Sanitizing Large Language Models in Bug Detection with Data-Flow

Chengpeng Wang[1], **Wuqi Zhang**[2], **Zian Su**[1], **Xiangzhe Xu**[1], **Xiangyu Zhang**[1]
[1] Purdue University [2] Hong Kong University of Science and Technology
{wang6590, su284, xu1415, xyzhang}@purdue.edu
wuqi.zhang@connect.ust.hk

Abstract

Large language models (LLMs) show potential in code reasoning tasks, facilitating the customization of detecting bugs in software development. However, the hallucination effect can significantly compromise the reliability of bug reports. This work formulates a new schema of bug detection and presents a novel *sanitization* technique that detects false positives for hallucination mitigation. Our key idea is to enforce LLMs to emit data-flow paths in few-shot chain-of-thought prompting and validate them via the *program-property decomposition*. Specifically, we dissect data-flow paths into basic properties upon concise code snippets and leverage parsing-based analysis and LLMs for validation. Our approach averagely achieves 91.03% precision and 74.00% recall upon synthetic benchmarks, and boosts the precision by 21.99% with the sanitization. The evaluation upon real-world Android malware applications also demonstrates the superiority over an industrial analyzer, surpassing the precision and recall by 15.36% and 3.61%, respectively. LLMSAN is open-sourced at https://github.com/chengpeng-wang/LLMSAN.

1 Introduction

The advancement of generative AI has significantly improved software development efficiency (Wang et al., 2021; Rozière et al., 2023). Many even believe that AIs can become proficient programmers. However, just like human developers, AIs can make mistakes that potentially lead to program bugs, which necessitates bug detection in the new development paradigm. Classic approaches predominantly apply program analysis in the production line before software release (Shi et al., 2018; Arzt et al., 2014). In Figure 1, for example, developers can diagnose and fix a divide-by-zero (DBZ) bug when the program analyzer determines that the potential zero value of `parseInt(arg)` at line 13 is finally used as the divisor at line 5. However, these techniques demand the implementation of complex semantic analyses and compilation of the examined code, which restrains the construction and application of bug detectors, especially during the early development stage, when programs are incomplete and cannot be compiled (Do et al., 2022).

```
 1 public static void bar(int x, int y){
 2   if (x != 0)
 3     return (y * 1.0 / x);
 4   else
 5     return (x * 1.0 / y); //bug
 6 }
 7 public static void main(){
 8   int a = 0; //zero
 9   int b = parseInt(“123”);
10   System.out.println(bar(b, a));
11   String arg = args[0];
12   int c = parseInt(arg); //potential zero
13   System.out.println(bar(a, c));
14   c = b;
15   System.out.println(bar(a, c));
16 }
```

Figure 1: A program with a divide-by-zero bug at line 5

Recent advancements in Large Language Models (LLMs) offer exciting opportunities, allowing non-experts of program analysis to construct bug detectors and handle incomplete programs. Ideally, developers can easily build a custom bug detector via few-shot chain-of-thought (CoT) prompting (Wei et al., 2022), where they only need to provide several buggy programs with explanations as examples. However, the non-alignment between LLMs and program semantics often leads to substantial hallucinations (Zhang et al., 2023), significantly compromising the validity of detection results. For example, LLMs may report a DBZ bug at line 3 because they cannot understand the semantic correlation between the division at line 3 and the zero-value check at line 2. Our preliminary experiment also shows that `gpt-3.5-turbo-0125` produces 184 false positives in 239 reports when detecting null pointer dereference (NPD) in Juliet Test Suite (Juliet Test Suite, 2024). The substantial false positives diminish user confidence in the detection results, thereby preventing the practical adoption of the detector (Christakis and Bird, 2016).

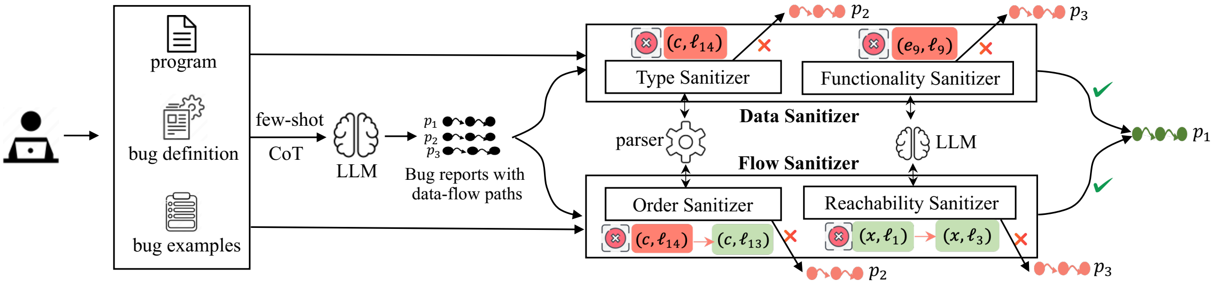
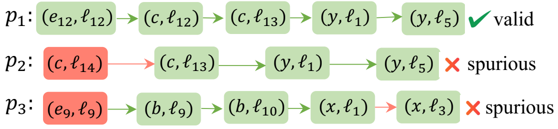


Figure 2: The workflow of LLMSAN. In the output, valid data-flow paths in green reveal true bugs, whereas spurious data-flow paths in red represent false positives. In the DBZ detection upon the program in Figure 1, the four sanitizers identify undesired start/end values in the data sanitization and incorrect intermediate data-flow facts in the flow sanitization, eventually labeling the paths p_2 and p_3 as spurious data-flow paths.



p_1: (e_{12}, ℓ_{12}) → (c, ℓ_{12}) → (c, ℓ_{13}) → (y, ℓ_1) → (y, ℓ_5) ✓ valid
p_2: (c, ℓ_{14}) → (c, ℓ_{13}) → (y, ℓ_1) → (y, ℓ_5) ✗ spurious
p_3: (e_9, ℓ_9) → (b, ℓ_9) → (b, ℓ_{10}) → (x, ℓ_1) → (x, ℓ_3) ✗ spurious

Figure 3: The examples of data-flow paths in Figure 1. e_9 and e_{12} are the function call expressions at lines 9 and 12, respectively. ℓ_n indicates the line n.

Program: [Example Program with a DBZ bug]
Explanation: v is assigned with 0 at line 3. Then u is initialized with v at line 7 and used as a divisor at line 9, causing a DBZ bug.
Dataflow path: [(v, ℓ_3), (u, ℓ_7), (u, ℓ_9)]

Figure 4: An example used in few-shot CoT prompting

This work facilitates few-shot CoT prompting-based bug detection by mitigating the hallucinations. The overarching idea is to enforce LLMs to emit a *data-flow path* as the bug proof and examine its validity. Intuitively, data-flow paths illustrate how faulty values are produced, propagated, and utilized by risky operations, leading to observable failures. For example, p_1 in Figure 3 depicts the zero-value propagation from the return value of `parseInt` at line 12 to the divisor at line 5 in Figure 1. Analogous to how CoT helps NLP applications, instructing the model to emit a data-flow path forces it to consider causality intrinsic to the examined program. Unlike a natural language explanation, which tends to be flexible and informal, a data-flow path serves as a rigorous certificate of a bug report and can hence be further validated. For instance, a valid data-flow path revealing a DBZ bug should start from a (potential) zero value and end at a divisor. If a data-flow path violates the property, such as p_2 and p_3 in Figure 3, the bug report would be a false positive. Meanwhile, any contradiction between intermediate data-flow facts and program semantics refutes the validity of the whole path. For example, the data-flow fact $(c, \ell_{14}) \rightarrow (c, \ell_{13})$ of p_2 violates the control-flow order, while $(x, \ell_1) \rightarrow (x, \ell_3)$ in p_3 is unreachable due to the zero-value check at line 2.

As shown in Figure 2, we present LLMSAN, a novel bug detection methodology that incorporates a *sanitization* technique for hallucination mitigation. By offering several examples in the form shown in Figure 4, we obtain the data-flow paths in the few-shot CoT prompting and then validate them via *program-property decomposition*. Concretely, LLMSAN performs *data* and *flow* sanitizations, which examine start/end values and intermediate data-flow facts, respectively. For each sanitization, we further decouple *syntactic* properties from *semantic* ones, with the former verified by parsing-based sanitizers (i.e., type and order sanitizers) and the latter examined by LLM-powered sanitizers (i.e., functionality and reachability sanitizers). In the flow sanitization, for example, the order sanitizer checks if any intermediate data-flow fact violates the control-flow order, while the reachability sanitizer checks if a faulty value can be propagated between different lines. Notably, each sanitizer validates a basic property upon local snippets, which is more manageable than prompting LLMs with the whole program for data-flow path validation. Also, multiple sanitizers check correlated properties and hence achieve a cross-checking effect.

Our contributions include: (1) We introduce a new method of LLM based bug detection, named LLMSAN, which supports non-experts in customizing the analysis without compilation by specifying several buggy code snippets as few-shot examples. (2) We propose a novel sanitization technique that effectively mitigates hallucination in bug detection by validating the emitted data-flow paths. (3) We evaluate LLMSAN and demonstrate that it improves the precision of few-shot CoT prompting-based bug detection upon benchmark programs from 69.04% to 91.03% with little sacrifice of its recall. Also, LLMSAN surpasses the precision and recall of an industrial analyzer upon real-world programs by 15.36% and 3.61%, respectively.

2 Related Work

A considerable body of literature demonstrates great successes of LLMs in program synthesis (Lee et al., 2023; Ye et al., 2021), testing (Meng et al., 2024), and repair (Olausson et al., 2023; Jimenez et al., 2023). However, only a limited number of works have focused on complex code reasoning tasks, such as program verification (Wen et al., 2024; Wu et al., 2023) and static bug detection (Li et al., 2023). Several approaches employ CoT prompting to generate program invariants (Pei et al., 2023; Wen et al., 2024), which can then be verified by a model checker for program verification. Similarly, LLIFT generates specifications of library functions using prompting to aid in bug detection (Li et al., 2023). Several other studies enhance the performance of LLMs in specific code reasoning tasks with trained small models (Steenhoek et al., 2024; Yadavally et al., 2024). This work targets LLM-powered bug detection with the support of customization and compilation-free analysis, which is not investigated by previous studies.

Hallucinations are a common issue of LLMs in many generative tasks, such as question-answering (Lin et al., 2021) and open-domain text generation (Mündler et al., 2023). In addition to general methods like self-consistency (Wang et al., 2022), self-check (Manakul et al., 2023), and CoT (Kojima et al., 2022), researchers have proposed domain-specific methods to address hallucinations. A recent study targets detecting self-contradictions to reduce hallucinations (Mündler et al., 2023). Other studies focus on hallucinations in various reasoning tasks, such as applying techniques like formal methods to mathematical word problems (Ye et al., 2024). Our study resolves the hallucinations in the bug detection, which involves the validation of sophisticated program properties.

3 Data-Flow Paths as Bug Proofs

Our work introduces a novel bug detection methodology named LLMSAN that mitigates the hallucination. As shown in Figure 2, the bug detection is achieved by (1) enforcing the LLMs to emit data-flow paths via few-shot CoT prompting and (2) designing a sanitization pipeline that further validates the emitted data-flow paths for hallucination mitigation. In this section, we introduce the concept of data-flow paths and formalize bug detection that emits data-flow paths as bug proofs.

Definition 1. (Data-flow Path) A data-flow path p is a sequence of k value-location pairs (v_i, ℓ_{n_i}). Each value v_i is a program variable or expression defined or used at line n_i. A *data-flow fact* is a pair of adjacent pairs indicating that the value v_i at line n_i directly affects the value v_{i+1} at line n_{i+1}.

Intuitively, a data-flow path demonstrates the value propagation within a program compactly. In the context of bug detection, a data-flow path from the origin and exposition point of a faulty value signifies the presence of a certain type of bug in the program. As proof of the bug, it can further help developers understand and repair the bug.

Example 1. In Figure 1, `parseInt(arg)` may evaluate to 0 as `arg` can be initialized to “0” by external input. The potential zero value is assigned to c at line 12 and eventually reach the divisor at line 5 as c is passed as the argument of the function `bar` at line 13. Hence, p_1 in Figure 3 is the proof of a potential DBZ bug in Figure 1.

Noting the power of LLMs in the code reasoning, we can customize a LLM-powered program analyzer using few-shot CoT prompting to explore and report data-flow paths for bug detection. Formally, we frame the problem of bug detection as follows.

Definition 2. (Bug Detection) Given a program P, a bug definition τ, and a set of few-shot examples $\mathcal{E}$, generate a set of data-flow paths $\mathcal{P}$ such that each $p \in \mathcal{P}$ indicates a bug τ in P. Here $e := (P_e, E_e, p_e) \in \mathcal{E}$ consists of the buggy program P_e, an explanation E_e, and a data-flow path p_e indicating a bug τ in P_e.

Example 2. To detect DBZ bugs in Figure 1, we offer examples in the format shown in Figure 4 to showcase how DBZ bugs arise. Specifically, zero values can be produced by numeric literals, such as 0 and 0.0, and function calls expressions, such as the ones invoking `parseInt` and `parseFloat` upon unconstrained strings. DBZ bugs can occur when the second operand of $/$ or $\%$ has zero value. Finally, we can obtain several data-flow paths as the proofs of DBZ bugs, such as the ones in Figure 3.

Definition 2 formulates a new paradigm of bug detection. Few-shot CoT prompting lowers the barrier for developers to customize the analysis and seamlessly supports analyzing programs in various forms, including incomplete code during the development stage. Different from explanations in natural language, the data-flow paths offer rigorous proofs of the bugs, facilitating the post-validation of bug reports upon the examined program P.

4 LLM Sanitization for Bug Detection

Utilizing data-flow paths as bug proofs, we propose a novel sanitization technique that validates them to mitigate the hallucinations in the bug reports. Specifically, validating data-flow paths entails handling two complexities. First, a data-flow path can bypass multiple program values in different functions, making it difficult to validate a long path in a large-sized code snippet. Second, the undecidability of the data-flow problem (Reps, 2000) hinders the full utilization of deterministic algorithms for validation with a correctness guarantee. To address the challenges, we introduce **program-property decomposition**, which resolves the complexities above from two orthogonal aspects.

I. Program Decomposition: The data-flow path for a bug essentially denotes step-by-step propagation of faulty values. Its validity fundamentally depends on (1) whether the start/end values adhere to faulty forms and (2) whether the intermediate data-flow facts align with program semantics. Hence, we can decompose data-flow path validation into two sub-problems, in which we separately examine the start/end values and individual intermediate data-flow facts upon small-sized snippets.

II. Property Decomposition: Along an orthogonal direction, we further separate syntactic properties from semantic properties. Specifically, the former focuses on properties that can be verified by deterministic algorithms based on parsers, such as checking if the start/end values must have certain syntactic types, while the latter concentrates on properties that require in-depth interpretation of program semantics using LLMs, such as the behavior of a specific function call.

Technically, we propose *data sanitization* and *flow sanitization* by designing several sanitizers for syntactic and semantic property validation. The data-flow paths successfully validated by all the sanitizers would be finalized as true bug reports. In what follows, we detail the designs of the two kinds of sanitizations.

4.1 Data Sanitization

As shown in Section 3, the first and last value-location pairs, i.e., (v_1, ℓ_{n_1}) and (v_k, ℓ_{n_k}), should indicate the origin and exposition point of a faulty value, respectively. To validate whether they conform to expected patterns, we introduce *type sanitizer* and *functionality sanitizer* to examine their syntactic and semantic properties, respectively.

Examples: Here are several examples containing DBZ bugs: [examples]. Please understand how zero values are produced.

Task: Analyze the code: [function]. Please check whether [value] at line [line number] can be zero. Please think step by step and conclude the answer with Yes or No.

Figure 5: The prompt template of functionality sanitizer

Type Sanitizer. This sanitizer leverages the observation that the start/end points of a reported valid data-flow path must have the same syntactic types as those in the provided few-shot examples. Therefore, we can leverage parsing techniques to identify syntactic types and verify the *type consistency*, which is formulated as follows.

Definition 3. (Type Consistency) α_h and α_l map a pair of program P and data-flow path p to the syntactic types for the start and the values of the data-flow path, respectively. Given a bug report, its data-flow path p being valid implies that it satisfies

$$\widehat{\alpha}(P,\ p)\ \cap \mathcal{A}(\mathcal{E}, \widehat{\alpha}) \neq \emptyset,\ \ \widehat{\alpha} \in \{\alpha_h, \alpha_l\} \qquad (1)$$

where $\mathcal{A}(\mathcal{E}, \widehat{\alpha})$ is defined as follows:

$$\mathcal{A}(\mathcal{E}, \widehat{\alpha}) = \bigcup_{(P_e,\ E_e,\ p_e) \in \mathcal{E}} \widehat{\alpha}(P_e,\ p_e) \qquad (2)$$

Example 3. As shown in Example 2, the start values in the provided few-shot examples are numeric literals and function calls, i.e., $\mathcal{A}(\mathcal{E}, \widehat{\alpha}) = \{\texttt{numeric literal}, \texttt{function call}\}$. We notice that the data-flow path p_2 starts from an identifier, i.e., $\alpha_h(P, p_2) = \{\texttt{identifier}\}$, implying that $\alpha_h(P, p_2) \cap \mathcal{A}(\mathcal{E}, \widehat{\alpha}) = \emptyset$. Hence, p_2 violates the type consistency, indicating that p_2 is spurious.

Functionality Sanitizer. In addition to type consistency, the start and end values of valid data-flow paths should also exhibit consistent functionalities with those in the few-shot examples. For instance, the data-flow paths revealing DBZ bugs should start with the variables or expressions that (potentially) evaluate to zero. Due to the undecidability of semantic analysis, the functionality sanitizer achieves the validation by prompting LLMs as follows.

Prompt Design: Following the prompt template in Figure 5, the functionality sanitizer validates the first value-location pair (v_1, ℓ_{n_1}) by checking if it can introduce a zero value. Formally speaking, it autoregressively samples response tokens from the conditional distribution p_θ denoting the LLM:

$$r_h \sim p_\theta(\cdot \mid \mathcal{E},\ F_1,\ v_1,\ \ell_{n_1}) \qquad (3)$$

Here F_1 is the function containing v_1. Similarly, we can apply functionality sanitizer for the value-location pair (v_k, ℓ_{n_k}).

Example 4. When comparing the start value of p_3 in Figure 1(c) with the few-shot examples sketched in Example 2, we obtain the answer "NO" via prompting, as the argument of the function call is a string literal "123". Hence, there is no zero value produced at line 10 in Figure 1.

4.2 Flow Sanitization

As stated in Definition 1, the validity of a data-flow path hinges on not just its start and end values but also its intermediate data-flow facts. Essentially, intermediate data-flow facts depict value propagation in detail and thus should align with program semantics. Hence, in addition to the data sanitization, we introduce the flow sanitization to examine intermediate data-flow facts, which is accomplished via *order sanitizer* and *reachability sanitizer*.

Order Sanitizer. Valid data-flow facts should adhere to the order dictated by runtime execution, i.e., the *control-flow order*. Intuitively, a faulty value can only be propagated from a program location to its next when the statement at the preceding one can be executed before the latter. Formally, we formulate the *order consistency* as follows.

Definition 4. (Order Consistency) A valid data-flow path $p : (v_1, \ell_{n_1}) \rightarrow \cdots \rightarrow (v_k, \ell_{n_k})$ satisfies

$$\forall 1 \leq i \leq (k-1),\ \ell_{n_i} \preceq \ell_{n_{i+1}} \quad (4)$$

Here, the partial order $\preceq$ over program locations denotes the control-flow order.

Technically, the order sanitizer derives program structures with a parser, such as branches, loops, and function calls, from which it determines the control-flow order for the order-consistency checking. We elide the details of determining the control-flow order, as it is not our contribution.

Example 5. In Figure 1, the statement at line 13 must be executed before the one at line 14, i.e., $\ell_{13} \preceq \ell_{14}$. The data-flow fact $(c, \ell_{14}) \rightarrow (c, \ell_{13})$ violates the order consistency, and thus, p_2 in Figure 3 is spurious.

Reachability Sanitizer. A data-flow fact satisfying the order consistency may still be invalid if the faulty value cannot be propagated along the data-flow path because of some check by a conditional statement. In such case, we say the data-flow fact is *unreachable*. Since examining the reachability requires understanding program semantics, we follow the design of functionality sanitizer and leverage an LLM to instantiate the reachability sanitizer.

Examples: There are examples containing DBZ bugs: [examples]. Please understand how program values are propagated.

Task: Analyze [function 1], [function 2]. Please check whether [value 1] at line [line number 1] can be propagated to [value 2] at line [line number 2]. Please think step by step and conclude the answer with Yes or No.

Figure 6: The prompt template of reachability sanitizer

Prompt Design: For a bug report with the data-flow path $p : (v_1, \ell_{n_1}) \rightarrow \cdots \rightarrow (v_k, \ell_{n_k})$, the reachability sanitizer examines the adjacent variable-location pairs with $(k-1)$ rounds of prompting, in which it instantiates the prompt template in Figure 6. Consider $1 \leq i \leq (k-1)$, the reachability of $(v_i, \ell_{n_i}) \rightarrow (v_{i+1}, \ell_{n_{i+1}})$ is determined by the sampling results according to the following conditional distribution:

$$r_i \sim p_\theta(\cdot \mid \mathcal{E},\ F_i,\ F_{i+1},\ v_i,\ \ell_{n_i},\ v_{i+1},\ \ell_{n_{i+1}}) \quad (5)$$

Here F_i and F_{i+1} are the functions that contain v_i and v_{i+1}, respectively.

Example 6. Consider the data-flow fact $(x, \ell_1) \rightarrow (x, \ell_3)$ within the data-flow path p_3 in Figure 3. We prompt the LLM with the function `bar` in Figure 1 and examine whether the value of x at line 1, which has a zero value, can propagate to the value of x at line 3. The answer "No" offered by the LLM enables us to refute the validity of p_3.

4.3 Summary

The data and flow sanitizations offer two key advantages. First, they separate the analysis of the syntactic property from the semantic one. This allows us to deterministically identify undesired start/end values and inconsistent control-flow order using a parsing technique without compilation, ensuring the soundness of type and order sanitizers. Second, the functionality and reachability sanitizers solely focus on the functions containing start/end values and individual intermediate data-flow facts, which makes the validation more manageable than directly prompting the LLM with the whole program. Therefore, we can effectively avoid additional hallucination during sanitization.

5 Evaluation

We implement a prototype of LLMSAN for Java bug detection, which has been released online. Utilizing the parsing library `tree-sitter` (Brunsfeld, 2018), we implement a light-weight compilation-free static analysis core to support the type sanitizer and order sanitizer.

Table 1: The selected bug types and their CWE IDs

Bug Type	CWE ID
Absolute Path Traversal (APT)	CWE-23
Cross-Site Scripting (XSS)	CWE-79
OS Command Injection (OSCI)	CWE-78
Divide-by-Zero (DBZ)	CWE-369
Null Pointer Dereference (NPD)	CWE-476

Table 2: The list of baselines

Name	Approach	No Compile	Custom
Pinpoint	(Shi et al., 2018)	✗	✗
CodeFuseQuery	(Xie et al., 2024)	✓	✗
FSCoT	(Ullah et al., 2024)	✓	✓
Ask-Check	(Mündler et al., 2023)	✓	✓
CoT-Check	(Kojima et al., 2022)	✓	✓
SC-CoT-Check	(Wang et al., 2022)	✓	✓

Table 3: The performance of LLMSAN using `gpt-4`. The columns P and R indicate the precision and recall, respectively. $|\Delta V|$ ($|\Delta S|$) and $\frac{|\Delta V|}{|V|}$ ($\frac{|\Delta S|}{|S|}$) indicate the number and proportion of pruned valid (spurious) data-flow paths in the sanitization, respectively.

Bug Type	P (%)	R (%)	$\|\Delta V\|$	$\|\Delta S\|$	$\frac{\|\Delta V\|}{\|V\|}$ (%)	$\frac{\|\Delta S\|}{\|S\|}$ (%)
APT	98.18	54.00	0	45	0.00	97.83
XSS	89.77	79.00	0	12	0.00	57.14
OSCI	98.94	93.00	0	6	0.00	85.71
DBZ	92.68	76.00	0	29	0.00	82.86
NPD	75.56	68.00	2	46	2.86	67.65
Average	91.03	74.00	0.40	27.60	0.57	78.24

Datasets. We choose two benchmarks for evaluation: Juliet Test Suite (Juliet Test Suite, 2024) and TaintBench (Luo et al., 2022). First, Juliet Test Suite is a synthetic benchmark, covering typical bug types in Common Weakness Enumeration. Considering the resource constraint, we choose five popular bug types shown in Table 1 and randomly select 100 programs for each bug type. Notably, these five bug types have been extensively studied in previous research on program analysis due to their popularity. Second, TaintBench comprises 39 Android malware applications with 203 taint flows that potentially lead to the leakage of sensitive information (Arzt et al., 2014). Detecting these taint flows necessitates customization to define how sensitive data is generated and leaked. Both datasets provide the ground truth.

Settings. We run the four sanitizers independently to quantify their effectiveness. To reduce randomness, we set the temperature to 0 in CoT prompting so that we enforce the LLMs to adopt greedy encoding. To show the effectiveness across diverse LLM architectures, we assess LLMSAN with `gpt-3.5-turbo-0125`, `gpt-4-turbo-preview`, `gemini-1.0-pro`, and `claude-3-haiku`, respectively. For brevity, we denote them with `gpt-3.5`, `gpt-4`, `gemini-1.0`, and `claude-3` in the rest of the paper.

Baselines. We consider two kinds of baselines shown in Table 2. First, we choose three existing bug detectors, including Pinpoint, CodeFuseQuery, and FSCoT. Pinpoint analyzes the intermediate representations generated by compilers to find bugs and does not support customization (Shi et al., 2018). CodeFuseQuery supports bug detection without requiring any compilation infrastructure and does not support non-expert customization either (Xie et al., 2024). FSCoT is one of the evaluated techniques in (Ullah et al., 2024), detecting bugs via few-shot CoT prompting. Second, we follow existing studies (Mündler et al., 2023) and evaluate three hallucination mitigation techniques by utilizing them for the data-flow path validation after the few-shot CoT prompting. Specifically, Ask-Check directly asks the LLMs to validate data-flow paths (Mündler et al., 2023). CoT-Check enforces LLMs to think step by step via CoT prompting (Kojima et al., 2022). To enhance the certainty, we set the temperatures to 0 in Ask-Check and CoT-Check. Additionally, we introduce self-consistency (Wang et al., 2022) to the CoT prompting under the temperature of 0.5 for the validation, leading to the SC-CoT-Check baseline. Because lengthy prompts induced by large programs significantly increase the resource cost, we avoid a large value for the sampling number and set it to five. Notably, FSCoT, Ask-Check, CoT-Check, and SC-CoT-Check are all compilation-free and customizable for non-experts.

Metrics. We compare the data-flow paths with the ground-truth offered by Juliet Test Suite and TaintBench to measure the precision and recall. Besides, we count the number of pruned valid and spurious data-flow paths during the sanitization. We also quantify the token cost of LLMSAN and the baseline FSCoT. Lastly, we introduce a group of ablation studies targeting various combinations of the four sanitizers, in which we count the numbers of identified spurious data-flow paths when multiple analyses are enabled simultaneously.

5.1 Performance of LLMSAN

Table 3 shows the precision and recall of LLMSAN in detecting the five types of bugs upon Juliet Test Suite. Specifically, LLMSAN achieves a precision of 91.03% and a recall of 74.00% on average. Benefiting from the sanitization, LLMSAN identifies 45, 12, 6, 29, and 46 spurious data-flow

Table 4: The performance of Ask-Check, CoT-Check, and SC-CoT-Check using gpt-4. The columns $|\Delta V|$, $|\Delta S|$, $\frac{|\Delta V|}{|V|}$, and $\frac{|\Delta S|}{|S|}$ indicate the same quantities as the ones in Table 3.

Bug Type	Ask-Check				CoT-Check				SC-CoT-Check			
	$\|\Delta V\|$	$\|\Delta S\|$	$\frac{\|\Delta V\|}{\|V\|}$(%)	$\frac{\|\Delta S\|}{\|S\|}$(%)	$\|\Delta V\|$	$\|\Delta S\|$	$\frac{\|\Delta V\|}{\|V\|}$(%)	$\frac{\|\Delta S\|}{\|S\|}$(%)	$\|\Delta V\|$	$\|\Delta S\|$	$\frac{\|\Delta V\|}{\|V\|}$(%)	$\frac{\|\Delta S\|}{\|S\|}$(%)
APT	6	3	11.11	6.52	19	14	35.19	30.43	18	12	33.33	26.09
XSS	0	0	0.00	0.00	4	3	5.06	14.29	0	0	0.00	0.00
OSCI	0	0	0.00	0.00	0	0	0.00	0.00	0	0	0.00	0.00
DBZ	45	20	59.21	57.14	0	11	0.00	31.43	0	9	0.00	25.71
NPD	2	33	2.86	48.53	0	40	0.00	58.82	0	37	0.00	54.41
Average	10.60	11.20	14.64	22.44	4.60	13.60	8.05	36.28	3.60	11.60	6.67	21.24

Table 5: The performance of Pinpoint, CodeFuseQuery, and FSCoT using gpt-4. The columns P and R indicate precision and recall, respectively.

Bug Type	Pinpoint		CodeFuseQuery		FSCoT	
	P (%)	R (%)	P (%)	R (%)	P (%)	R (%)
APT	100.00	81.00	84.21	64.00	54.00	54.00
XSS	100.00	47.00	91.86	79.00	79.00	79.00
OSCI	100.00	31.00	87.10	54.00	93.00	93.00
DBZ	92.65	63.00	29.45	81.00	68.47	76.00
NPD	87.36	76.00	35.78	73.00	50.72	70.00
Average	96.00	59.60	65.68	70.20	69.04	74.40

paths associated with APT, XSS, OSCI, DBZ, and NPD bugs, respectively. Notably, the majority of the spurious data-flow paths are pruned by sanitization, accounting for 97.83%, 57.14%, 85.71%, 82.86%, and 67.65% of the spurious paths associated with the five bug types, respectively. On average, 78.24% spurious data-flow paths can be successfully identified in the sanitization. Meanwhile, LLMSAN only misidentifies two valid data-flow paths associated with NPD bugs as spurious ones, which decreases the recall by 2.00% in the NPD detection. The statistics in Appendix A.1 further demonstrate that when powered with diverse LLMs, LLMSAN can detect an average of 84.82% of spurious data-flow paths while minimizing the sacrifice of valid ones, prompting the precision of the bug detection without losing the recall.

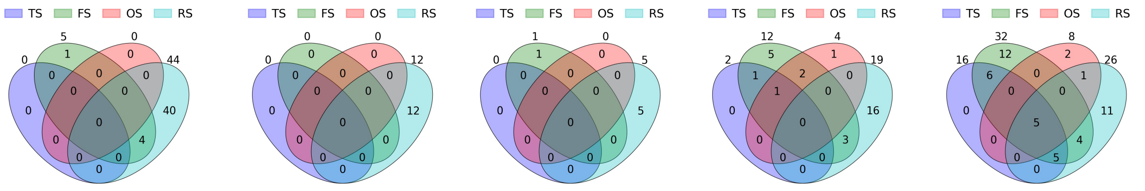
5.2 Comparison with Existing Bug Detectors

As shown in Table 5, Pinpoint achieves a high precision in detecting five types of bugs. However, its recall is not satisfactory in the detection of XSS and OSCI bugs, only obtaining 47.00% and 31.00%, respectively. The lack of customization support makes it fail to identify several forms of sensitive data that may introduce the XSS and OSCI bugs. CodeFuseQuery produces many false positives due to its inability of reasoning path conditions, especially in the detection of DBZ and NPD bugs. LLMSAN achieves a comparable and even better performance than the two analyzers, while it does not demand compilation or expert customization, ensuring better applicability and usability.

FSCoT achieves 0.40% higher recall than LLMSAN, while its average precision is 21.99% lower than that of LLMSAN. Without sanitization, the insufficient alignment between LLMs and program semantics causes the proliferation of false positives. We also quantify the token costs of LLMSAN and FSCoT. On average, the input and output token costs of LLMSAN are 1.82 and 3.37 times the ones of FSCoT, respectively, implying that LLMSAN only spends 92% more financial resources than FSCoT. Owing to the program-property decomposition, the prompts utilized in the two sanitizers consist of concise code snippets, ensuring that LLMSAN remains cost-effective while mitigating hallucination. More details of token costs are presented in Appendix A.4.

5.3 Comparison with Hallucination Mitigation Techniques

Table 4 shows the performance of Ask-Check, CoT-Check, and SC-CoT-Check using gpt-4. On one hand, they sacrifice several valid data-flow paths. For example, the three baselines misidentify 6, 19, and 18 valid data-flow paths associated with APT bugs as spurious ones, respectively. 59.21% valid data-flow paths re misidentified by Ask-Check in the DBZ detection. However, LLMSAN does not sacrifice any valid data-flow paths except for two valid ones in the NPD detection. On the other hand, LLMSAN has the overwhelming superiority in detecting spurious data-flow paths. While the baselines detect several spurious data-flow paths associated with specific bug types, their performance significantly differs across different bug types. Specifically, Ask-Check identifies 57.14% and 48.53% spurious data-flow paths associated with the DBZ and NPD bugs, respectively, whereas it detects few spurious data-flow paths for the other bug types. Similar observations can be derived for CoT-Check and SC-CoT-Check. Appendix A.2 presents the results when using other LLMs, from which we can derive the same findings.



(a) APT (b) XSS (c) OSCI (d) DBZ (e) NPD

Figure 7: The numbers of spurious data-flow paths detected by different combinations of sanitizers using `gpt-4`

Lastly, CoT prompting and self-consistency do not consistently mitigate hallucinations. For instance, Ask-Check identifies more spurious data-flow paths than CoT-Check and SC-CoT-Check in the DBZ detection. SC-CoT-Check identifies fewer spurious data-flow paths for the APT, XSS, DBZ, and NPD bugs compared to CoT-Check. The poor performance of CoT-Check and SC-CoT-Check may be attributed to the ineffective alignments with program semantics, leading to erroneous reasoning steps in CoT prompting and thereby preventing exploring correct reasoning paths.

5.4 Ablation Study

Figure 7 shows that all four sanitizers identify spurious data-flow paths when LLMSAN is powered by `gpt-4`. Take the NPD detection as an example. The type sanitizer (TS) and order sanitizer (OS) detect 16 and 8 spurious data-flow paths out of the total 46 identified spurious data-flow paths, respectively. The functionality sanitizer (FS) and reachability sanitizer (RS) uncover 32 and 26 spurious data-flow paths, respectively, indicating that they play significant roles in resolving the majority of spurious data-flow paths associated with NPD bugs. While the sets of spurious data-flow paths identified by different sanitizers can overlap, each sanitizer can contribute to mitigating hallucinations by uniquely identifying specific spurious data-flow paths for a specific bug type, which is further demonstrated by the results of ablations powered by other LLMs in Appendix A.3. We also demonstrate several cases of spurious data-flow paths identified by different sanitizers in Appendix B.1.

5.5 Experiments on Real-world Programs

As shown in Table 6, when analyzing real-world Android malware applications, all the four sanitizers can identify spurious data-flow paths effectively. Only the functionality sanitizer and reachability sanitizer introduce the loss of two and one valid data-flow paths, respectively, when LLMSAN is powered by `gemini-1.0`. The proportion of identified spurious data-flow paths can reach 86.59%, 88.27%, 75.88%, and 94.20% when using the four LLMs, respectively. Powered with `gpt-4`, LLMSAN reports 89 data-flow paths in total, of which 48 are valid, achieving the precision of 44.04% (= 48/89). Due to the old Gradle version, the applications are not compilable in our environment, making Pinpoint inapplicable in such scenarios. Hence, we only evaluate CodeFuseQuery and find that it reports 136 taint bugs while only 39 of them are valid, obtaining the precision of 28.68%. The results reveal the impressive potential of LLMSAN in real-world bug detection. Powered with `gpt-4`, LLMSAN surpasses the precision and recall of CodeFuseQuery by 15.36% and 3.61%, respectively, while the latter requires substantial expertise and implementation efforts for customization.

Table 6: The statistics of LLMSAN upon TaintBench. $(\downarrow a)$, $(\downarrow b)$, and $(\downarrow c)$ indicate a data-flow paths, b valid data-flow paths, and c spurious data-flow paths are not reported due to the sanitization.

LLM	# Pruned				# Reported		
	TS	FS	OS	RS	Total	Valid	Spurious
`gemini-1.0`	25	74	57	103	40(↓ 145)	18(↓ 3)	22(↓ 142)
`gpt-3.5`	24	139	41	120	34(↓ 173)	11(↓ 0)	23(↓ 173)
`gpt-4`	34	51	54	85	**89**(↓ 129)	**48**(↓ 0)	**41**(↓ 129)
`claude-3`	123	516	147	335	52(↓ 536)	19(↓ 0)	33(↓ 536)

6 Conclusion

This paper introduces LLMSAN, an innovative bug detection methodology that employs a sanitization technique to mitigate hallucination. Based on the program-property decomposition, LLMSAN utilizes parsing-based sanitizers and LLM-powered sanitizers to verify multiple basic syntactic and semantic properties upon small code snippets, ultimately identifying spurious data-flow paths emitted in the few-shot CoT prompting. Our evaluation shows its promising performance in detecting bugs upon both synthetic benchmark programs and real-world Android malware applications. Our work lays the foundation for more reliable LLM-driven program analysis techniques, encompassing areas such as bug detection, automatic debugging, and program repair.

7 Limitations

Although LLMSAN has shown effectiveness in the evaluation subjects, several limitations necessitate further enhancements in the future. First, the functionality sanitizer and reachability sanitizer in LLMSAN may erroneously flag valid data-flow paths as spurious. Due to the usage of LLMs, they can potentially introduce hallucinations in the sanitization. To enhance the soundness, we can incorporate existing strategies, such as self-consistency, into the prompting process. Second, LLMSAN can fail to identify specific spurious data-flow paths, hindering precision enhancement in bug detection. In Appendix B.2, we list an example of a spurious data-flow path that cannot be identified. Our work borrows insights from compiler design to alleviate hallucinations via program-property decomposition. Similar to the optimizations that can be added independently into a compiler pipeline, additional sanitizers can be easily integrated into our pipeline, further suppressing hallucination. New sanitizers should follow the same design principles, focusing on validating basic and local properties upon small code snippets. Third, LLMSAN only validates data-flow paths as a post-processing step and does not contribute to increasing valid data-flow paths. In the future, it is promising to assist the self-reflection with the sanitization, which can guide the LLMs to sample more valid paths iteratively.

Acknowledgments

We are grateful to the Center for AI Safety for providing computational resources. This work was funded in part by the National Science Foundation (NSF) Awards SHF-1901242, SHF-1910300, IIS-2416835, DARPA VSPELLS - HR001120S0058, IARPA TrojAI W911NF-19-S0012, ONR N000141712045, N000141410468 and N000141712947. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM.

Max Brunsfeld. 2018. Tree-sitter-a new parsing system for programming tools. In *Strange Loop Conference,. Accessed–. URL: https://www. thestrangeloop. com//tree-sitter—a-new-parsing-system-for-programming-tools. html.*

Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 332–343. ACM.

Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. 2022. Why do software developers use static analysis tools? A user-centered study of developer needs and motivations. *IEEE Trans. Software Eng.*, 48(3):835–847.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Juliet Test Suite. 2024. Juliet Test Suite for Java. `https://github.com/find-sec-bugs/juliet-test-suite`. [Online; accessed 21-Apr-2024].

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Celine Lee, Abdulrahman Mahmoud, Michal Kurek, Simone Campanoni, David Brooks, Stephen Chong, Gu-Yeon Wei, and Alexander M Rush. 2023. Guess & sketch: Language model guided transpilation. *arXiv preprint arXiv:2309.14396*.

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. The hitchhiker's guide to program analysis: A journey with large language models. *CoRR*, abs/2308.00245.

Stephanie Lin, Jacob Hilton, and Owain Evans. 2021. Truthfulqa: Measuring how models mimic human falsehoods. *arXiv preprint arXiv:2109.07958*.

Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. 2022. Taint-bench: Automatic real-world malware benchmarking of android taint analyses. *Empir. Softw. Eng.*, 27(1):16.

Potsawee Manakul, Adian Liusie, and Mark JF Gales. 2023. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. *arXiv preprint arXiv:2303.08896*.

Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. NDSS.

Niels Mündler, Jingxuan He, Slobodan Jenko, and Martin Vechev. 2023. Self-contradictory hallucinations of large language models: Evaluation, detection and mitigation. *arXiv preprint arXiv:2305.15852*.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants? In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 27496–27520. PMLR.

Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):162–186.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 693–706. ACM.

Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 16:1–16:13. ACM.

Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2024. Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 199–199. IEEE Computer Society.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*.

Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by large language models using static analysis and program verification. *arXiv preprint arXiv:2404.00762*.

Haoze Wu, Clark Barrett, and Nina Narodytska. 2023. Lemur: Integrating large language models in automated program verification. In *The Twelfth International Conference on Learning Representations*.

Xiaoheng Xie, Gang Fan, Xiaojun Lin, Ang Zhou, Shijie Li, Xunjin Zheng, Yinan Liang, Yu Zhang, Na Yu, Haokun Li, Xinyu Chen, Yingzhuang Chen, Yi Zhen, Dejun Dong, Xianjin Fu, Jinzhou Su, Fuxiong Pan, Pengshuai Luo, Youzheng Feng, Ruoxiang Hu, Jing Fan, Jinguo Zhou, Xiao Xiao, and Peng Di. 2024. Codefuse-query: A data-centric static code analysis system for large-scale organizations. *CoRR*, abs/2401.01571.

Aashish Yadavally, Yi Li, Shaohua Wang, and Tien N Nguyen. 2024. A learning-based approach to static program slicing. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):83–109.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2021. Optimal neural program synthesis from multimodal specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pages 1691–1704. Association for Computational Linguistics.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2024. Satlm: Satisfiability-aided language models using declarative prompting. *Advances in Neural Information Processing Systems*, 36.

Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, Longyue Wang, Anh Tuan Luu, Wei Bi, Freda Shi, and Shuming Shi. 2023. Siren's song in the AI ocean: A survey on hallucination in large language models. *CoRR*, abs/2309.01219.

Table 7: The performance of LLMSAN using different LLMs. The columns P and R indicate the precision and recall, respectively. $|\Delta V|$ ($|\Delta S|$) and $\frac{|\Delta V|}{|V|}$ ($\frac{|\Delta S|}{|S|}$) indicate the number and proportion of pruned valid (spurious) data-flow paths, respectively.

Model	Bug Type	P (%)	R (%)	$\lvert\Delta V\rvert$	$\lvert\Delta S\rvert$	$\frac{\lvert\Delta V\rvert}{\lvert V\rvert}$ (%)	$\frac{\lvert\Delta S\rvert}{\lvert S\rvert}$ (%)
`gpt-3.5`	APT	82.98	39.00	0	69	0.00	89.61
	XSS	94.52	69.00	0	52	0.00	92.86
	OSCI	96.43	81.00	0	117	0.00	97.50
	DBZ	0.00	0.00	0	164	0.00	82.00
	NPD	56.70	55.00	0	142	0.00	77.17
`gpt-4`	APT	98.18	54.00	0	45	0.00	97.83
	XSS	89.77	79.00	0	12	0.00	57.14
	OSCI	98.94	93.00	0	6	0.00	85.71
	DBZ	92.68	76.00	0	29	0.00	82.86
	NPD	75.56	68.00	2	46	2.86	67.65
`gemini-1.0`	APT	87.50	14.00	0	84	0.00	97.83
	XSS	91.78	67.00	0	27	0.00	81.82
	OSCI	87.50	42.00	0	52	0.00	89.66
	DBZ	12.50	2.00	0	84	0.00	85.71
	NPD	97.30	36.00	1	62	2.70	98.41
`claude-3`	APT	87.50	14.00	0	84	0.00	97.67
	XSS	88.64	78.00	0	86	0.00	89.58
	OSCI	87.50	42.00	0	52	0.00	89.66
	DBZ	33.10	47.00	0	205	0.00	68.33
	NPD	39.50	62.00	0	228	0.00	70.59

Appendix

A Additional Experimental Results

To show the effectiveness of the sanitization using diverse LLM architectures, we conduct the evaluation upon four different LLMs. In the main body of the paper, we only demonstrate the results of LLMSAN, its ablations, and baselines using `gpt-4`. In what follows, we demonstrate the complete evaluation results when using all the four models. Also, we present detailed statistics of the token costs of LLMSAN and FSCoT when using `gpt-4`.

A.1 Performance of LLMSAN using Different LLMs

Figure 7 shows the precision and recall of the bug detection with sanitization. Generally, LLMSAN achieves high precision in the APT, XSS, and OSCI detection. Due to the inherent drawbacks of the models, the recall of the DBZ detection is not as good as the one of the detection of other bug types. After investigating the benchmark programs used in the DBZ detection, we find that there are many different assignments from literal values, such as `Integer.MIN_VALUE`. On the one hand, such literal values can make LLMs wrongly identify them as potential zero values due to the insufficient alignment between LLMs and Java semantics, degrading the precision of the DBZ detection. On the other hand, a large number of non-zero literals may mislead LLMs to pay less attention to zero literals and the potential zero values returned by specific function calls, which eventually causes the low recall of the DBZ detection. Overall, although the performance of LLMSAN diverse among different LLMs, the statistics shown in Table 7 demonstrates the potential of LLMSAN in the bug detection.

Meanwhile, it is worth noting that the sanitization technique can identify most of the spurious data-flow paths, which is shown by the column $\frac{|\Delta S|}{|S|}$. Aside from the loss of one and two valid data-flow paths when using `gemini-1.0` and `gpt-4` in the NPD detection, respectively, no other detection instances sacrifice valid data-flow paths. Therefore, we can conclude that the sanitization technique can effectively address the hallucinations in bug detection with little loss of true bug reports.

Table 8: The performance of FSCoT using different LLMs. The rows P and R indicate the precision and recall, respectively.

		APT	XSS	OSCI	DBZ	NPD
`gpt-3.5`	P(%)	33.62	55.20	40.30	0.00	23.01
	R(%)	39.00	69.00	81.00	0.00	55.00
`gpt-4`	P(%)	54.00	79.00	93.00	68.47	50.72
	R(%)	54.00	79.00	93.00	76.00	70.00
`gemini-1.0`	P(%)	14.00	67.00	42.00	2.00	37.00
	R(%)	14.00	67.00	42.00	2.00	37.00
`claude-3`	P(%)	14.00	44.83	42.00	13.54	16.10
	R(%)	14.00	78.00	42.00	47.00	62.00

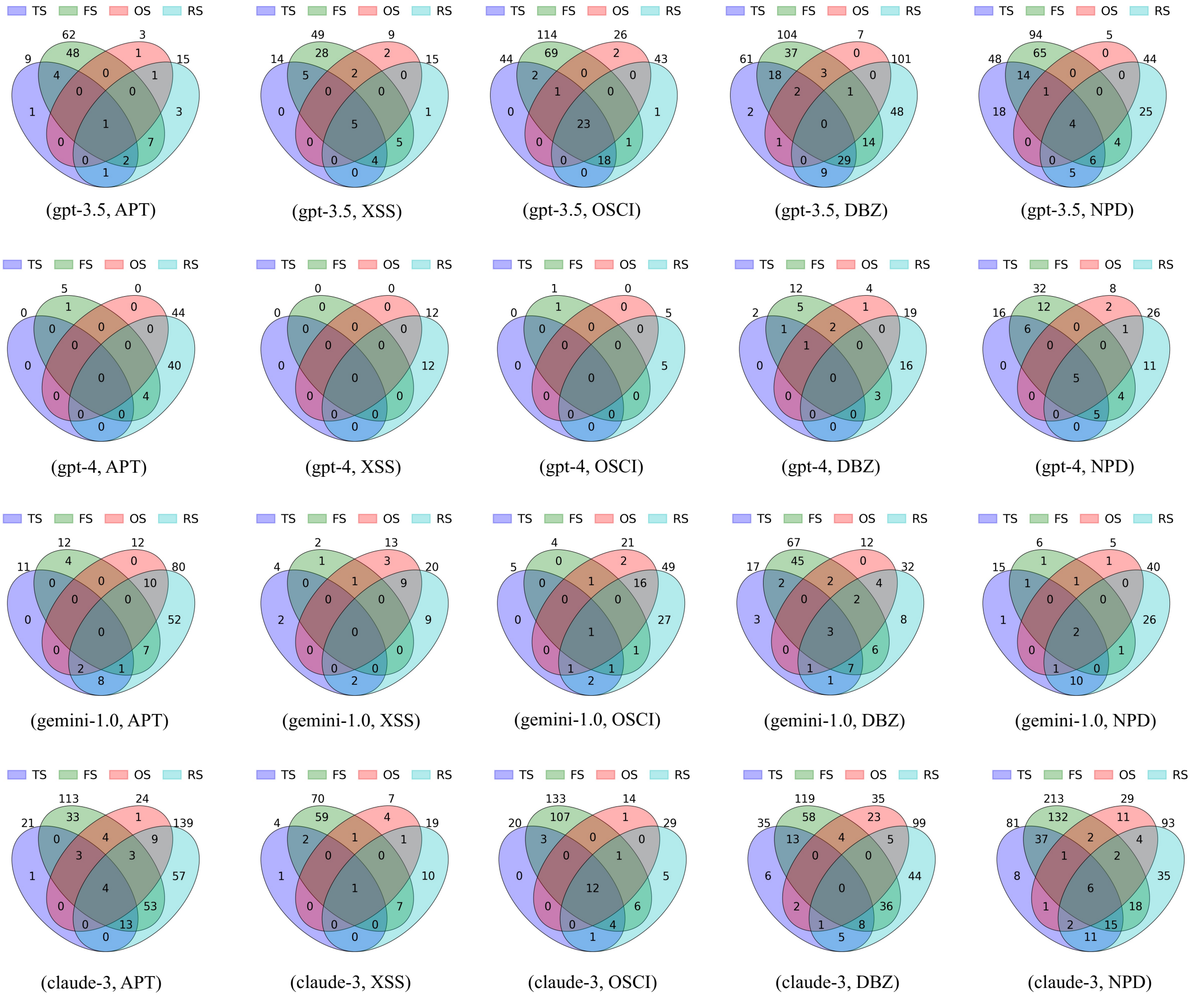
A.2 Comparison with Baselines using Different LLMs

Table 8 shows the performance of FSCoT using different LLMs. Compared with the statistics in the columns P and R of Table 7, we find that LLMSAN has the overwhelming superiority over FSCoT in terms of the preicions and achieves almost the same recall as LLMSAN. On average, LLMSAN obtains 28.71% higher precision in the detection of the five types of bugs using the four LLMs.

Table 9 shows the performance of different hallucination mitigation techniques. It is shown that the sanitization technique in LLMSAN has better performance than Ask-Check, CoT-Check, and SC-CoT-Check. Particularly, the three baselines can hardly resolve any spurious data-flow paths when they are powered with `gemini-1.0`. However, as shown in Table 7, LLMSAN can reduce the number of spurious data-flow paths dramatically. Additionally, LLMSAN sacrifices few valid data-flow paths when using different LLMs. Overall, the evaluation findings are consistent with the ones derived from the statistics obtained using `gpt-4`, showing that the sanitization technique surpasses existing approaches in hallucination mitigation.

Table 9: The performance of Ask-Check, CoT-Check, and SC-CoT-Check using different LLMs. The columns $|\Delta V|$, $|\Delta S|$, $\frac{|\Delta V|}{|V|}$, and $\frac{|\Delta S|}{|S|}$ indicate the same quantities as the ones in Table 7.

Model	Bug Type	Ask-Check				CoT-Check				SC-CoT-Check			
		$\|\Delta V\|$	$\|\Delta S\|$	$\frac{\|\Delta V\|}{\|V\|}(\%)$	$\frac{\|\Delta S\|}{\|S\|}(\%)$	$\|\Delta V\|$	$\|\Delta S\|$	$\frac{\|\Delta V\|}{\|V\|}(\%)$	$\frac{\|\Delta S\|}{\|S\|}(\%)$	$\|\Delta V\|$	$\|\Delta S\|$	$\frac{\|\Delta V\|}{\|V\|}(\%)$	$\frac{\|\Delta S\|}{\|S\|}(\%)$
`gpt-3.5`	APT	6	9	15.38	11.69	0	1	0.00	1.30	0	1	0.00	1.30
	XSS	1	2	1.45	3.57	0	1	0.00	1.79	0	7	0.00	12.50
	OSCI	0	0	0.00	0.00	0	16	0.00	13.33	0	5	0.00	4.17
	DBZ	0	127	0.00	63.50	0	26	0.00	13.00	0	10	0.00	5.00
	NPD	1	2	1.82	1.09	0	9	0.00	4.89	0	3	0.00	1.63
`gpt-4`	APT	6	3	11.11	6.52	19	14	35.19	30.43	18	12	33.33	26.09
	XSS	0	0	0.00	0.00	4	3	5.06	14.29	0	0	0.00	0.00
	OSCI	0	0	0.00	0.00	0	0	0.00	0.00	0	0	0.00	0.00
	DBZ	45	20	59.21	57.14	0	11	0.00	31.43	0	9	0.00	25.71
	NPD	2	33	2.86	48.53	0	40	0.00	58.82	0	37	0.00	54.41
`gemini-1.0`	APT	0	0	0.00	0.00	2	0	2.33	0.00	0	0	0.00	0.00
	XSS	0	0	0.00	0.00	0	0	0.00	0.00	0	0	0.00	0.00
	OSCI	0	0	0.00	0.00	0	0	0.00	0.00	0	0	0.00	0.00
	DBZ	0	0	0.00	0.00	0	1	0.00	1.02	0	3	0.00	3.06
	NPD	0	0	0.00	0.00	0	1	0.00	1.59	0	0	0.00	0.00
`claude-3`	APT	0	0	0.00	0.00	2	0	2.33	0.00	0	0	0.00	0.00
	XSS	0	9	0.00	9.38	0	9	0.00	9.38	0	51	0.00	53.13
	OSCI	0	0	0.00	0.00	0	10	0.00	17.24	0	0	0.00	0.00
	DBZ	0	27	0.00	9.00	0	19	0.00	6.33	0	8	0.00	2.67
	NPD	0	80	0.00	24.77	0	44	0.00	13.62	0	16	0.00	4.95



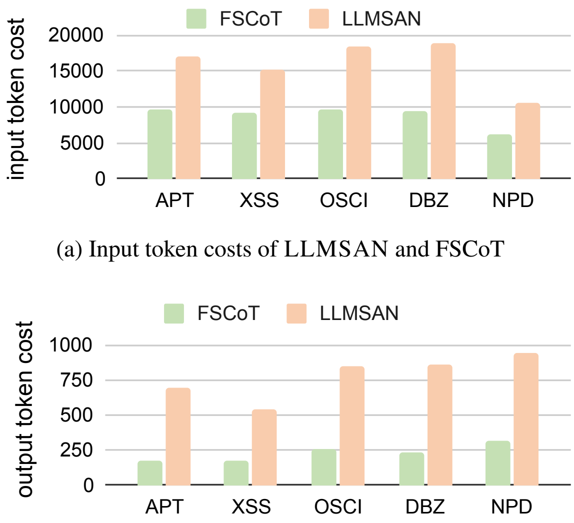
(gpt-3.5, APT) (gpt-3.5, XSS) (gpt-3.5, OSCI) (gpt-3.5, DBZ) (gpt-3.5, NPD)

(gpt-4, APT) (gpt-4, XSS) (gpt-4, OSCI) (gpt-4, DBZ) (gpt-4, NPD)

(gemini-1.0, APT) (gemini-1.0, XSS) (gemini-1.0, OSCI) (gemini-1.0, DBZ) (gemini-1.0, NPD)

(claude-3, APT) (claude-3, XSS) (claude-3, OSCI) (claude-3, DBZ) (claude-3, NPD)

Figure 8: The numbers of spurious data-flow paths detected by various combinations of sanitizers using four LLMs



(a) Input token costs of LLMSAN and FSCoT

(b) Input token costs of LLMSAN and FSCoT

Figure 9: Comparison between token costs of LLMSAN and FSCoT

A.3 Ablation Study using Different LLMs

Figure 8 depicts the numbers of detected spurious data-flow paths when using diverse combinations of sanitizers across four different LLMs. As shown by the sub-figures, each sanitizer is capable of detecting spurious data-flow paths in different categories of bug detection. Particularly, if we exclude any sanitizer from the sanitization pipeline, the count of spurious data-flow paths will increase in the detection of APT, XSS, and OSCI bugs using `gpt-3.5`. Likewise, all four sanitizers are able to uniquely identify several spurious data-flow paths during the APT, XSS, DBZ, and NPD detection when employing `claude-3`. Overall, the findings of the ablation study highlight the effectiveness of each sanitizer in mitigating hallucinations during bug detection across various bug types and using different LLMs.

A.4 Token Costs of LLMSAN and FSCoT

Figure 9 presents the token cost comparison between LLMSAN and FSCoT utilizing `gpt-4`. During the detection of five types of bugs, LLMSAN consumes 1.75, 1.66, 1.90, 2.01, and 1.72 times the input tokens compared to FSCoT, and 3.95, 3.14, 3.37, 3.61, and 2.99 times the output tokens. According to the billing policy, the additional financial costs incurred by LLMSAN compared to FSCoT are 0.86, 0.74, 1.00, 1.13, and 0.89, indicating that investing 95% more in financial costs would result in a 21.99% precision improvement on average.

B Case Study

According to Section 5 and Appendix A, LLMSAN can significantly mitigate the hallucinations in the bug detection. In what follows, we will show several typical examples of spurious data-flow paths identified by the four sanitizers. Besides, we will present an example spurious data-flow path that cannot be identified by LLMSAN.

B.1 Identified Spurious Data-flow paths

Case I: Consider the program in Figure 10. `gemini-1.0` emits the data-flow path starting from (“”, ℓ_{53}), while the few-shot examples do not contain any string literals as sensitive data. The type sanitizer detects the inconsistency between the syntactic types of the statement at line 53 and the ones in the few-shot examples, and thus, detects this spurious data-flow path.

Case II: Consider the program in Figure 11. `gpt-3.5` emits the data-flow path starting from a zero value and ending at (`data`, ℓ_{178}), while the value of the variable `data` cannot be zero due to the branch condition `data != 0`. The functionality sanitizer leverages `gpt-3.5` to validate the semantic property by only focusing on the function shown in Figure 11, eventually identifying the spurious data-flow path and mitigating the hallucination. This spurious data-flow path can also be identified by the reachability sanitizer.

Case III: Consider the program in Figure 12. `gemini-1.0` emits the data-flow path passing from (`data`, ℓ_{69}) to (`data`, ℓ_{56}). The order sanitizer discovers that the statement at line 56 should be executed before the statement at line 69 in the control flow graph, implying that the order consistency is violated by the data-flow path. Hence, LLMSAN identifies it as spurious one.

Case IV: Consider the program in Figure 13. `claude-3` emits the data-flow path from (`data`, ℓ_{43}) to (`data`, ℓ_{77}). The order sanitizer discovers that the statement at line 43 can not be executed before the one at line 77, implying that the order consistency is violated. Hence, LLMSAN identifies the data-flow path as spurious one.

Case V: Consider the program in Figure 14. `gpt-4` emits the data-flow path from (`data`, ℓ_{34}) to (`dataSerialized`, ℓ_{46}). By focusing on the two lines in the single function, the reachability sanitizer powered by `gpt-4` discovers that the value of `dataSerialized` at line 46 cannot be `null`, and detects this spurious data-flow path.

```
public void bad() throws Throwable
{
  String data;
  if (privateReturnsTrue())
  {
    data = ""; /* Initialize data */
    /* Read data using an outbound tcp connection */
    {
      Socket socket = null;
      BufferedReader readerBuffered = null;
      InputStreamReader readerInputStream = null;
      try
      {
        /* Read data using an outbound tcp connection */
        socket = new Socket("host.example.org", 39544);
        /* read input from socket */
        readerInputStream = new InputStreamReader(socket.getInputStream(), "UTF-8");
        readerBuffered = new BufferedReader(readerInputStream);
        /* POTENTIAL FLAW: Read data using an outbound tcp connection */
        data = readerBuffered.readLine();
      }
```

Figure 10: An example of spurious data-flow paths identified by **type sanitizer**

```
public void CWE369_Divide_by_Zero__int_Environment_modulo_22b_goodB2G1Sink(int data)
{
  if (CWE369_Divide_by_Zero__int_Environment_modulo_22a.goodB2G1PublicStatic)
  {
    /* INCIDENTAL: CWE 561 Dead Code, the code below will never run
    * but ensure data is inititialized before the Sink to avoid compiler errors */
    data = 0;
  }
  else
  {
    /* FIX: test for a zero modulus */
    if (data != 0)
    {
      IO.writeLine("100%" + data + " = " + (100 % data) + "\n");
    }
    else
    {
      IO.writeLine("This would result in a modulo by zero");
    }
  }
}
```

Figure 11: An example of spurious data-flow paths identified by **functionality sanitizer**

```
private String bad_source(HttpServletRequest request, HttpServletResponse response) throws Throwable
{
    String data;

    if (badPrivate)
    {
        data = ""; /* Initialize data */
        /* read input from URLConnection */
        {
            URLConnection urlConnection = (new URL("http://www.example.org/")).openConnection();
            BufferedReader readerBuffered = null;
            InputStreamReader readerInputStream = null;
            try
            {
                readerInputStream = new InputStreamReader(urlConnection.getInputStream(), "UTF-8");
                readerBuffered = new BufferedReader(readerInputStream);
                /* POTENTIAL FLAW: Read data from a web server with URLConnection */
                /* This will be reading the first "line" of the response body,
                    * which could be very long if there are no newlines in the HTML */
                data = readerBuffered.readLine();
            }
            catch (IOException exceptIO)
            {
               IO.logger.log(Level.WARNING, "Error with stream reading", exceptIO);
            }
```

Figure 12: An intra-procedural example of spurious data-flow paths identified by **order sanitizer**

```java
public void bad()
{
    StringBuilder data;
    if (privateReturnsTrue())
    {
        /* POTENTIAL FLAW: data is null */
        data = null;
    }
    else
    {
        /* INCIDENTAL: CWE 561 Dead Code, the code below will never run
        * but ensure data is initialized before the Sink to avoid compiler errors */
        data = null;
    }

    if (privateReturnsTrue())
    {
        /* POTENTIAL FLAW: null dereference will occur if data is null */
        IO.writeLine("" + data.length());
    }
}
```

```java
private void goodG2B1()
{
    StringBuilder data;
    if (privateReturnsFalse())
    {
        /* INCIDENTAL: CWE 561 Dead Code, the code below will never run
        * but ensure data is initialized before the Sink to avoid compiler errors */
        data = null;
    }
    else
    {
        /* FIX: hardcode data to non-null */
        data = new StringBuilder();
    }

    if (privateReturnsTrue())
    {
        /* POTENTIAL FLAW: null dereference will occur if data is null */
        IO.writeLine("" + data.length());
    }
}
```

Figure 13: An inter-procedural example of spurious data-flow paths identified by **order sanitizer**

```java
public void bad() throws Throwable
{
    StringBuilder data;

    /* POTENTIAL FLAW: data is null */
    data = null;

    /* serialize data to a byte array */
    ByteArrayOutputStream streamByteArrayOutput = null;
    ObjectOutput outputObject = null;

    try
    {
        streamByteArrayOutput = new ByteArrayOutputStream() ;
        outputObject = new ObjectOutputStream(streamByteArrayOutput) ;
        outputObject.writeObject(data);
        byte[] dataSerialized = streamByteArrayOutput.toByteArray();
        CWE476_NULL_Pointer_Dereference__StringBuilder_75b_badSink(dataSerialized);
    }
    catch (IOException exceptIO)
    {
        IO.logger.log(Level.WARNING, "IOException in serialization", exceptIO);
    }
```

Figure 14: An example of spurious data-flow paths identified by **reachability sanitizer**

```
27 private boolean privateReturnsTrue()
28 {
29     return true;
30 }
31
32 private boolean privateReturnsFalse()
33 {
34     return false;
35 }
36
37 public void bad() throws Throwable
38 {
39     String data;
40     if (privateReturnsTrue())
41     {
42         /* POTENTIAL FLAW: data is null */
43         data = null;
44     }
45     else
46     {
47         /* INCIDENTAL: CWE 561 Dead Code, the code below will never run
48         * but ensure data is inititialized before the Sink to avoid compiler errors */
49         data = null;
50     }
51
52     if (privateReturnsTrue())
53     {
54         /* POTENTIAL FLAW: null dereference will occur if data is null */
55         IO.writeLine("" + data.length());
56     }
57 }
```

Figure 15: An example of spurious data-flow paths not identified by LLMSAN

B.2 Unidentified Spurious Data-flow Paths

Consider the program in Figure 15. The model `claude-3` infers the data-flow path from (`data`, ℓ_{49}) to (`data`, ℓ_{55}), while the function `privateReturnsTrue` always returns `false`, implying that the data-flow path is spurious. The prompts of the functionality sanitizer and reachability sanitizer do not contain the function body of `privateReturnsTrue`, which makes LLMSAN fails to detect the underlying hallucination.