alpha = 1$ and $\beta = 0.5$[1]. The impact of using nonexistent packages is more serious because it means that the model constructs packages that do not exist, whereas using otherlanguage packages may be due to the model confusing the language of the package. Thus, HFUZZER sets a larger constant to α.

Example B4. In Figure 1, LLM_{target} recommends two packages, which are classified as an "existPackage" and a "nonExistentPackage". Hence, the HS is 0.5.

5) Power Adjustment: To increase the diversity of coding tasks, HFUZZER adjusts the power of phrases based on the HS. Let $Power_o$ be the original power of the phrase, N_{new} be the number of hallucinated packages found for the first time in the fuzzing loop, and N_{old} be the number of hallucinated

[1]All parameters are determined through an initial experiment.

packages found in the previous rounds. The adjusted power of the phrase $Power$ is defined as follows:

$$Power = \begin{cases} Power_o \cdot \left(\frac{k_1 \cdot HS \cdot N_{\text{new}}}{N_{\text{old}}+N_{\text{new}}} + k_2\right) & , \text{ if } N_{\text{old}} + N_{\text{new}} > 0 \\ Power_o \cdot k_2 & , \text{ elif } N_{\text{package}} > 0 \\ Power_o \cdot k_3 & , \text{ otherwise} \end{cases} \quad (2)$$

where $k_1 = 0.15$, $k_2 = 0.8$, and $k_3 = 0.6$. HFUZZER reduces the power of phrases corresponding to the task that do not recommend packages by a factor k_3, while the power of phrases that do not recommend hallucinated packages is decreased by a factor k_2, thereby lowering their chance of reselection. For phrases where the task triggers package hallucination, the power reduction is determined by the HS and the ratio $\frac{N_{\text{new}}}{N_{\text{old}}+N_{\text{new}}}$. By incorporating HS, HFUZZER fine-tunes power based on the specific response of the target model. HFUZZER also assigns relatively higher power to phrases that find new hallucinated packages by $\frac{N_{\text{new}}}{N_{\text{old}}+N_{\text{new}}}$. Intuitively, similar to seeds for generating new coverage in fuzzing, these phrases are more likely to generate coding tasks that can trigger package hallucination in the model.

Example B5. Based on the HS (0.5), assuming the old power is 1 and the hallucinated package is the first appearance, the updated power is 0.875.

6) Seed Pool Expansion: Traditional fuzzers generate more inputs through mutation. In contrast, HFUZZER leverages LLMs to generate new coding tasks based on phrase compositions. However, the phrases in the initial seed pool come from the input of Section II-A, and their compositions are limited. As the number of running rounds increases, the probability of repeated compositions will also increase, thereby reducing the diversity of generated tasks. To overcome this, HFUZZER instructs LLM to extract phrase compositions from coding tasks. Additionally, as in Section II-A, HFUZZER assigns initial power to newly extracted phrases. Let $Power_{\text{initial}}$ be the initial power used in Section II-A. The power of newly extracted phrases $Power_{\text{new}}$ is defined as follow:

$$Power_{\text{new}} = Power_{\text{initial}} \cdot \left(k + \frac{(1-k) \cdot N_{\text{new}}}{N_{\text{new}}+N_{\text{old}}}\right) \quad (3)$$

where $k = 0.6$. Unlike $Power$ in Formula 2, $Power_{\text{new}}$ places more emphasis on finding new hallucinated packages. Therefore, we enlarge the impact of the proportion of using new hallucination packages on power. After preliminary experiments, we set k as 0.6.

Example B6. In Figure 1, HFUZZER extracts new phrase compositions (i.e., GET and POST requests, respond, and easy to set up and run) from the task. Their power is 1.0.

At the end of each round, HFUZZER records the generated coding task, the output of the target model, and the hallucination evaluation results, which can be used to reproduce the package hallucinations found during the test.

III. EVALUATION

In this section, we study the following research questions:

RQ1: How effective is HFUZZER in testing LLMs for package hallucinations? This RQ studies HFUZZER's ability to trigger package hallucination on different models, and evaluates whether it is more effective in testing LLMs for package hallucination compared with GPTFUZZER-A.

RQ2: Whether the tasks generated by HFUZZER more diverse? This RQ studies the diversity of tasks generated by HFUZZER and GPTFUZZER-A.

RQ3: Whether each module of HFUZZER contributes to its performance? This RQ explores the impact of different modules of HFUZZER on its performance.

A. Experimental Setup

Implementation. In our implementation, we access the model through the API provided by the OpenAI library and obtain package information through the APIs of the package repository and libraries.io [19].

Baseline. To our knowledge, no study has focused on testing LLMs for package hallucinations. Existing studies mainly concentrate on empirical studies and mitigating related hallucinations. Note that HFUZZER is intended to test LLMs for hallucinations, not to detect whether the model output contains hallucinations. The study closest to ours is Drowzee [16], which tests LLMs for fact-conflicting hallucinations in NLG through metamorphic testing. However, it assumes a unique answer and cannot be applied to package hallucinations. Therefore, we compare HFUZZER with GPTFUZZER-A, which is adapted from GPTFuzzer [20]. We modify the mutation operators and the initial seeds of GPTFuzzer, and choose a random strategy for seed selection. The mutation operators are adapted to generate new coding tasks instead of new templates. The initial seeds, originally jailbreak templates, are replaced with coding tasks derived from package descriptions to ensure input consistency between HFUZZER and GPTFUZZER-A.

Language Selection. We choose Python as the target language to evaluate HFUZZER. Python is a popular language, ranked number one on the TIOBE index [22], with a well-developed ecosystem and widespread use in related studies [9], [10], [5].

Model Selection. To comprehensively evaluate HFUZZER on LLMs with different training data and architectures, we select multiple open-source popular models and closed-source models. The detailed information is shown in Table I.

TABLE I
THE DETAILS OF SELECTED MODELS

Model	Size	Code Model	Open Weights	Full Name
Meta-Llama-3	8B	No	Yes	Meta-Llama-3-8B-Instruct [23]
Qwen2.5-Coder	7B	Yes	Yes	Qwen2.5-Coder-7B-Instruct [24]
DeepSeek-Coder	6.7B	Yes	Yes	DeepSeek-Coder-Instruct 6.7B [25]
Meta-Llama-3.1	8B	No	Yes	Meta-Llama-3.1-8B-Instruct [26]
Mistral-v0.3	7B	No	Yes	Mistral-7B-Instruct-v0.3 [27]
Meta-Llama-3.3	70B	No	Yes	Meta-Llama-3.3-70B-Instruct [28]
DeepSeek-V3	671B	No	Yes	DeepSeek-V3 [29]
GPT-4o mini	-	No	No	GPT-4o mini [30]
GPT-4o	-	No	No	GPT-4o [31]

Metric. We evaluate the effectiveness of HFUZZER using the Package Hallucination Rate (PHR) and the number of unique hallucinated packages (RQ1 and RQ3). PHR is the proportion of model responses containing hallucinated packages over the total number of responses [9], and is used to assess whether the coding tasks generated by HFUZZER can trigger package hallucinations. The number of unique hallucinated packages is analogous to a widely used metric in fuzzing (i.e., unique bugs)

TABLE II
RQ1: UNIQUE HALLUCINATED PACKAGES RESULTS

Tester \ Target		Meta-Llama-3		Qwen2.5-Coder		DeepSeek-Coder		Meta-Llama-3.1		Mistral-v0.3		Meta-Llama-3.3		DeepSeek-V3		GPT-4o mini		GPT-4o		Avg. (same Tester)
		P_{uniq}	R_{inc}	P_{uniq}	R_{inc}	P_{uniq}	R_{inc}	P_{uniq}	R_{inc}	P_{uniq}	R_{inc}	P_{uniq}	R_{inc}	P_{uniq}	R_{inc}	P_{uniq}	R_{inc}	P_{uniq}	R_{inc}	
Meta-Llama-3	G-A	23	↑2.22	5	↑1.20	14	↑1.71	44	↑1.48	54	↑1.76	3	↑2.00	0	↑ -	5	↑1.40	0	↑ -	1.68
	H	51		6		24		65		95		6		3		7		1		
Qwen2.5-Coder	G-A	29	↑1.66	1	↑6.00	19	↑2.11	47	↑1.53	55	↑1.56	2	↑3.00	1	↑3.00	1	↑4.00	0	-	2.65
	H	48		6		40		72		86		6		3		4		0		
DeepSeek-Coder	G-A	13	↑2.77	1	↑8.00	30	↑3.70	27	↑4.22	40	↑4.60	2	↑7.50	0	↑ -	0	↑ -	0	↑ -	5.13
	H	36		8		111		114		184		15		3		3		1		
Meta-Llama-3.1	G-A	35	↑1.17	5	↑1.80	38	↑1.13	40	↑2.03	66	↑2.27	3	↑3.00	3	↑2.33	2	↑2.50	1	1.00	1.91
	H	41		9		43		81		150		9		7		5		1		
Mistral-v0.3	G-A	49	↑1.04	1	1.00	17	↑2.94	54	↑1.39	100	↑1.14	3	↑3.67	1	↑5.00	4	↑1.25	0	-	2.05
	H	51		1		50		75		114		11		5		5		0		
Meta-Llama-3.3	G-A	45	↑1.18	4	↑1.25	20	↑2.35	53	↑1.40	68	↑1.72	7	↑1.71	1	↑9.00	2	↑5.00	0	↑ -	2.95
	H	53		5		47		74		117		12		9		10		5		
DeepSeek-V3	G-A	53	↑1.13	5	↑1.20	11	↑3.09	61	↑1.13	46	↑2.43	6	↑2.17	1	↑3.00	1	↑4.00	2	↑2.00	2.24
	H	60		6		34		69		112		13		3		4		4		
GPT-4o mini	G-A	24	↑1.79	3	↑1.67	18	↑3.28	39	↑2.05	52	↑2.02	4	↑ 3.75	0	↑ -	0	↑ -	0	-	2.22
	H	43		5		59		80		105		15		5		3		0		
GPT-4o	G-A	38	↑1.05	5	↑2.40	11	↑3.36	45	↑1.20	64	↑ 1.50	2	↑7.00	1	↑7.00	0	↑ -	2	↑2.00	3.19
	H	40		12		37		54		96		14		7		4		4		
Avg. (same Target)		1.56		2.72		2.63		1.82		2.11		3.76		4.89		3.03		1.33		2.60 (all)

G-A is GPTFUZZER-A and H is HFUZZER. P_{uniq} is the number of unique hallucinated packages found by HFUZZER or GPTFUZZER-A, and R_{inc} indicates the improvement rate of HFUZZER compared to GPTFUZZER-A.

TABLE III
RQ1: MULTIPLE RUNS RESULTS

Tester \ Target		Meta-Llama-3.1			Mistral-v0.3			GPT-4o mini		
		μP_{uniq}	μR_{inc}	P	μP_{uniq}	μR_{inc}	P	μP_{uniq}	μR_{inc}	P
Meta-Llama-3	G-A	39.33	↑1.62	0.00	54.33	↑1.85	0.00	2.67	↑1.75	0.15
	H	63.67			100.33			4.67		
Qwen2.5-Coder	G-A	44.67	↑1.57	0.04	59.33	↑1.69	0.01	0.33	↑10.00	0.06
	H	70.33			100.00			3.33		
deepseek-coder	G-A	27.67	↑4.23	0.02	36.00	↑4.17	0.01	0.00	↑ -	0.03
	H	117.00			150.00			4.00		
Meta-Llama-3.1	G-A	33.67	↑2.28	0.01	69.00	↑1.91	0.01	2.00	↑2.67	0.02
	H	76.67			131.67			5.33		
Mistral-v0.3	G-A	42.00	↑2.06	0.01	50.33	↑2.43	0.01	3.00	↑1.44	0.16
	H	86.67			122.33			4.33		
Meta-Llama-3.3	G-A	53.67	↑1.30	0.01	74.67	↑1.62	0.00	2.33	↑3.71	0.01
	H	70.00			121.00			8.67		
DeepSeek-V3	G-A	43.67	↑1.60	0.05	60.00	↑1.76	0.01	0.33	↑14.00	0.00
	H	70.00			105.33			4.67		
GPT-4o mini	G-A	40.67	↑1.77	0.00	58.33	↑1.86	0.00	0.67	↑5.00	0.04
	H	72.00			108.67			3.33		
GPT-4o	G-A	48.33	↑1.14	0.03	54.33	↑2.09	0.01	0.00	↑ -	0.09
	H	55.00			113.67			4.67		
Avg.(Same Target)		1.95			2.15			5.51		
μCV (H/G-A)		0.10/0.17			0.11/0.14			0.39/0.72		

μP_{uniq} and μR_{inc} are the means of multiple runs results. P is the value of Welch's t-test. μCV is the mean of the coefficient of variation.

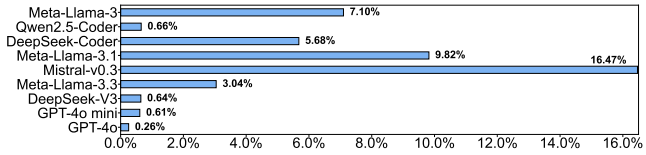

Fig. 3. RQ1: Average PHR of Different Target Models

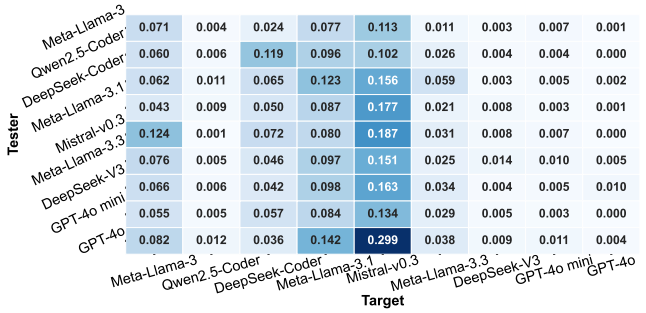

Fig. 4. RQ1: Heatmap of PHR with Different Models

and is used to evaluate HFUZZER's ability to test LLMs for package hallucinations. We use the Diversity Index to assess task diversity (RQ2 and RQ3). The Diversity Index is defined as the total of clusters and noise points.

Environment. Our experiments are conducted on a server with four NVIDIA A800 GPUs. The server has an Intel Xeon Platinum 8358P CPU with 32 cores and 2TB of memory. The version of vLLM [32] used is v0.6.1.

B. RQ1: The Capability in Testing LLMs

To answer RQ1, we calculate the number of unique hallucinated packages found by HFUZZER and GPTFUZZER-A, as well as the PHR for different target models. We select the top 100 Python packages from libraries.io [19] according to the "SourceRank" and collect their information. Each run is limited to 1,000 rounds. To ensure comprehensiveness, we use nine models as tester and target models, forming 81 model combinations. The temperature of the tester models is fixed at 0 for determinism, whereas the target model's temperature is set to 0.7 to balance creativity and stability [33]. The maximum token limit is set to 3,000.

Table II reports the results of unique hallucinated packages. Overall, HFUZZER outperforms GPTFUZZER-A across most model combinations, with an average number of unique hallucinated packages increasing by 2.60×. We combine Table II and Figure 3 to analyze the results on different target models. Figure 3 shows the average PHR for each target model, computed by averaging the PHR under different tester models. HFUZZER triggers package hallucinations across all target models. GPT-4o, GPT-4o mini, DeepSeek-V3, and Qwen2.5-Coder exhibit relatively low PHRs (0.26–0.66%), while Mistral-v0.3 shows the highest (16.47%). The PHR for DeepSeek-Coder, Meta-Llama-3, and Meta-Llama-3.1 are 5.68%, 7.10%, and 9.82%. Compared with GPTFUZZER-A, HFUZZER finds more unique hallucinated packages across all target models, demonstrating its applicability. The largest improvement is achieved when the target model is DeepSeek-V3 (4.89x). Additionally, when GPT-4o mini and GPT-4o are used as target models, HFUZZER outperforms GPTFUZZER-A, highlighting its practical value. To analyze the impact of different tester models, we analyze the results using different tester models, as shown in Table II and Figure 4. For the number of unique hallucinated packages, HFUZZER outperforms GPTFUZZER-A under all tester models, with the largest improvement (5.13×) achieved when the tester is

DeepSeek-Coder. Moreover, HFUZZER triggers hallucinations in most model combinations, and no tester model consistently outperforms the others. This suggests that the performance of HFUZZER does not strongly depend on the specific tester model, and it remains effective even with smaller-scale LLMs.

To evaluate stability, we follow MetaQA [18], select GPT-4o mini, Meta-Llama-3.1, and Mistral-v0.3 as target models. For each target, we use all tester models and run HFUZZER and GPTFUZZER-A three times. Following prior studies [34], [35], we use Welch's t-test to assess statistical significance and report the coefficients of variation (CV) [36]. As shown in Table III, HFUZZER yields an average 3.02× improvement over GPTFUZZER-A. This improvement is statistically significant ($p<0.05$) for most model combinations, and HFUZZER exhibits consistently lower CV, indicating greater stability. For some combinations, improvements are not significant due to the low hallucination rates of the target models. The low hallucination rates result in limited differences in means relative to variances, leading to smaller effect sizes in the Welch's t-test and larger CV compared to other target models.

Further analysis shows a common phenomenon across most models: the tendency to recommend packages composed of technical terms mentioned in the task. This phenomenon is particularly pronounced when the hallucinated package exists in another programming language. For instance, DeepSeek-Coder tends to generate the hallucinated package `jsonwebtoken` when responding to tasks involving "JSON Web Tokens", and both Meta-Llama-3.1 and Mistral-v0.3 tend to generate the hallucinated package `apache-arrow` when responding to tasks involving "Apache Arrow". We also investigate the distribution of hallucinated packages across different models. Our analysis reveals that there are 190 non-existent packages and 176 other-language packages recommended by multiple models. Among them, the package `pkg_resources` is recommended by all models, and 17 packages are recommended by more than half of the models. This indicates that the same hallucinated packages exist across different models. Considering the architectural differences between the models, we believe that most of these hallucinated packages likely originate from the training data.

Qwen2.5-Coder shows a low PHR in our evaluation, which is inconsistent with the results of other studies [9], [10]. We find that Qwen2.5-Coder often refuses to generate code even when the same task can be responded to by other models, which limits the effectiveness of both HFUZZER and GPTFUZZER-A. To further investigate this issue, we query other models of Qwen using the same tasks that triggered refusals in Qwen2.5-Coder. Several models in the Qwen2.5-code series also refuse to respond, whereas the Qwen2.5 series and newer Qwen3/Qwen3-code models respond normally. This behavior is similar to the over-refusal phenomenon reported in the previous study [37]. Since it appears confined to specific models, we consider our results sufficient to validate HFUZZER's effectiveness. Additionally, as over-refusal is beyond the scope of this paper, we leave further investigation for future work. We also find that there are more unique hallucinated packages in Llama 3.1 compared to Llama 3. Further analysis of these hallucinated packages reveals that those beginning with `google-cloud` are widely recommended, especially when the code contains a statement `from google.protobuf import xx`. We suspect that this issue is related to the training data. Additionally, even after removing these specific packages, Llama 3.1 still recommends more hallucinated packages compared to Llama 3. These results show an unexpected observation from specific cases: although Llama 3.1 generally outperforms Llama 3 in tasks such as code generation, this does not necessarily correspond to a lower hallucination rate. A similar observation is also found by Spracklen et al. [10] (CodeLlama 34B and CodeLlama 13B). Note that whether this observation holds for more models requires broader experiments, which we plan to investigate further in the future.

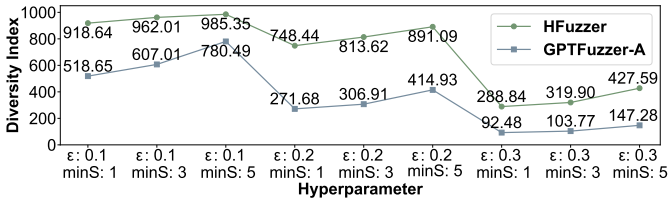

Fig. 5. RQ2: Average Diversity index of tasks generated under different DBSCAN parameter settings ($\varepsilon \in 0.1, 0.2, 0.3$, $minS \in 1, 3, 5$).

Summary for RQ1: The tasks generated by HFUZZER trigger package hallucinations on multiple models. Compared with GPTFUZZER-A, HFUZZER finds more unique hallucinated packages, with the average number increasing by 2.60×. Further analysis reveals that: (1) most models recommend hallucinated packages consisting of technical terms; (2) some hallucinated packages appear across different models.

C. RQ2: The Diversity of Generated Tasks

To answer RQ2, we employ the model text-embedding-3-small [38] to transform the tasks generated in Section III-B into high-dimensional vectors, followed by clustering with the DBSCAN algorithm [39]. We set the neighborhood radius parameter (ε) to $0.1, 0.2, 0.3$ and the minimum samples parameter (*minS*) to $1, 3, 5$ to explore the impact of different parameter settings. For each setting, we compute the Diversity Index for every model combination and report the average Diversity Index across all combinations.

Across all parameter settings, the tasks generated by HFUZZER consistently have higher Diversity Indices than those generated by GPTFUZZER-A, with an average improvement of 2.36×, as shown in Figure 5. The Diversity Index decreases as ε increases, because a larger ε merges more tasks into the same cluster. In contrast, increasing *minS* leads to higher Diversity Index, as originally small clusters are split and more tasks are treated as noise points. Further analysis shows that across all model combinations, tasks generated by HFUZZER exhibit greater diversity compared to those generated by GPTFUZZER-A. For different tester models, the average CV of the Diversity Index is 0.15 for HFUZZER

TABLE IV
RQ3: UNIQUE HALLUCINATED PACKAGES RESULTS OF ABLATION

Tester Model	Target Model	HFUZZER	W/O Expansion		W/O Power		W/O Phrase		W/O All	
		P_{uniq}	P_{uniq}	R_{inc}	P_{uniq}	R_{inc}	P_{uniq}	R_{inc}	P_{uniq}	R_{inc}
Meta-Llama-3	Meta-Llama-3.1	65	48	↑1.35	44	↑1.48	43	↑1.51	1	↑65.00
	Mistral-v0.3	95	80	↑1.19	31	↑3.06	58	↑1.64	1	↑95.00
Qwen2.5-Coder	Meta-Llama-3.1	72	74	↑1.26	56	↑1.29	32	↑2.25	0	↑ -
	Mistral-v0.3	86	86	1.00	59	↑1.46	48	↑1.79	6	↑14.33
DeepSeek-Coder	Meta-Llama-3.1	114	74	↑1.54	76	↑1.50	31	↑3.68	1	↑114.00
	Mistral-v0.3	184	87	↑2.11	143	↑1.29	56	↑3.29	1	↑184.00
Meta-Llama-3.1	Meta-Llama-3.1	81	66	↑1.23	59	↑1.37	44	↑1.84	8	↑10.13
	Mistral-v0.3	150	119	↑1.26	134	↑1.12	85	↑1.76	0	↑ -
Mistral-v0.3	Meta-Llama-3.1	75	73	↑1.03	69	↑1.09	42	↑1.79	0	↑ -
	Mistral-v0.3	114	88	↑1.30	98	↑1.16	60	↑1.90	2	↑57.00
Meta-Llama-3.3	Meta-Llama-3.1	74	63	↑1.17	45	↑1.64	36	↑2.06	1	↑74.00
	Mistral-v0.3	117	113	↑1.04	77	↑1.52	39	↑3.00	1	↑117.00
DeepSeek-V3	Meta-Llama-3.1	69	57	↑1.21	60	↑1.15	59	↑1.17	14	↑4.93
	Mistral-v0.3	112	94	↑1.19	97	↑1.15	52	↑2.15	11	↑10.18
GPT-4o mini	Meta-Llama-3.1	80	56	↑1.43	52	↑1.54	36	↑2.22	1	↑80.00
	Mistral-v0.3	105	86	↑1.22	100	↑1.05	53	↑1.98	15	↑7.00
GPT-4o	Meta-Llama-3.1	54	53	↑1.02	53	↑1.02	42	↑1.29	5	↑10.80
	Mistral-v0.3	96	77	↑1.25	91	↑1.05	57	↑1.68	14	↑6.86
Avg. R_{DI}		-	1.26		1.25		4.40		116.46	
Avg. R_{inc}		-	1.27		1.39		2.06		56.68	

Avg. R_{DI} is the mean improvement (HFUZZER/variant) across all model combinations and DBSCAN parameter settings.

and 0.29 for GPTFUZZER-A. When $\varepsilon = 0.1$, $\varepsilon = 0.2$, and $\varepsilon = 0.3$, the corresponding CVs are (0.03/0.16), (0.11/0.24), and (0.32/0.47), respectively. For different target models, the CVs are always around 0.1. For multiple runs in RQ1, the tasks generated by HFUZZER exhibit a significant improvement in diversity across all parameter settings, with an average Diversity Index increase of 2.46x ($p < 0.05$ and $CV \approx 0.01$).

Summary for RQ2: HFUZZER consistently yields more diverse coding tasks than GPTFUZZER-A.

D. RQ3: The Impact of Different Modules

To evaluate the impact of different modules, we design four variants with different modules removed:

- *w/o Expansion*: removes the Seed Pool Expansion module introduced in Section II-B6;
- *w/o Power*: removes the Power module and uses random selection;
- *w/o Phrase*: removes the Phrase module and directly uses package descriptions;
- *w/o All*: removes all modules and relies only on the LLM to generate coding tasks.

We compare HFUZZER with these variants to analyze how each model improves HFUZZER. We use two target models (Mate-Llama-3.1 and Mistral-v0.3) that recommended the largest number of unique hallucinated packages in RQ1 to clearly reveal the impact, and use all tester models. Other settings are consistent with RQ1.

The results are shown in Table IV. Each module enhances the performance of HFUZZER. The Phrase module (*w/o Phrase*) has the largest impact, improving the number of unique hallucinated packages by 2.06× and the diversity Index by 4.40×. The Power module (*w/o Power*) and the Seed Pool Expansion module (*w/o Expansion*) improve the number of unique hallucinated packages by 1.39× and 1.27×, respectively, and the diversity Index by 1.25× and 1.26×. *w/o All* performs poorly, as relying solely on the LLM often leads to repetitive coding tasks, highlighting the importance of guiding the LLM.

Summary for RQ3: For the number of unique hallucinated packages, each module brings improvement of 1.27x, 1.39x, and 2.06x. For the Diversity Index, each module brings improvements of 1.26x, 1.25x, and 4.40x.

IV. CASE STUDY

We use HFUZZER to test the model GPT-4o [40], which is widely applied across various fields, observe its results, and conduct an in-depth analysis to inspire subsequent studies. For large-scale evaluation, we use the information of the top 1,000 packages as input and run 10,000 rounds. We use GPT-4o mini as the tester model to reduce costs.

We manually inspect hallucinated packages and conduct a detailed analysis. HFUZZER finds 46 unique hallucinated packages, 11 of which are other-language packages. Examining the intermediate results reveals two types of hallucinated packages: **code error** and **package error**. **Code error** occurs when the generated code contains incorrect import statements. As shown in Figure 6 (A), the model is prompted to develop a Python application for handling multipart data uploads from a web interface. The generated code includes `from flask_livereload import LiveReload`, attempting to import a non-existent package. By analyzing the code, we find that the intended package is `livereload`, but the model produces an incorrect import statement. **Package error** occurs when the import statement is correct, but the model returns hallucinated packages in the installation command. Figure 6 (B) illustrates this with an example in which the model is required to apply wavelet transforms to images for contour computation. In the generated code, the model generates the correct import statement `import pywt` to import the package `PyWavelets`. However, when providing the installation command, the model incorrectly suggests using `pip install pywt`, which indicates that the model cannot correctly match the installation command with the import statement. The preliminary examination is performed by one author. To

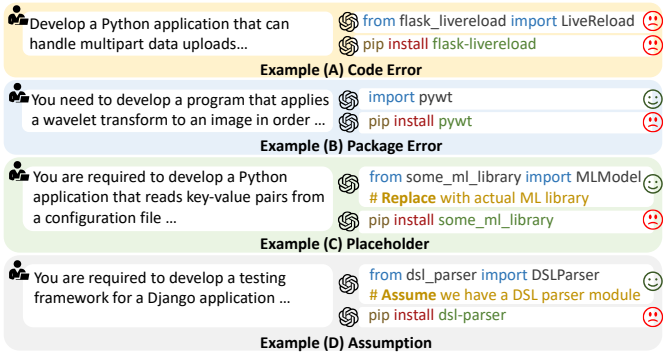

Fig. 6. Examples of Case Study

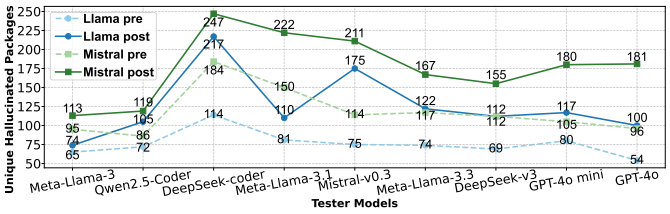

Fig. 7. Results with Different HFUZZER Input

classify all hallucinated packages, two authors independently classify hallucinated packages as either code errors or package errors. The classification is consistent due to the clear criteria: after installing the correct package, if the corresponding code can parse this package, it is classified as a package error; otherwise, it is a code error. Our classification results show that 34 hallucinated packages belong to package error, while only 12 belong to code error. This suggests a potential hypothesis: current LLMs are more prone to package hallucinations when assisting with environment configuration. Existing studies [41], [9] use regular expressions to extract packages, overlooking potential hallucinations that can arise from this process. Due to the limitations of regular expressions (e.g., the inability to handle Python aliases), automated environment configuration often relies on LLM implementation in real-world scenarios [42]. Whether this hypothesis holds across different models requires further investigation, but it is noteworthy as it highlights a potential scenario influenced by package hallucinations.

Inspired by Latendresse et al. [41], we examine hallucinated packages to identify placeholders. Following the definition of Latendresse et al. [41], two authors independently review all hallucinated packages and resolve disagreements through discussion. We find five placeholders: `ammonia_library`, `some_ml_library`, `some_multimedia_sdk`, `some-security-sdk`, and `some-cloud-language-api`. For the first three placeholders, the model explicitly includes comments in the generated code to indicate their placeholder status. To further analyze the impact of such comments, we ask two authors to independently review the code snippets containing comments. Their findings are consistent: all hallucinated packages with comments are marked as placeholders or assumptions. In total, 12 hallucinated packages contain similar comments, including both placeholders and assumptions (e.g., "Assuming the library is named this"). However, when queried for installation commands, the model ignores these comments and still returns the corresponding installation commands, which is the input-conflicting hallucination. In Figure 6 (C), the model generates code that includes a placeholder `some_ml_library` and a comment clarifying its placeholder state. Despite this, the model still returns an installation command containing this placeholder. Similarly, in Figure 6 (D), the model adds a comment assuming the existence of `dsl_parser`. However, the model ignores the comment when returning an installation command and incorrectly returns `pip install dsl_parser`. These findings indicate that the model may ignore code comments when analyzing the required packages, which motivates further investigation into the influence of code comments on LLM responses in code-related tasks.

> **Findings:** (1) For GPT-4o, Package hallucinations occur not only during code generation, but also when assisting with environment configuration, even if the correct code has been provided. (2) The model may ignore comments when analyzing the required packages in the code.

V. DISCUSSION

A. The Effect of Potential Data Contamination

To investigate the effect of potential data contamination, according to the "SourceRank", we select the first 100 Python packages released after the training cutoff dates of the chosen models (post-cutoff packages). We run 1000 rounds using all tester models and two target models (Meta-Llama-3.1 and Mistral-v0.3). Since the top 100 packages used in Section III-B are released before the cutoff dates (pre-cutoff packages), we compare the post-cutoff results with those in Section III-B.

The results are shown in Figure 7. HFUZZER finds more hallucinated packages when using post-cutoff packages. Compared with pre-cutoff packages, post-cutoff ones yield slightly higher diversity, with the Diversity Index increasing by 0.3%, 7.6%, and 24.8% under DBSCAN parameters $\varepsilon = 0.1, 0.2$, and 0.3, respectively. Further analysis reveals that tasks derived from these packages contain elements unfamiliar to the models (e.g., model context protocol), and the seed pool includes more phrases. When the information sought extends beyond the model's training data, LLM may fail to provide accurate answers [43], resulting in more hallucinated packages. Moreover, information outside the training data may introduce greater randomness in responses, thus affecting the diversity of the generated tasks. Considering that LLMs are rarely applied to tasks involving unknown content, in Section III-B, we use pre-cutoff packages to evaluate HFUZZER.

B. The Effect of Parameter Settings

To investigate the impact of HFUZZER's parameters, we run HFUZZER with different parameter settings on the two model combinations that find the largest number of unique hallucinated packages in RQ1 (i.e., DeepSeek-coder+Mistral-v0.3 and Meta-Llama-3.1+Mistral-v0.3). For score parameters (α/β), we test two variants to adjust the impact of

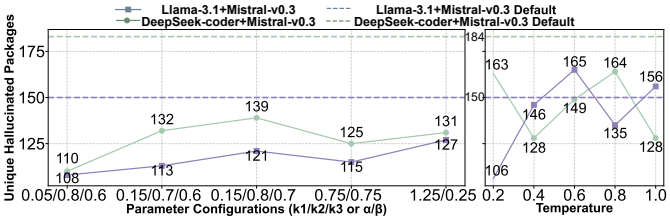

Fig. 8. The results of different parameters.

"nonexistentPackage": 1.25/0.25 and 0.75/0.75. For power parameters ($k_1/k_2/k_3$), we test three variants: 0.05/0.8/0.6 (reduces the impact of discovering new hallucinated packages); 0.15/0.7/0.6 (reduces the impact of recommending packages); and 0.15/0.8/0.7 (reduces the impact of selection). Additionally, we test five temperature variants to analyze their impacts. Other settings are consistent with RQ1. The Default is the parameter setting used in Section III.

Figure 8 shows that, for the number of unique hallucinated packages, the parameter variants find fewer hallucinated packages than the Default. In contrast, the effect of temperature is not significant, with no clear trend observed. The parameter variants generate less diverse tasks, with Diversity Indices between 85% and 96% of the Default, averaged over all clustering parameters. Temperature also influences diversity: at 0.2, the Diversity Index decreases to 89%, whereas at 0.4–1.0, it increases slightly (by 1.02%–1.26%). Overall, parameter and temperature settings affect the performance of HFUZZER, but their impact remains limited, and the results consistently outperform those of GPTFUZZER-A.

C. The Impact of Package Hallucinations

To further assess the impact of package hallucinations on real-world developers, we manually investigate instances of GPT-4o's hallucinated package usage on GitHub [44] and find that some hallucination packages have been used in the repository. For instance, we identify that the hallucinated package "data-analysis-toolkit" appears in several open-source repositories, such as research-manual[2]. Similarly, Lanyado conducts a case study on hallucinated packages [45], finding that a hallucinated package is downloaded over 30,000 more times than a randomly named package within three months, and is even incorporated into repositories of some commercial companies. These findings illustrate that developers can be inadvertently exposed to hallucinated packages during development, potentially increasing their vulnerability to malicious attacks. Moreover, recent advances in LLM-based software development agents, such as Devin [14], which support end-to-end application development and automatic deployment, further exacerbate this risk, as these agents may unknowingly install hallucinated packages in practice.

D. Potential downstream tasks

Package Hallucination Mitigation: In existing studies, Spracklen et al. [10] have proposed their insights to mitigate package hallucination. Spracklen et al. [10] make a preliminary attempt to mitigate package hallucination using several popular techniques. Their experimental results show that Retrieval Augmented Generation (RAG), Self-Detected Feedback, and Fine-tuning all help reduce the model PHR, with Fine-tuning proving to be the most effective. On the DeepSeek Coder 6B, PHR is reduced by 83% through fine-tuning. HFUZZER can provide sufficient data for fine-tuning, enabling more effective mitigation and future evaluation.

Model Performance Improvement: According to the research of Krishna et al. [9], PHR is negatively correlated with the performance of the model on the code benchmark. Therefore, reducing inherent model package hallucinations through techniques such as fine-tuning and model editing is a promising direction for improving model performance in the field of code in the future. Our work provides sufficient data support and evaluation methods for such follow-up studies.

E. Testing LLM VS. Detecting LLM Hallucinations

Existing studies [46], [47], [48], [18], [49] are proposed to detect hallucinations. Other studies [50], [51], [52], [53], [54], [55], [56] can also detect package hallucinations as a side effect. In contrast, HFUZZER is a testing approach to test LLMs, representing a distinct yet complementary research direction. The core differences are as follows:

Technique Difference: Whereas existing studies analyze LLM outputs, HFUZZER generates diverse and logical inputs.

Goal Difference: The goal of related studies is to detect whether LLm outputs are hallucinations, while the goal of HFUZZER is to test LLMs.

F. Threats to Validity

Limited of Language. In this paper, we primarily evaluate HFUZZER on Python. However, the framework is designed to be language-agnostic. We focus on Python because it is widely used and frequently studied in related work [9], [10]. In future work, we plan to extend our evaluation to multiple programming languages.

Limited Accuracy of Regular Extraction. We rely on regular expressions to extract information from the model's output. Due to the inherent instability of the LLM's output, the extracted content may not always align with expectations. To improve the stability of the LLM's output format, we employ one-shot prompting to guide the output format and implement validation checks on the extracted results.

Limited LLM Sampling Strategy. The LLM sampling strategy may affect the performance of LLM-based methods. Investigating the impact of sampling strategies on HFUZZER helps to assess its robustness. Therefore, we examine the effect of the temperature of the tester model in Section V-B. For cost considerations, the study is conducted on two model combinations. We plan to further extend it in future work.

VI. RELATED WORK

A. Fuzzing

Fuzzing is one of the most popular testing techniques, which can find weaknesses in a program [57], [58]. Lee et al. [59]

[2]https://github.com/iHuman-Lab/research-manual

group seeds by syntax and semantic similarity and use a customized Thompson sampling method to choose effective mutation strategies for each group. Liu et al. [60] propose mutating inputs directly in memory and using print functions to regenerate files, improving fuzzing for complex file formats. Yang et al. [61] implement a fully automated API-level fuzzer for automatic differentiation in deep learning libraries. Hough et al. [62] exploit dynamic execution information to identify and exchange similar parts of parameter sequences for parametric fuzzing. Wang et al. [63] use fine-grained semantic alignment techniques to generate semantically correct test inputs for fuzzing browsers. With the widespread use of LLMs, some researchers have attempted to combine LLMs with fuzzing. Xia et al. [64] leverage an LLM to generate and mutate inputs for systems that take programming languages or formal languages as inputs. Eom et al. [65] combine coverage feedback with an LLM-based mutator using reinforcement learning to test JavaScript engines. Deng et al. [66] propose FuzzGPT, which uses LLMs to generate anomalous programs for fuzzing deep learning libraries. In contrast to these works, HFUZZER treats the LLM as the target of testing.

B. LLM Jailbreak

LLM jailbreak refers to exploiting carefully crafted prompts to elicit content that violates service guidelines [67]. To better guide defense strategies, existing studies have conducted extensive testing on LLM jailbreak vulnerabilities through red team attacks [67], [20], [68]. Yao et al. [67] use templates to preserve prompt structure and isolate jailbreak features as constraints, creating an automated framework to test and find jailbreak vulnerabilities in LLMs. Yu et al. [20] exploit manually crafted templates as initial input and mutate them to generate new templates. Additionally, some studies have also made efforts to defend against LLM jailbreak. Zhang et al. [69] adjust the target LLM's hidden representations by enhancing toxic concepts and weakening jailbreak concepts, ensuring the LLM generates safe content. Wang et al. [70] leverage reverse inference on the initial responses of the LLM to reveale the actual intent behind the original prompt. They then re-prompt the LLM to generate responses in a way that mitigates potential jailbreak attacks. Extensive studies have been conducted through red team attacks to test LLMs for jailbreak vulnerabilities and guide defense strategies. However, studies on hallucinations are relatively scarce. HFUZZER fills this gap, laying the foundation for better mitigation of LLM package hallucinations, which is a high-risk hallucination type.

C. LLM Hallucination

Despite recent progress, LLMs still generate hallucinations [9], which can be categorized into input-, context-, and fact-conflicting types [6]. To reduce the impact of hallucinations, researchers conduct studies on detecting and mitigating hallucinations. Jones et al. [71] utilize prefix-tuning on synthetic tasks to optimize the system message, then transfer this message to realistic tasks to reduce hallucination. Chen et al. [46] propose to explore the dense semantic information retained within LLMs' internal states for hallucination detection. Zhang et al. [47] introduce a reference-free, uncertainty-based method to detect hallucinations in LLMs. Quevedo et al.[48] construct two simple classifiers for hallucination detection using four numerical features, using supervised learning. Yang et al. [18] use metamorphic testing to mutate the model response and evaluate whether the mutated content is correct through LLM to detect hallucinations. Note that their mutation method is still based on the assumption of a unique answer.

Given the application of LLMs in code-related tasks, researchers study hallucinations in the code domain [72]. Liu et al. [5] explore hallucinations in code generation and propose a novel classification for code hallucinations. Jain et al. [73] mitigate API hallucinations in low-frequency APIs through documentation augmented generation. Tian et al. [49] propose a hallucination detection algorithm based on execution validation and a code hallucination classification method. Spracklen et al. [10] and Krishna et al. [9] further investigate a special type of code hallucinations, i.e., package hallucination. They define package hallucination as an LLM generates code that either recommends or contains a reference to a package that is not registered in the appropriate package repository or is first registered after the model's knowledge cutoff date. These hallucinations pose security risks, as attackers may register phantom packages with malicious code, which LLMs then recommend to developers [10], [9], [11], [45]. Considering the security risks posed by package hallucination, we propose HFUZZER and apply it to package hallucination. Different from existing methods for hallucination detection and mitigation, HFUZZER focuses on generating tasks to trigger model hallucinations, which is a method similar to red team attacks.

VII. Conclusion and Future Work

In this paper, we present HFUZZER, a novel phrase-based fuzzing framework to test LLMs for package hallucinations. By automatically generating diverse coding tasks based on phrases, HFUZZER extensively tests LLMs for package hallucinations. Through extensive evaluation, we demonstrate that HFUZZER can trigger package hallucination across all selected models, find 2.60x more unique hallucinated packages compared with GPTFUZZER-A, and generate more diverse tasks. We further test the model GPT-4o and find 46 unique hallucinated packages. Our analysis shows that for GPT-4o, package hallucinations not only occur in code generation but also occur when assisting with environment configuration.

In the future, we plan to conduct more extensive testing on more LLMs and more types of languages, and promote the study of LLM package hallucination mitigation through open-sourcing results. Additionally, we hope to further expand our framework on code hallucination.

VIII. Acknowledgments

This research is supported by the National Key R&D Program of China (No. 2024YFB4506400).

REFERENCES

[1] I. Ahmed, A. Aleti, H. Cai, A. Chatzigeorgiou, P. He, X. Hu, M. Pezzè, D. Poshyvanyk, and X. Xia, "Artificial intelligence for software engineering: The journey so far and the road ahead," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 5, May 2025. [Online]. Available: https://doi.org/10.1145/3719006

[2] M. Yang, Y. Chen, Y. Liu, and L. Shi, "Distillseq: A framework for safety alignment testing in large language models using knowledge distillation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 578–589. [Online]. Available: https://doi.org/10.1145/3650212.3680304

[3] M. L. Siddiq, L. Roney, J. Zhang, and J. C. D. S. Santos, "Quality assessment of chatgpt generated code and their use by developers," in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. MSR '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 152–156. [Online]. Available: https://doi.org/10.1145/3643991.3645071

[4] M. L. Siddiq, J. C. da Silva Santos, S. Devareddy, and A. Muller, "Sallm: Security assessment of generated code," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, ser. ASEW '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 54–65. [Online]. Available: https://doi.org/10.1145/3691621.3694934

[5] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, "Exploring and evaluating hallucinations in llm-powered code generation," 2024. [Online]. Available: https://arxiv.org/abs/2404.00971

[6] Y. Zhang, Y. Li, L. Cui, D. Cai, L. Liu, T. Fu, X. Huang, E. Zhao, Y. Zhang, Y. Chen, L. Wang, A. T. Luu, W. Bi, F. Shi, and S. Shi, "Siren's song in the ai ocean: A survey on hallucination in large language models," *Computational Linguistics*, pp. 1–46, 09 2025. [Online]. Available: https://doi.org/10.1162/COLI.a.16

[7] C. Gao, X. Hu, S. Gao, X. Xia, and Z. Jin, "The current challenges of software engineering in the era of large language models," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 5, May 2025. [Online]. Available: https://doi.org/10.1145/3712005

[8] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Comput. Surv.*, vol. 55, no. 12, Mar. 2023. [Online]. Available: https://doi.org/10.1145/3571730

[9] A. Krishna, E. Galinkin, L. Derczynski, and J. Martin, "Importing phantoms: Measuring llm package hallucination vulnerabilities," 2025. [Online]. Available: https://arxiv.org/abs/2501.19012

[10] J. Spracklen, R. Wijewickrama, A. H. M. N. Sakib, A. Maiti, B. Viswanath, and M. Jadliwala, "We have a package for you! a comprehensive analysis of package hallucinations by code generating llms," 2025. [Online]. Available: https://arxiv.org/abs/2406.10279

[11] B. Lanyado, "Can you trust chatgpt's package recommendations?" https://vulcan.io/blog/ai-hallucinations-package-risk/, 2023, last accessed Mar. 2025.

[12] S. Neupane, G. Holmes, E. Wyss, D. Davidson, and L. De Carli, "Beyond typosquatting: an in-depth look at package confusion," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23. USA: USENIX Association, 2023.

[13] "Pypi," https://pypi.org/, 2000, last accessed Mar. 2025.

[14] "Devin," https://devin.ai/, 2024, last accessed Mar. 2025.

[15] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, Jaskirat *et al.*, "Openhands: An open platform for AI software developers as generalist agents," in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=OJd3ayDDoF

[16] N. Li, Y. Li, Y. Liu, L. Shi, K. Wang, and H. Wang, "Drowzee: Metamorphic testing for fact-conflicting hallucination detection in large language models," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3689776

[17] G. Guo, A. Aleti, N. Neelofar, C. Tantithamthavorn, Y. Qi, and T. Y. Chen, "Mortar: Multi-turn metamorphic testing for llm-based dialogue systems," 2025. [Online]. Available: https://arxiv.org/abs/2412.15557

[18] B. Yang, M. A. Al Mamun, J. Zhang, and G. Uddin, "Hallucination detection in large language models with metamorphic relations," *Proceedings of the ACM on Software Engineering*, vol. 2, pp. 425–445, 06 2025.

[19] "Libraries.io," https://libraries.io/, 2015, last accessed Mar. 2025.

[20] J. Yu, X. Lin, Z. Yu, and X. Xing, "Gptfuzzer: Red teaming large language models with auto-generated jailbreak prompts," 2024. [Online]. Available: https://arxiv.org/abs/2309.10253

[21] "Replication package," https://github.com/YukZhao/HFuzzer, 2025, last accessed Mar. 2025.

[22] "Tiobe index," https://www.tiobe.com/tiobe-index/, 2000, last accessed Mar. 2025.

[23] A. Meta, "Introducing meta llama 3: The most capable openly available llm to date," https://ai.meta.com/blog/meta-llama-3/, 2024, last accessed Mar. 2025.

[24] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang *et al.*, "Qwen2.5-coder technical report," 2024. [Online]. Available: https://arxiv.org/abs/2409.12186

[25] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang *et al.*, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024. [Online]. Available: https://arxiv.org/abs/2401.14196

[26] A. Meta, "Llama 3.1," https://ai.meta.com/blog/meta-llama-3-1/, 2024, last accessed Mar. 2025.

[27] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[28] A. Meta, "Llama 3.3," https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/, 2024, last accessed Mar. 2025.

[29] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," 2025. [Online]. Available: https://arxiv.org/abs/2412.19437

[30] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2303.08774

[31] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford *et al.*, "Gpt-4o system card," 2024. [Online]. Available: https://arxiv.org/abs/2410.21276

[32] "vllm," https://docs.vllm.ai/, 2024, last accessed Mar. 2025.

[33] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, J. Yu, Y. Wang, X. Lin, T. Chen, and Z. Zheng, "When chatgpt meets smart contract vulnerability detection: How far are we?" *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 4, Apr. 2025. [Online]. Available: https://doi.org/10.1145/3702973

[34] D. Liyanage, N. Attanayake, Z. Luo, and R. Gopinath, "Assessing reliability of statistical maximum coverage estimators in fuzzing," 2025. [Online]. Available: https://arxiv.org/abs/2507.17093

[35] D. Shriver, S. Elbaum, M. B. Dwyer, and D. S. Rosenblum, "Evaluating recommender system stability with influence-guided fuzzing," in *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'19/IAAI'19/EAAI'19. AAAI Press, 2019. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.33014934

[36] H. Abdi, "Coefficient of variation," *Encyclopedia of research design*, vol. 1, no. 5, pp. 169–171, 2010.

[37] J. Cui, W.-L. Chiang, I. Stoica, and C.-J. Hsieh, "OR-bench: An over-refusal benchmark for large language models," 2025. [Online]. Available: https://openreview.net/forum?id=obYVdcMMIT

[38] OpenAI, "text-embedding-3-small," https://platform.openai.com/docs/models/text-embedding-3-small/, 2024, last accessed Aug. 2025.

[39] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "Dbscan revisited, revisited: Why and how you should (still) use dbscan," *ACM Trans. Database Syst.*, vol. 42, no. 3, Jul. 2017. [Online]. Available: https://doi.org/10.1145/3068335

[40] "Chatgpt," https://chatgpt.com/, 2022, last accessed Mar. 2025.

[41] J. Latendresse, S. Khatoonabadi, A. Abdellatif, and E. Shihab, "Is chatgpt a good software librarian? an exploratory study on the use of chatgpt for software library recommendations," 2024. [Online]. Available: https://arxiv.org/abs/2408.05128

[42] R. Hu, C. Peng, X. Wang, J. Xu, and C. Gao, "Repo2run: Automated building executable environment for code repository at scale," 2025. [Online]. Available: https://arxiv.org/abs/2502.13681

[43] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," 2024. [Online]. Available: https://arxiv.org/abs/2312.10997

[44] "Github," https://github.com/, 2008, last accessed Mar. 2025.

[45] B. Lanyado, "Diving deeper into ai package hallucinations," https://www.lasso.security/blog/ai-package-hallucinations/, 2024, last accessed Mar. 2025.

[46] C. Chen, K. Liu, Z. Chen, Y. Gu, Y. Wu, M. Tao, Z. Fu, and J. Ye, "INSIDE: LLMs' internal states retain the power of hallucination detection," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=Zj12nzlQbz

[47] T. Zhang, L. Qiu, Q. Guo, C. Deng, Y. Zhang, Z. Zhang, C. Zhou, X. Wang, and L. Fu, "Enhancing uncertainty-based hallucination detection with stronger focus," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 915–932. [Online]. Available: https://aclanthology.org/2023.emnlp-main.58/

[48] E. Quevedo, J. Y. Salazar, R. Koerner, P. Rivas, and T. Cerny, "Detecting hallucinations in large language model generation: A token probability approach," in *Artificial Intelligence and Applications*, H. R. Arabnia, L. Deligiannidis, S. Amirian, F. Shenavarmasouleh, F. Ghareh Mohammadi, and D. de la Fuente, Eds. Cham: Springer Nature Switzerland, 2025, pp. 154–173.

[49] Y. Tian, W. Yan, Q. Yang, X. Zhao, Q. Chen, W. Wang, Z. Luo, L. Ma, and D. Song, "Codehalu: investigating code hallucinations in llms via execution-based verification," in *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'25/IAAI'25/EAAI'25. AAAI Press, 2025. [Online]. Available: https://doi.org/10.1609/aaai.v39i24.34717

[50] M. H. Tanzil, J. Y. Khan, and G. Uddin, "Chatgpt incorrectness detection in software reviews," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639194

[51] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2023.

[52] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs? an evaluation on quixbugs," in *Proceedings of the Third International Workshop on Automated Program Repair*, ser. APR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 69–75. [Online]. Available: https://doi.org/10.1145/3524459.3527351

[53] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 1469–1481. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00128

[54] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 5, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3643674

[55] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '24. ACM, Sep. 2024, p. 819–831. [Online]. Available: http://dx.doi.org/10.1145/3650212.3680323

[56] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639226

[57] X. Zhao, H. Qu, J. Xu, X. Li, W. Lv, and G.-G. Wang, "A systematic review of fuzzing," *Soft Comput.*, vol. 28, no. 6, p. 5493–5522, Oct. 2023. [Online]. Available: https://doi.org/10.1007/s00500-023-09306-2

[58] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[59] M. Lee, S. Cha, and H. Oh, "Learning seed-adaptive mutation strategies for greybox fuzzing," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 384–396. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00043

[60] X. Liu, W. You, Y. Ye, Z. Zhang, J. Huang, and X. Zhang, "Fuzzinmem: Fuzzing programs via in-memory structures," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639172

[61] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, "Fuzzing automatic differentiation in deep-learning libraries," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 1174–1186. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00105

[62] K. Hough and J. Bell, "Crossover in parametric fuzzing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639160

[63] J. Wang, P. Qian, X. Huang, X. Ying, Y. Chen, S. Ji, J. Chen, J. Xie, and L. Liu, "Tacoma: Enhanced browser fuzzing with fine-grained semantic alignment," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1174–1185. [Online]. Available: https://doi.org/10.1145/3650212.3680351

[64] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2024, pp. 1547–1559. [Online]. Available: https://doi.ieeecomputersociety.org/

[65] 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*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1656–1668. [Online]. Available: https://doi.org/10.1145/3650212.3680389

[66] 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 IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623343

[67] D. Yao, J. Zhang, I. G. Harris, and M. Carlsson, "Fuzzllm: A novel and universal fuzzing framework for proactively discovering jailbreak vulnerabilities in large language models," in *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2024, pp. 4485–4489.

[68] G. Deng, Y. Liu, Y. Li, K. Wang, Y. Zhang, Z. Li, H. Wang, T. Zhang, and Y. Liu, "Masterkey: Automated jailbreaking of large language model chatbots," in *Proceedings 2024 Network and Distributed System Security Symposium*, ser. NDSS 2024. Internet Society, 2024. [Online]. Available: http://dx.doi.org/10.14722/ndss.2024.24188

[69] S. Zhang, Y. Zhai, K. Guo, H. Hu, S. Guo, Z. Fang, L. Zhao, C. Shen, C. Wang, and Q. Wang, "Jbshield: defending large language models from jailbreak attacks through activated concept analysis and manipulation," in *Proceedings of the 34th USENIX Conference on Security Symposium*, ser. SEC '25. USA: USENIX Association, 2025.

[70] Y. Wang, Z. Shi, A. Bai, and C.-J. Hsieh, "Defending llms against jailbreaking attacks via backtranslation," 2024. [Online]. Available: https://arxiv.org/abs/2402.16459

[71] E. Jones, H. Palangi, C. S. Ribeiro, V. Chandrasekaran, S. Mukherjee, A. Mitra, A. H. Awadallah, and E. Kamar, "Teaching language models to hallucinate less with synthetic tasks," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=xpw7V0P136

[72] X. Yu, L. Liu, X. Hu, J. W. Keung, J. Liu, and X. Xia, "Fight fire with fire: How much can we trust chatgpt on source code-related tasks?" *IEEE Trans. Softw. Eng.*, vol. 50, no. 12, p. 3435–3453, Dec. 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3492204

[73] N. Jain, R. Kwiatkowski, B. Ray, M. K. Ramanathan, and V. Kumar, "On mitigating code llm hallucinations with api documentation," 2024. [Online]. Available: https://arxiv.org/abs/2407.09726