`LineBreaker`: Finding Token-Inconsistency Bugs with Large Language Models

Hongbo Chen*†, Yifan Zhang*‡††, Xing Han*§, Tianhao Mao†, Huanyao Rong †,
Yuheng Zhang ‖, XiaoFeng Wang **, Luyi Xing † ¶, Xun Chen ‡‡ ††, Hang Zhang †
†Indiana University Bloomington, USA ‡San Diego State University, USA
§The Hong Kong University of Science and Technology, China ¶University of Illinois Urbana-Champaign, USA
‖Tsinghua University, China **Nanyang Technological University, Singapore ‡‡Independent researcher
{hc50, tianmao, ronghua, hz64}@iu.edu, yzhang24@sdsu.edu, xhanbg@cse.ust.hk, zhangyuh25@mails.tsinghua.edu.cn,
xiaofeng.wang@ntu.edu.sg, lxing2@illinois.edu, xunchen@outlook.com

***Abstract*—Token-inconsistency bugs (TIBs) involve the misuse of syntactically valid yet incorrect code tokens, such as misused variables and erroneous function invocations, which can often lead to software bugs. Unlike simple syntactic bugs, TIBs occur at the semantic level and are subtle - sometimes they remain undetected for years. Traditional detection methods, such as static analysis and dynamic testing, often struggle with TIBs due to their versatile and context-dependent nature. However, advancements in large language models (LLMs) like GPT-4 present new opportunities for automating TIB detection by leveraging these models' semantic understanding capabilities.**

This paper reports the first systematic measurement of LLMs' capabilities in detecting TIBs, revealing that while GPT-4 shows promise, it exhibits limitations in precision and scalability. Specifically, its detection capability is undermined by the model's tendency to focus on the code snippets that do not contain TIBs; its scalability concern arises from GPT-4's high cost and the massive amount of code requiring inspection. To address these challenges, we introduce `LineBreaker`, a novel and cascaded TIB detection system. `LineBreaker` leverages smaller, code-specific, and highly efficient language models to filter out large numbers of code snippets unlikely to contain TIBs, thereby significantly enhancing the system's performance in terms of precision, recall, and scalability. We evaluated `LineBreaker` on 154 Python and C GitHub repositories, each with over 1,000 stars, uncovering 123 new flaws, 45% of which could be exploited to disrupt program functionalities. Out of our 69 submitted fixes, 41 have already been confirmed or merged.

***Index Terms*—Semantic Bug, Large Language Model for Security, Token-inconsistency Bug, Logic Bug, Bug Detection**

I. Introduction

Semantic bugs can cause significant damage in real-world systems, leading to unexpected behaviors, security vulnerabilities, and operational failures. Yet, these bugs are notoriously difficult to detect because they often require a deep understanding of program logic and context. In this work, we focus on the automatic and scalable detection of Token-Inconsistency Bugs (TIBs)[1], a subclass of semantic bugs which is challenging for existing approaches.

Token-Inconsistency Bugs. TIBs are introduced when syntactically valid code tokens — such as variables, calls, and operators — are placed incorrectly, leading to subtle yet impactful flaws. For example, Listing 1 demonstrates a TIB detected by our tool in WolfSSL - a security-sensitive project with over 2.3K GitHub stars, where a redundant `sizeof()` call leads to incorrect buffer size and memory errors. In Figure 1, `quote()` mistakenly takes the argument `query` rather than `params`, disrupting normal program logic. These syntactically valid but semantically incorrect bugs are frequently introduced by developers [62], leading to several security consequences such as control/data flow integrity violations, denial of service, *etc.* [34], [28], evidenced by CVEs assigned to TIBs (*e.g.,* CVE-2022-30780, CVE-2022-23574, and CVE-2020-8889).

```
socklen_t xSize = sizeof(struct freertos_sockaddr);
xSocket_t xClientSocket = NULL;
struct freertos_sockaddr xRemoteAddress;
...
FreeRTOS_bind(xClientSocket, &xRemoteAddress,
    sizeof(xSize) xSize );
```

Listing 1: An example TIB: `sizeof(xSize)` in red is buggy; the correct code `xSize` is in green.

Difficulties in TIB Detection. Despite their security consequences, TIBs often elude conventional detection methods. As in Figure 1, although a human reviewer might identify the misplaced pair (**`query`** and **`params`**`_quoted`) from the program's context (`path` vs. `path_quoted` and `query` vs. `query_quoted`), the buggy code lacks clear-cut patterns as observed in vulnerabilities like use-after-free. Static analysis struggles to capture such context-dependent bugs without explicit rules, and dynamic testing (*e.g.,* fuzzing) may fail due to limited code coverage or subtle bug symptoms (*e.g.,* logic errors may not cause program crashes that are captured by fuzzers). As a result, TIBs can remain undetected for years. The TIB shown in Figure 1 had persisted in a popular GitHub repository for seven years [18], and the bug in Listing 1 went unnoticed for four years.

LLM-based TIB Detection & Challenges. Human developers may identify TIBs by leveraging their capability to predict the "correct" code token based on the program's context, such as the correlations between argument names and function names

*Equally contributed to this work.
†† These two authors were in Samsung Research America.
[1]To differentiate between lexical tokens in programming languages (e.g., variable names) and basic text units in natural language processing (e.g., subwords), we use the terms "code token" and "text token" respectively.

Prompt: Find bugs in this Python code snippet.

```python
def quote_url(url_text):
    (scheme, netloc, path, params, query, fragment) =
        six.moves.urllib.parse.urlparse(url_text)
    # netloc_quoted = quote(netloc)
    path_quoted = quote(path)
    params_quoted = quote( query params )
    query_quoted = quote_plus(query)
    fragment_quoted = quote(fragment)
    url_quoted = six.moves.urllib.parse.urlunparse(...)
    return url_quoted
```

Output by ChatGPT-4: The code snippet you provided has a couple of issues that could be causing problems:

1. **Incorrect usage of the `quote` function for `params`**: In the code, `params_quoted = quote(query)`. This line mistakenly passes the `query` to the `quote` function instead of `params`. It should be `params_quoted = quote(params)`.

2. **Potential Misuse of `quote` vs. `quote_plus`**: Using `quote_plus` for the query is appropriate, but if any other component requires handling spaces as +, then `quote_plus` should be used accordingly. Conversely, if spaces should be encoded as `%20` in the query, then `quote` should be used instead of `quote_plus`

Another false alarm and a corrected code snippet are truncated.

Figure 1: An Example of ChatGPT detecting an existing bug. The variable `query` in the red rectangular is buggy.

in Figure 1 and between variable and function semantics in Listing 1. Recent advances in large language models (LLMs) offer a similar predictive capacity: given a partial code, the LLM can predict the most likely subsequent tokens. Any inconsistency between the actual code and the prediction result could indicate a potential TIB. Moreover, unlike previous ML-based semantic bug detectors that need training on datasets tailored to specific programming languages [8], [64], [53], LLMs are pre-trained on massive corpora, providing them with *generality* across different languages and *flexibility*, often requiring little to no additional training. For example, on our constructed benchmark dataset, which includes 100 synthesized TIBs in 5.7K lines of Python code, GPT-4 can capture up to 72% of them with proper prompts (§II-C).

Despite promising preliminary results, LLMs are not ready for comprehensive, highly scalable, and reasonably accurate TIB detection. Current models typically restrict the input/output token length, making it difficult to incorporate the entire code base into them. For example, Pytorch has roughly 1.4 million lines of code, which can overwhelm GPT-4o's 128k context length. However, slicing large codebases and iteratively prompting capable off-the-shelf LLMs can be prohibitively expensive (*e.g.,* the API cost increases quadratically with code size under an LLM-based detector, FLAG [5]).

Furthermore, LLMs' answers can be error-prone, as pointed by OpenAI [4] and corroborated by our measurements, leading to many false reports in TIB detection; half of the functions sampled from popular GitHub repositories are marked "buggy" by GPT-4. Often, these reports occur because the model is *distracted* by code contexts unrelated to the targeted flaws [60]. For example, in Figure 1, GPT-4 flags `quote_plus` incorrectly, probably due to it being distracted by `plus`.

Our Solution. To address those challenges, we introduce *lightweight* models for TIB detection and *cascade* models of different sizes, capabilities, and efficiency in a unified analytical model to seek a balance between accuracy, cost, and effectiveness through optimal selection (§III-A). We also improve LLMs' performance on TIB detection using meticulously designed prompt structures (§III-C) and token generation algorithms (§III-B), as motivated by the measurement.

Exploring the design space through this analytical model, we propose `LineBreaker`, a TIB detection pipeline (§III). It first performs static analysis, enumerating code tokens that may involve TIBs. Then, `LineBreaker` utilizes lightweight language models to filter potential TIBs locally. We compare predicted tokens with the original tokens, and any difference indicates a potential TIB worth further inspection. Applying such a filter several times efficiently excludes most TIB-free positions, boosting the TIB density and detection precision. Finally, the remaining suspicious TIB sites are uploaded to a SOTA model for in-depth analysis. This cascaded design significantly reduces costs, since smaller, faster, and cheaper models filter out most faults. It also improves the precision of TIB detection, as the final SOTA model can thoroughly analyze a much smaller yet more relevant set of potential buggy locations, mitigating the distraction problem.

Findings. We extensively evaluated `LineBreaker` on large-scale real-world datasets. `LineBreaker` successfully discovered 123 new TIBs in popular C and Python repositories on GitHub with more than 1K stars. We submitted 69 pull requests, and developers have acknowledged 41 of them. `LineBreaker` also achieves an improved precision (up to 36.36%) compared to an existing technique for detecting semantic bugs in real-world code (only 12.0% [6]). It also maintains practical monetary cost, spending only $92. Furthermore, the explanation provided by the LLM pipeline enables a human expert to conveniently identify the true bugs in the filtered code snippets. We responsibly reported all bugs and vulnerabilities to the relevant developers.

Contributions. We summarize our contributions as follows.

• **Study** (§II). We report the first comprehensive measurement study on the capabilities of various LLMs in TIB detection, shedding light on their strengths and limitations.

• **System** (§III). Based upon our measurement study and analytical model, we develop `LineBreaker`, a cascaded pipeline integrated with new mechanisms to enhance LLMs' performance in TIB detection. `LineBreaker` is both practical and efficient for various languages. The `LineBreaker` is publicly available.[2]

• **Findings** (§IV). We use `LineBreaker` to perform the first TIB detection in real-world code at scale, detecting 90 and 33 previously unknown TIBs, respectively, from popular Python and C repositories. 45% of them have various security implications. We have submitted 69 fixes, of which 41 have been confirmed or merged.

[2]https://github.com/Gao-Chuan/LineBreaker

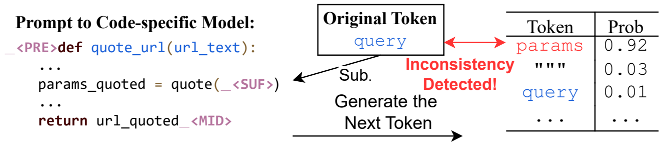

Figure 2: TIB detection using code-specific LLM.

• **Dataset**. We curated a comprehensive dataset of TIBs from historical bug fixes, new bugs discovered by `LineBreaker`, and our synthesized ones. We will release it for future research.

II. Measurement Study

To optimally leverage LLMs for TIB detection, it is essential to understand their performance in this specific task. In this section, we comprehensively evaluate the performance of various language models, encompassing the SOTA model and smaller open-source models designed for coding tasks.

A. Language-Model based TIB Detection

Instruct Large Language Models. Represented by OpenAI's GPT-4 model family [4], instruct models are trained on vast, diverse corpora. This allows them to process tasks effectively in various domains, including software security. However, they are often proprietary (*e.g.,* GPT-4) and expensive to use at scale, restricting their adaptability and making experimental modifications infeasible. Furthermore, these models are also prone to producing inaccurate or fabricated responses due to distraction and well-documented "hallucination", leading to a high false rate in TIB detection. The false positive case in Figure 1 exemplifies this concern.

Code-specific Language Models. Recent efforts have produced models specialized for code-related tasks, such as code generation and completion. These models share a similar transformer-based architecture that generates token probabilities based on context. CodeBERT is a bidirectional encoder model pre-trained on masked token infilling tasks [24], enabling it to predict masked tokens in the middle of a snippet. However, its context length is strictly limited to 512 tokens, making it impractical for real-world applications. Recent models adopt a decoder-only transformer design, generating tokens sequentially. Despite their unidirectional nature, some are trained on fill-in-the-middle (FIM) tasks, enabling code infilling like CodeBERT. While not as powerful as GPT-4 in some respects, these open-source code-specific models offer broader accessibility and opportunities for customization.

Example. Code-infilling can naturally and novelly support TIB detection by comparing the original tokens against a model's predicted tokens. If the predicted tokens differ, this indicates a potential bug. For instance, when feeding the buggy code snippet in Figure 1 into a code-specific model supporting FIM as in Figure 2, the model recommends `params` as a more contextually appropriate token, strongly suggesting a TIB.

B. Dataset Preparation

Since no dedicated datasets for TIBs exist, we constructed bipartite datasets to evaluate key metrics, including recall and specificity. To mitigate memorization and contamination concerns, we sampled code snippets from popular (>1k stars) Python repositories on GitHub committed after the LLMs' knowledge cutoff date. Functions were grouped by length and proportionally sampled to maintain a comparable context length distribution. The resulting dataset, D, contains 1,000 manually verified functions, all of which are free of TIBs.

We also implement a process akin to prior code token mutation methodologies [55], [51] to introduce synthesized TIBs on the dataset D, obtaining a synthesized dataset D'. D' comprises artificial yet plausible TIBs suitable for, *e.g.,* recall evaluation. This process is initiated by enumerating all permissible code token candidates that could feasibly replace another token without causing syntax errors. Permissible candidates span over four types of code tokens: variable uses, function calls, operators (*e.g.,* $>=$ and `or`), and literals (*e.g.,* integer). We then select a substituted TIB token with slight semantic divergence based on embeddings from MPNet [57], reflecting common confusion between tokens with similar semantics. Function snippet in D' has a single TIB. Three security experts independently validated the datasets and then convened to discuss and resolve all identified inaccuracies.

C. GPT-4's Performance on TIB Detection

To assess the TIB detection capabilities of general-purpose LLMs, we selected GPT-4 models for their superior performance [60], [58], [66] and API's support for JSON output.

Prompt Design. In-context examples do not significantly improve GPT-4's accuracy in identifying vulnerabilities [58]. Therefore, given the diverse nature of TIB patterns, we opted for zero-shot prompts to avoid the influence of potentially irrelevant examples [56]. Specifically, we utilized various templates and conducted multiple rounds of interactive prompts. These templates incorporate various prompt engineering techniques, including Chain-of-Thought (CoT) [65] and cross-examination [19]. Drawing on the prompt engineering methods introduced in [67], we assigned the role of a programmer to the LLM and provided it with an output pattern. Additionally, following the prompt taxonomy and templates proposed in [32], we outlined the workflow steps to offer detailed instructions to the LLM on effectively processing the given prompt.

We illustrate an example of a template configuration in Figure 3, which represents a simplified version of the one ultimately adopted by `LineBreaker`. The system prompt instructs the model to assume the role of a programming language expert (*i.e.,* persona adoption [48]). In the initial round, the model is requested to identify potential buggy code lines and provide explanations. The second round further cross-examines the identified potential TIBs to minimize false positives. Additionally, we incorporate three selectable properties in this round: self-evaluated bug categorization, fix proposal, and token-level predicate. The details of these

Table 1: Analytical properties for TIB detection and inspection in JSON-formatted tool calling.

Property	Abbr.	Type	Description	Purpose	Prompt Tec.
code_line	/	Mandatory	the exact line of code with semantic bug	locating the bug	/
explanation	/	Mandatory	a concise explanation of this bug	reasoning about of the bug	/
token_level	T	Selective	the bug appears is related to single/few tokens	filtering out non-TIB bugs	Cross-exam.
fixed_line	F	Selective	the line with the bug fixed	CoT for token_level and fixing	CoT
category	Ca	Selective	the category of this bug	filtering out non-semantic bugs	Cross-exam.
priority	P	Selective	the priority level of this bug: high/medium/low	filtering out less interesting bugs	Cross-exam.

System: You're a language expert. Your job is to inspect if the code contains any semantic bugs. Semantic bugs are a type of bug that occurs when the code is syntactically correct but does not behave as intended or produces incorrect results. These bugs arise from mismatches between the programmer's intended logic and the actual implementation in the code, where incorrect variable/method name usage or assignment can lead to bugs or vulnerabilities in the program.
Implement your work by the following steps:

1. Check the given code line by line.
2. For each line, extract the variables, calls, operators, and literals in this line.
3. Check whether the extracted variables, calls, operators, and literals are correctly used in the statements.
4. If there is no wrong usage in any line, return the code is correct; otherwise, record each line of the bug and the reason for the bug.

Assume the code is syntactically correct and input parameters to the functions are well-formed and valid. Focus solely on detecting semantic bugs, and ignore other problems. Return your response in JSON format.

User Prompts
Round 1 Properties: Mandatory: {code_line, explanation}
Round 1 User Prompt: {code for detection} Output exact lines of semantic bugs and concise explanations.
Round 2 Properties: Mandatory: {code_line, explanation};
Selective: {fixed_line, token_level, category}
Round 2 User Prompt: Inspect these bugs, excluding 1. incorrect or unlikely bugs; 2. non-semantic bugs. Check the left bugs only break the intended functionality or lead to vulnerabilities. Finish the following tasks: 1. Answer if the bug is related to a single or a few tokens. 2. Using the code and previous contexts, classify the bugs into these categories: Security Vulnerability, Logic Bug, Enhancement, Unexpected Behavior, Symbol Not Defined, Module Not Imported, Bad Smell, Not a Bug, or Others. Note that the snippet is from popular repositories and runs, so correct symbols not defined in the snippet might be defined at other places, and you can regard it as Symbol not Defined. If you assign Others category, explicitly name the category.

Figure 3: Example: a two-round prompt template (1/2FTCa).

properties are summarized in Table 1. Mandatory properties are adopted to identify bugs, while selective properties are employed for filtering. Multi-round prompting enhances effectiveness; for instance, allowing GPT-4 to reflect on its prior results, facilitating the exclusion of potential false alarms (*i.e.,* cross-examination). Depending on the properties required in each round, prompts for querying these properties can be automatically generated under various template setups. If the property in Table 1 is selected in a round, the digits represent the number of the round, and the alphabetic abbreviations represent chosen properties. For instance, the template in Figure 3 is denoted as 1/2FTCa. Please refer to our supplementary material for other prompt templates.

Measurement Results. Our objective is to assess the performance of various prompt templates regarding recall, specificity, and monetary cost. While each function in D' contains only one known TIB, the LLM may identify multiple TIBs within the queried function. We adopt two statistical approaches: 1) counting the number of lines flagged with TIBs. Flagged lines are regarded as positive, as the model is prompted to mark buggy lines; 2) counting the number of entire functions, as the security engineer usually audits the function as a whole. The metrics under these two approaches are subscripted by $_L$ and $_F$, respectively. To reduce the high API costs, we evaluate all templates on subsets of D' and D, each containing 100 cases.

Table 2 presents the measurement results of prompt templates. We observe a high false positive rate when using a single-round template with no selective properties, and GPT-4 is directly asked to identify TIBs. This is primarily due to distractions, as confirmed by selective manual inspection.

First, we unveil the effectiveness of selective properties (in Table 1). Our findings indicate that the token_level property is particularly effective at filtering out non-TIB instances, which are not bugs associated with a few code tokens. This filtering improves line-level specificity by up to 22.6% at the cost of a 1% reduction in recall. Additionally, excluding candidates whose category is not in {Logic Bug, Security Vulnerability, Bad Smell} further enhances specificity. However, filtering based on priority $\neq$ high, while reducing false positives, leads to missed bugs. This trade-off may be attributed to GPT-4's limited understanding of bug severity.

Next, we investigate the impact of iterative rounds. Introducing second and third rounds with varied properties yields mixed results, with cross-examination occasionally reducing false positives. However, adding a third prompt round proves less advantageous regarding specificity, recall, and monetary cost, as additional rounds increase token usage. This can be attributed to superfluous cross-examination, where identified bugs are unnecessarily filtered out. Distraction may also be a potential root cause, as irrelevant information from previous rounds can misguide the LLM. As a result, two-round prompting emerges as the optimal approach.

Furthermore, we evaluate the best overall prompt template, 1/2FTCa, on three models in the GPT-4 family, as shown in Table 3. Interestingly, GPT-4 Turbo slightly outperforms the newer GPT-4o on specificity with triple the cost.

Conclusion. We arrive at two primary conclusions:
(1) Template Selection. The template 1/2FTCa in Figure 3 shows anticipated overall performance. LineBreaker adopted it with suspicious lines highlighted (§III-C).
(2) Limitations of GPT-4 in TIB Detection. Our study reveals that GPT-4 demonstrates relatively high recall for TIB detection. However, its current performance remains insufficient for

Table 2: GPT-4o TIB detection on different prompt templates.

Template	Synthesized $\subset D'$					Bug-free $\subset D$					Cost
	TP_L	FP_L	TP_F	FP_F	Rec.	FP_L	E. FP_L	FP_F	E. FP_F	Spe.	
1	81	203	**72**	28	**72.0**	404	0	29	0	**0.0**	\$0.83
1FT	75	110	71	24	71.0	229	67	14	7	22.6	\$0.88
1/2FT	79	79	67	33	67.0	298	89	24	7	23.0	\$2.11
1/2FT/3P	57	99	55	36	55.0	211	73	20	10	**25.7**	\$3.37
1/2FT/3Ca	72	129	66	31	66.0	245	43	19	10	14.9	\$3.43
1/2FTCa	78	126	**72**	26	**72.0**	294	92	**23**	7	**23.8**	\$2.14
1/2FTCa w/ HL	92	93	**84**	12	**84.0**	174	138	**56**	34	**44.2**	\$2.07

Rec. refers to *recall* at the function-level and Spe. refers to filtering capability (i.e., *specificity*); X_L and X_F represent the numbers counted at the *line*- and *function*-level; FP and E. FP represent the numbers of incorrect bugs reported and *excluded*.

Table 3: TIB detection performance of GPT-4 model family.

Model	Synthesized $\subset D'$					Bug-free $\subset D$					Cost
	TP_L	FP_L	TP_F	FP_F	Rec.	FP_L	E. FP_L	FP_F	E. FP_F	Spe.	
GPT-4o	78	126	**72**	26	**72.0**	294	92	23	7	23.8	\$2.14
GPT-4 Turbo	86	129	**72**	24	**72.0**	256	87	**19**	10	**25.4**	\$6.82
GPT-4o Mini	86	242	64	33	64.0	388	92	29	4	19.2	\$0.19

practical use. Even with the most effective "1/2FTCa" template, a significant portion of TIBs is still missed. Moreover, as TIBs in real-world code are relatively rare, relying solely on GPT-4 for TIB detection would likely result in a high false positive rate due to its limited specificity. As shown by the specificity evaluated on D, many false positive reports persist. Our investigation into some cases reveals that GPT-4 is overly cautious when evaluating software security, resulting in the generation of false alarms. Consequently, more targeted guidance is necessary to enhance the model's performance.

D. Code-Specific Models on TIB Detection

While general-purpose LLMs can also perform code infilling, their usage cost can quickly become prohibitive for large codebases. For instance, GPT-4 costs approximately \$2.1 to infill every code token for a function of 40 lines of code, which increases quadratically with code size. Therefore, we focus on smaller open-source code language models in this evaluation. The example in §II-A indicates that the code-infilling capability of LLMs can also facilitate TIB detection. TIB detection could leverage infilling because the probabilities of correct tokens are significantly higher than those of buggy tokens. We thus evaluate the performance of various code-specific LLMs on TIB detection.

Hardware and Parameters. We conduct the experiments on a Linux server with two AMD EPYC 9124 CPUs and one NVIDIA H100 GPU. The temperature is set to 0.

Infilling Task Description. Code infilling is a foundational capability of these code-specific models, enabling numerous applications (e.g., auto-completion). Here, each infilling task masks one code token and asks the model to predict that missing token. We aim to identify models performing accurately, prompt organizations leading to better results, and potential reasons for successful or failed infilling. Thus, regarding TIB as positive cases, we evaluate three metrics:

- **Specificity** (Consistency). On the TIB-free D, if the model predicts the same as the masked token, we consider it to have no TIB. High specificity indicates fewer false alarms.
- **Recall** (Sensitivity). On the D' of synthesized TIBs, if the predicted token differs from the masked, a potential TIB may exist. High recall indicates more detectable TIBs.
- **Speed**. This measures the time required to complete the infilling task, reflecting scalability.

Similar to previous work [6], we intentionally avoid using metrics like F1 score because the distribution of TIB varies (*e.g.,* on languages), and such distribution is unavailable. Figure 4 summarizes the results. Due to GPU memory constraints, models exceeding 15B are excluded from our evaluation. While our H100 GPU with 80 GB memory can successfully load these models, the practical memory requirements extend beyond simple model loading. The self-attention mechanism, fundamental to transformer inference, induces space-complexity quadratic in the input/output sequence length [33]. For example, the major memory requirements break down as follows for CodeLlama 13B model quantized at FP16: ~24 GB for model weights ($13\times10^9\times2$ bytes) and ~48GB for attention layers of 4,000 tokens ($4000^2\times40$ layers$\times40$ heads$\times2$ bytes). Adding other components (*e.g.,* activation and KV-cache), inference with code length beyond 4,000 tokens could exhaust GPU memory on our system. Therefore, cherry-picking the input context becomes crucial for efficiency. We discuss potential countermeasures in §VI. That said, our solution design (§III) is orthogonal to specific models.

Measurement Results. Our findings reveal that models universally exhibit high recall (~98%). However, specificity varies considerably. StarCoder2 frequently fails to restore the original tokens, whereas DeepSeek, CodeGemma, and Qwen models achieve a specificity of 70–80%. Larger models in the same family generally exhibit better specificity and recall but experience nonlinear slowdowns due to GPU memory constraints when handling extended contexts.

A key factor affecting specificity is that models usually overlook the validity of infilling results. Some models insert irrelevant text, such as comments or extraneous statements, because they are unaware of legitimate code token syntax.

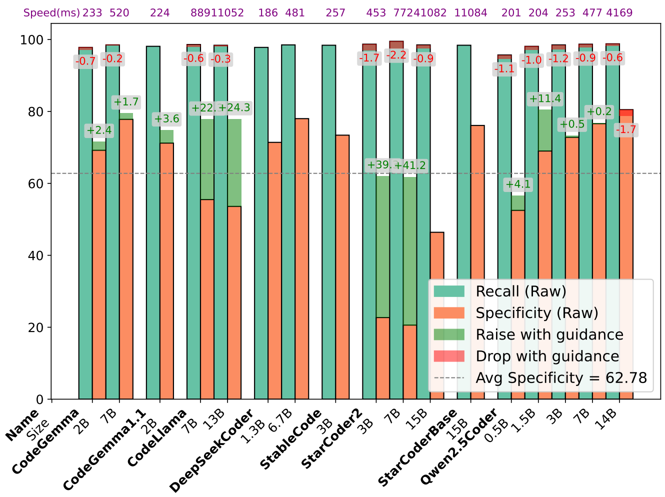

Figure 4: Code-specific models' infilling performance.

Table 4: Infilling performance under different contexts.

Model		Function		File		Sliced File	
Name and Size	Length	Spe.	Time	Spe.	Time	Spe.	Time
CodeGemma 2B	8K	71.2	441	19.8	**5621**	56.9	1442
StableCode 3B	16K	73.4	**257**	68.5	10657	71.2	**989**
DeepSeek 6.7B	16K	**78.0**	481	**85.8***	9340	**77.4**	1638
Code Llama 13B	100K	53.6	11052	60.1*	8355	57.3*	5075
Qwen2.5 7B	32K	76.6	477	85.6	8773	75.1	11129

* Some tasks are excluded due to runtime errors (e.g., insufficient GPU memory).

Besides, function calls exhibit lower specificity than variables, operators, or literals. We hypothesize that these results are from missing function definitions or external context, which models cannot infer from the snippet alone.

Constrained Token Generation. Some frameworks can impose constraints on generated tokens [68]. We also integrated Guidance [1] into our experiments and denote the metric changes on the bars in Figure 4. By restricting generated tokens through regular expressions or lists of candidates, specificity significantly improves (especially for Code Llama and StarCoder2), albeit with a slight reduction in recall. However, some models, like DeepSeek Coder, are incompatible with Guidance due to byte encoder errors. Guidance also introduces additional runtime overhead.

Context Dependency. We also explored different context organizations: 1) Function: only the function definition, which is used previously in Figure 4; 2) File: the entire file; 3) Sliced File: function definition plus global variables and function declarations in the file.

Table 4 shows that richer contexts can improve specificity by up to 9% (e.g., DeepSeek Coder, Qwen2.5 Coder, Code Llama). At the same time, the effect on recall is not significant and is omitted from the figure. However, extensive context truncation degrades performance for smaller models like CodeGemma with stricter token length limits. We also note that extended inputs incur slowdowns in inference, and may even raise runtime errors due to insufficient GPU memory (*e.g.,* Code Llama 13B). Thus, context engineering should be paired with various models for better overall performance.

Impact on Nearby Tokens. TIB can negatively affect the infilling task in its nearby context. When a TIB presents in a line, the specificity of the infilling tasks on other positions in the same line drops 6%-7%. We further design a highlighting prompt in §III-C to enhance the robustness of our pipeline. A potential explanation is that TIBs increase the perplexity, leading to larger uncertainty in token prediction [63]. This also implies that inconsistent infilling results conducted on single tokens may unveil bugs related to multiple code tokens.

Conclusion. We highlight these conclusions from our study: *(1) High Recall but Mixed Specificity.* Most models detect more than 97% of synthesized TIBs, showing strong sensitivity to TIB. This could further foster TIB detectors where only a few TIBs can evade. However, many models produce 15–30% inconsistencies on TIB-free samples, even with constrained token generation and rich contexts. As no single model–prompt configuration consistently excels across both performance metrics, TIB detection should not rely on a single model.
(2) Scalability Limitations on TIB Detection. Despite their ability to filter out numerous true negatives, plenty of false alarms can be fired. However, as TIBs in real-world code might account for less than 1%, using a single model can result in poor precision. Even though constrained token generation and richer context can enhance specificity and mitigate this concern, the enhancement is neither universal to arbitrary models nor enough for a scalable detector. We need additional measures to overcome this concern.

III. Design and Implementation of `LineBreaker`

Inspired by the measurement, we present `LineBreaker`, a system to detect TIBs effectively and efficiently in various programming languages. Our objective is to build a TIB detector that is scalable, flexible, and language-agnostic. Motivated by the examples and measurements, we believe that LLMs are well-suited for TIB detection, potentially outperforming other techniques for three reasons. First, transformer-based models naturally handle context-dependent TIB. Semantically incorrect code tokens, contextualized by surrounding code, typically have a lower probability, plus LLMs could "understand" code semantics. Thus, TIBs are detectable via infilling- or prompt-based methods. Second, LLM-based approaches are more flexible and general than specialized ML-based solutions, which often rely on strict assumptions. For example, syntactically similar functions must exist in target codebases [6]. Because LLMs are trained on massive multilingual corpora, they inherently support different languages. Finally, LLMs can further assist developers in fixing bugs by offering interactive, natural-language explanations.

However, the measurement study (§II) also reveals shortcomings. GPT-4 and smaller code-specific models alone cannot reliably handle TIB detection at scale because of excessive cost and/or impractical accuracy. Fortunately, Such weaknesses can be mitigated with the *synergy of SOTA and lightweight models*, leveraging the strengths of each category.

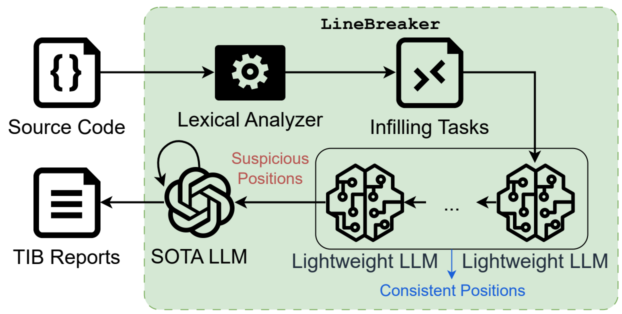

Figure 5: Pipeline overview of `LineBreaker`.

The high expenses and FP rate of GPT-4 can be contained if smaller and cheaper models eliminate most TIB-free tokens in advance at lower costs. Meanwhile, suspicious locations identified by these smaller models can be verified further by GPT-4, reducing the distractions to GPT-4. We also design new algorithms and adaptive prompts to facilitate TIB detection.

`LineBreaker`'s Pipeline. We depict `LineBreaker`'s pipeline in Figure 5. Initially, a lexical analyzer processes the code, compiling a comprehensive set of infilling tasks with code tokens being masked and token type information. Then lightweight LLMs deployed locally conduct the infilling task, sequentially filtering out most TIB-free tasks and drastically reducing the search space. It is worth mentioning that `LineBreaker` involves several different local models. Next, following a carefully engineered prompt template, `LineBreaker` prompts GPT-4 for further bug inspection. GPT-4's responses undergo an automated filtration and are then aggregated into TIB reports for manual review.

By cascading models in this manner, `LineBreaker` significantly reduces false positives. Early stages remove most negatives before GPT-4 is invoked, avoiding distractions and lowering costs (§III-A). Meanwhile, strategic token-generation control in the local models boosts specificity (§III-B), ensuring that most true negatives are discarded early. Finally, GPT-4 provides in-depth analysis of suspicious tokens, highlighting prompt engineering (§III-C). We evaluate the effectiveness of these new techniques later in §IV-A.

A. Cascaded Detection

The measurements in §II indicate that no single model can simultaneously deliver high recall, high specificity, and fast throughput for TIB detection. `LineBreaker` employs a cascaded design that combines multiple models to balance these objectives. We formalize our approach and discuss the rationale behind model selection.

Problem Formalization. Suppose our pipeline consists of n stages, each equipped with a different language model. Let p_i denote the true negative rate (*i.e.,* TNR or specificity) of stage i, q_i the true positive rate (*i.e.,* TPR or recall), and t_i the throughput (*i.e.,* execution speed). Let N_i represent the number of potential TIB cases to be processed at stage i, and ϵ_i represents the ratio (*i.e.,* density) of actual TIB instances among N_i after stage i, where ϵ_0 being the initial TIB density before detection. We can derive the number of true/false positives/negatives at each stage:

Table 5: TIB detection cost under different parameters.

ϵ_0	$pi = 0.8$				$pi = 0.6$		
	$n=2$	$n=3$	$n=4$	$n=5$	$n=3$	$n=4$	$n=5$
10^{-2}	365.4	**218.6**	250.5	316.8	388.5	**330.4**	351.7
10^{-3}	295.7	76.4	38.6	**37.1**	247.8	119.3	**72.3**
10^{-4}	288.7	62.1	17.4	**9.1**	233.7	98.2	**44.4**

$$N_{TN,i} = N_i p_i (1-\epsilon_{i-1}) \quad N_{TP,i} = N_i q_i \epsilon_{i-1}$$
$$N_{FP,i} = N_i (1-p_i)(1-\epsilon_{i-1}) \quad N_{FN,i} = N_i (1-q_i)\epsilon_{i-1}$$

After stage i, TN cases $N_{TN,i}$ are correctly excluded, while FN cases $N_{FN,i}$ are TIBs missed at this stage. TPs and FPs are escalated to the next stage, remaining $N_{i+1} = N_{TP,i} + N_{FP,i}$. Thus, the total number of missed TIBs M across the entire pipeline and the cumulative execution time T are:

$$M = \sum_{i=1}^{n} N_i (1-q_i)\epsilon_{i-1} \qquad T = \sum_{i=1}^{n} N_i t_i$$

Additionally, the TIB density after stage i can also be derived:

$$\epsilon_i = \frac{N_{TP,i-1}}{N_{TP,i-1} + N_{FP,i} - 1} = \frac{q_i \epsilon_{i-1}}{1 - p_i - \epsilon_{i-1} + (p_i + q_i)\epsilon_{i-1}}$$

For an n-stage pipeline, the aggregated precision of all models before the final stage is then ϵ_{n-1}. The final stage adopts state-of-the-art GPT-4, which processes N_{n-1} TIB candidates, with the specificity p_n and recall q_n.

We break down the cost into four factors for scalable TIB detection: 1) the cost of computing resources, C_{comp} per unit time; 2) the cost of API invocations, C_{api} per call for commercial LLMs; 3) the penalties associated with missed bugs, C_{miss}, due to potential financial loss; and 4) manual inspection cost, C_{check} per TIB. We thus define the total cost

$$C = C_{api} N_{n-1} + C_{comp} T + C_{miss} M + C_{check} N_n$$

While the values of C_{api} and C_{comp} can be obtained from service providers, C_{miss} and C_{check} are configurable. This formula transforms the model selection problem into an optimization problem to minimize the total cost.

`LineBreaker`'s Pipeline Establishment. First, we fix the constants in the cost formula. We set C_{miss} at \$500, which is preservative according to [25]. Given the example in Figure 1 that inspecting the report is eased by GPT-4, we set C_{check} to \$2. Based on the measurement (§II-C) and online data [35], we set C_{api} and C_{comp} as \$0.025 per API call and \$2.49/hour.

Second, we discuss the number of optimal stages under an experimental configuration. Fixing GPT-4 as the final stage because its reasoning and natural-language feedback are instrumental for security experts. We then simulate how many local lightweight LLMs should precede GPT-4. TIB density ϵ_0 varies concerning the dataset (*e.g.,* of different languages), so we pick representative values according to previous findings, where the number of bugs per 1,000 lines of code ranges from 0.1 to 25 [45]. Borrowing average $q_i = 0.98$, selecting average and best p_i as 0.6 and 0.8 from Figure 4, we calculate the cost of our pipeline in Table 5. We observe that 1) cascading more does not always derive cost-efficiency because missing bug penalties accumulate, and the marginal benefit shrinks; 2) enhancing specificity reduces the cost significantly; 3) positioning the faster model further to the SOTA reduces computation cost, although local model permutation is irrelevant to the overall accuracy. These findings motivate us to construct better algorithms on token generation with enhanced specificity, and guide our model selection(see §IV-B).

B. Controlled Token Generation

As measured in §II-D, local LLMs generate tokens in an unregulated manner, leading to many incorrect infilling results (*e.g.,* generating a docstring where a variable is expected).

Decoder-only models like Code Llama do not constrain which *type* of code token to generate. They blindly select most probable tokens sequentially until a special end-of-sequence symbol or length limit is reached. A naive approach might check if the original token ranks high in probability. However, once the model deviates from the correct token at any intermediate step, subsequent tokens are conditioned on the wrong prefix, rendering the original token improbable. For example, when a variable name is masked but the model begins to generate documentation (*e.g.,* starting with `"""`), subsequent generation follows up this pattern rather than the expected variable name. Existing frameworks like guidance can help but fail to constrain the generation of specific models. Our solution is to inspect the probabilities of generated tokens and apply syntactic constraints on the output.

The high-level idea is to select tokens that adhere to the original code tokens in each iteration and inspect the probabilities to judge consistency. As shown in Figure 5, the lexical analyzer aids infilling tasks construction with syntactical knowledge. It inserts special tokens indicating the infilling position in each function, as shown in Figure 2. Auxiliary information including the original tokens, token type (*e.g.,* variable and literal) is also recorded to constrain the generation in the subsequent steps. Thus, `LineBreaker` selects each generated token by verifying whether it remains syntactically valid and aligns with the original code token. For example, valid variable names in Python can only contain letters, underscores, and non-beginning digits.

Algorithm 1 inspects the probabilities of the generated candidates in each step. For each generated *token*, `LineBreaker` first checks if the token is syntactically valid. Then, if *leftTokens* starts with the generated *token* (*i.e.,* the generation follows the original code token), it updates *leftTokens* and continues the next generation. Otherwise, this generation is penalized for not being consistent with the code token. In cases where the probability *prob* is greater than a threshold *probThresh*, meaning there is another, more probable, and valid token that can deviate from the generation, the check directly returns *False*. Nevertheless, when deviation happens

Algorithm 1: Decoder-only Model Consistency Check

```
Input: maskedCode, originalTokens, probThresh, rankThresh,
Output: consistencyFlag
rankSum ← 0;
generatedTokens, leftTokens ← [], originalTokens;
while leftTokens.length >0 do
    nextTokenProbList ← top_k(CodeLlamaPredict(maskedCode,
      generatedTokens));
    foreach token, prob in nextTokenProbList do
        if validateToken(token, typeof(token)) then
            if leftTokens.startswith(token) then
                leftTokens.remove(token)
                generatedTokens.append(Token)
                break to generate the next token;
            else
                if prob >probThresh then
                    return False;
                rankSum ← rankSum + 1;
        else
            continue to the next iteration;
    if rankSum >rankThresh then
        return False;
return True;
```

but *prob* is less than *probThresh*, it adds one to *rankSum*, penalizing the original code token for not being the most probable generated *token*. Finally, consistency is determined by comparing *rankSum* with a threshold.

C. Highlighting Inconsistent Positions

As observed in §II-C, GPT-4 can produce many FN cases, especially with lengthy code snippets that distract the model with irrelevant context [60]. Thus, we design a prompt engineering technique emphasizing the suspicious tokens identified by lightweight LLMs. Based on the most effective prompt template we measured, `LineBreaker` instructs GPT-4: "Also, pay additional attention to these lines: `{suspicious_lines}`". Rather than the suspicious token, the whole line is highlighted because of our findings in §II-D, which indicate that a TIB at one position can increase perplexity for nearby tokens on the same line. Thus, inconsistencies may also imply TIBs at another position in the same line.

D. Implementation

We implement `LineBreaker` in Python with roughly 7K lines of code. It relies on tree-sitter [3] to preprocess the source code, as it can parse multiple programming languages. `LineBreaker` sources models from Hugging Face, and utilizes the transformer library to generate tokens. Algorithm 1 is based on PyTorch framework. `LineBreaker` also relies on OpenAI's official library to invoke SOTA LLMs deployed on the cloud. Besides Python integration, we support the C language with 500 additional lines to demonstrate generality.

IV. Evaluation

In this section, we first conduct controlled experiments on our prepared datasets (§II-B). This is to (1) evaluate the effectiveness of individual components in our design and (2) compare `LineBreaker` with existing bug detectors. After

Table 6: Performance of Selected Models with Algorithm 1.

Model	*rankT*=0		*rankT*=1		*rankT*=2		Time
	Rec.	Spe.	Rec.	Spe.	Rec.	Spe.	
StarCoder2 3B	96.7	74.6	91.0↓	82.4↑	87.1	85.2	727
Code Llama 7B	97.5	65.8	92.2↓	75.3↓	89.6	78.5	1601
DeepSeek Coder 6.7B*	97.4	79.7	92.3↓	86.6↑	89.3	89.2	1606

* Guidance does not support DeepSeek models, so the metrics are compared with Raw setup. Others are compared with guidance in Figure 4. *rankT* referes to *rankThreshold*.

that, we evaluate LineBreaker on real-world code repositories and demonstrate its ability to discover new bugs. The experiment uses the same platform as in §II-D.

A. Controlled Experiments

We continue using dataset D (TIB-free) and D' (synthesized TIBs), measuring recall and specificity as in §II. The evaluation also helps us concretize some parameters, which will be used in a scalable real-world TIB detection in §IV-B.

Controlled Token Generation Algorithm. Next, we evaluate Algorithm 1 on decoder-only models. We evaluate §II-D again with different *rankThresh* (*rankT*) on decoder-only models, and present the results of selected models in Table 6. We set *probThresh* = 0.9 after observing minor variations in performance at different probability thresholds. Our algorithm enables constrained token generation for models not supported by guidance, such as DeepSeek Coder. While our algorithm incurs a modest speed penalty, it notably boosts specificity that offsets minor recall loss, especially when *rankthresh* ≤ 1. The recall and specificity of our algorithm are at least comparable to guidance with selected parameters. Since TIBs are rare in real-world code, we choose *rankThresh* $= 1$, to reduce false positives and improve overall precision.

Highlighting. To measure the effectiveness of the highlighting technique (§III-C), we integrate it into the best prompt template. The results are listed in Table 2 along with the results in measurement §II-C. The highlighted lines also include up to four randomly selected lines in each sample to simulate false positives propagated from the previous models. Considering both specificity and recall, highlighting drastically enhances the performance. This is because GPT-4 refocuses on the suspicious line, mitigating the distraction problems. Importantly, highlighting is only feasible when local models have already identified specific suspicious tokens.

B. Real-world TIB Detection

To demonstrate its real-world impact, we evaluate LineBreaker on Python and C repositories on Github, which are (1) popular (with 1k+ stars), and (2) actively maintained. The results are summarized in Table 7.

New Bugs. LineBreaker successfully found 123 new TIBs, we have submitted 69 bug fixes as pull requests, of which 41 have been merged or confirmed by developers.

Precision and Practicality. LineBreaker achieves a bug detection precision of 23.5%-36.3%, significantly outperforming existing semantic bug detection works [6] on real-world projects (12.0%). We believe this precision is also practical, especially for developers who prioritize security, as suggested by

Table 7: Summary of real-world TIB detection results.

	Python Repos	**C Repos**
Repositories	80	74
# Submitted PR	55	14
# Merged / Confirmed PR	32	9
# Functionality Bugs	37	7
# Security Bugs	8	3
# Bad Smell	48	20
# Inconsis. Token after Local models	2,679	733
# Reports from GPT-4	1,082	473
# Reports after Filtering	314	77
# Correct Reports (may contain $\geq$ 1 bugs)	74	28
Precision (%)	23.57%	36.36%

the recent study [10]. Moreover, LLM's detailed and intuitive explanations make our bug report easy to inspect. Actually, each report takes only about *one minute* for our team members to review. In other words, by inspecting LineBreaker's output, one can *find one true TIB roughly every four minutes*.

Model Selection. As we conduct the evaluation on popular repositories, we assume ϵ_0 ranges around 10^{-3} [45]. Thus, our pipeline consists of three local models plus GPT-4o according to §III-A. Such a configuration misses fewer TIBs with additional but acceptable human inspection cost and slight accuracy reduction. The optimal selection of three local models is Qwen2.5 Coder 1.5B, CodeGemma 7B, and Qwen2.5 14B. However, as fewer models were available when we conducted this evaluation, we utilized StableCode and CodeLlama models. GPT-4o was used as the state-of-the-art LLM to inspect the filtered results. We use the prompt template shown in Figure 3 plus suspicious lines highlighting. New models such as Llama 3 have been recently released. However, their APIs previously offered limited support for JSON output, which makes it difficult to integrate into an automated detection pipeline. We leave the measurement and integration of these models as future work.

We thus estimate LineBreaker's end-to-end performance using the formulas in §III-A, deriving a theoretical accuracy of 10.5%. Such value is comparable to our evaluations in §IV-B. Notably, relying solely on GPT-4 as a TIB detector yielded a precision of only 0.7%. In contrast, LineBreaker effectively amplifies the density of TIB by 105 times, making scalable TIB detection and manual inspection practical.

Detecting TIBs in Real-world Repositories. LineBreaker processed 80 popular Python repositories from GitHub. Initially, 80K infilling tasks are derived from static analysis. After dropping the consistent ones and tasks associated with very long code snippets (*e.g.,* configuration and data files), 2,679 inconsistencies are left after local model filtering. Escalating these functions containing inconsistencies to GPT-4 for TIB detection, 1,082 reports remained, of which 314 persisted after filtering based on the properties mentioned in §II-C. Three security experts spent 12 man-hours totally inspecting these reports, confirming that 74 reports are true positives. This resulted in a precision of 23.57% on Python repositories. The detection requires less than 20 machine hours in total and less than \$40 to query GPT-4o APIs. With a server cost of \$52, the total cost to find these TIBs is less than $100.

We also experimentally test LineBreaker on C code to demonstrate its generality. It generates infilling tasks according to abstract syntax tree AST similarity strategies [11], [29] following a previous research [6]. We observe an enhanced accuracy of LineBreaker on C repositories, 36.36%. A potential reason is that Code Llama performs significantly better on C, filtering out 97.1% consistent tokens. A potential explanation is that C language induces fewer defects than Python [54], making masked tokens more predictable. Besides, we prototype support for the Rust programming language and experimentally discover one TIB in a repository. These evaluations showcase the generality of LineBreaker across various languages, backing its scalable deployment.

Repository Exclusion. We intentionally exclude 10 repositories from our dataset due to their negative impact on LLM's performance: (a) Projects written in outdated languages (*e.g.,* Python 2), which could confuse LLMs as they are trained on a more recent code corpus; (b) AI-related projects (*e.g.,* chat-langchain [2]). We observed downgraded LLM performance for them, likely because their fast-evolving AI techniques surpass LLMs' training corpus; (c) Security-related or low-level system projects (*e.g.,* hosts [59]), whose special or unusual functionalities often lead to reduced specificity. For example, exploit generators intentionally include insecure code, rendering respective infilling tasks regarded as inconsistent by local models, or reported as vulnerable by GPT-4. We believe such repositories could be better handled with fine-grained prompt engineering, which we leave as future work.

C. Study of the Discovered TIBs

LineBreaker successfully identifies multiple real-world TIBs in impactful software projects, some of which lead to severe security consequences like memory corruption and denial-of-service attacks. Notably, a director of wolfSSL [69] - a popular and security-critical library - scheduled a meeting with us to discuss our found vulnerabilities and fixes. We showcase some TIBs we found in this section.

```
switch (optionId) { ...
case InterpreterThreadFrameStackSize:
  return s_threadFrameStackSize;
case InterpreterThreadExceptionFlowSize:
  return ~~s_threadFrameStackSize~~ s_threadExceptionFlowSize ;}
```

Case 1. The second switch-case in the above code snippet wrongly returns the same variable as the first one, where the correct identifier should be s_threadExceptionFlowSize. Since the returned size here will be used for buffer initialization, buffer overflow can happen later, causing severe security consequences such as privilege escalation. We suspect the developer forgot to change the variable name when copying and pasting the return statement.

Case 2. Listing 1 shows a TIB in the security-critical WolfSSL library. The sizeof() function is redundantly applied to xSize, which is already a size-representing variable. This leads to an unexpectedly smaller buffer size, likely causing overflows with severe security consequences. We suspect this is caused by variable naming confusion.

```
if has_aux:
    example, *aux = example
else:
    aux = tuple()
rand_state = random.Random( ~~aux[-1]~~ aux[-1] if aux else 0 )
```

Case 3. aux could be an empty tuple in the above Python code snippet, where accessing its last element with aux[-1] will cause an index out of range exception, leading to DoS.

Table 8: Line-level metrics comparison with FLAG on the testset.

	Precision	Recall	TP_L	FP_L	TN_L	FN_L	Cost
FLAG	0.9%	32.0%	32	3732	5396	68	37.4$
LineBreaker	**14.3%**	**69.0%**	69	413	8715	31	5.1$

D. Comparison with Previous Work

TIB covers semantic bugs like variable misuse, which are also targeted by some existing works. However, direct comparisons with them are often difficult due to reasons like artifact unavailability [64], high training costs [9], and different targeted languages [8], [21], [53]. Therefore, we compare with some representative works with thorough design-level analysis, with auxiliary experiments when applicable.

Pre-LLM Approaches. Before LLMs' flourish, machine learning based approaches relied on limited and dedicated datasets for model training, affecting flexibility and generality. DeepBugs [53] targets only three specific types of name-based bugs with known patterns, limiting its generality. FICS [6] recognizes outlier functions as potential bugs, with AST-based embeddings. However, semantic information (*e.g.,* variable names) is not included in such embeddings, rendering it ineffective in TIB detection. Furthermore, FICs assumes the existence of a "correct" function in the codebase to spot the similar but buggy one, which may not hold especially in smaller codebases. Unlike these systems, LineBreaker requires no training, fine-tuning, or strong assumptions on bug patterns, codebase, or programming languages.

LLM-Based Approaches. FLAG [5] utilizes LLM for semantic bug detection at the line level and is the closest related work. It compares the original code against the code generated by LLM line by line, given other lines as the prompt in each line's iteration. To ensure a fair comparison, considering FLAG's high economic costs and line-by-line detection, we evaluated FLAG on 200 evenly sampled functions from D and D' and present the line-level metrics in Table 8. Among these, 100 samples/lines contain TIB (*i.e.,* 100 positive line-level cases), while the remaining lines constitute negative instances. The results show LineBreaker significantly outperforms FLAG in all essential metrics: precision (0.9% $\rightarrow$ 14.3%), recall (32% $\rightarrow$ 69%), and monetary cost on LLM APIs.

As seen, FLAG has many false alarms and misses the majority of TIBs, yet with high API cost (~$0.19 per function). We avoid timing the executions because both designs permit extensive concurrency optimization as computation power and API rate limit allow. LineBreaker's superior performance roots in its effective pipeline design. It combines efficient yet capable local LLMs with the SOTA LLM, boosting the performance while maintaining a lower cost. Additionally, FLAG's cost increases quadratically with the number of lines. Although `LineBreaker` induces expense on GPU hours because of running models locally, its API cost is then linear in the number of lines when using the SOTA LLM.

V. Related Work

Semantic Bug Detection. Semantic bug detection is an active research area. Besides comparative work introduced in §IV-D, various methodologies address bugs exhibiting well-defined and ambiguous patterns. Li et al. pioneered the application of LLMs to target Use Before Initialization bugs, demonstrating LLMs' potential to enhance conventional static analysis techniques [38]. Further advancements by researchers like Sun et al. and Wei et al. have expanded LLM applications to address a wider range of logic bugs, including API misuse [60], [61], [66], [72], [41], [74]. Traditional methods such as static analysis and formal verification also play significant roles in identifying and resolving semantic bugs with fixed patterns [42], [27], [23], [39], [47]. Recently, Natural Language Processing (NLP) has been leveraged to analyze documentation and pinpoint potential semantic inconsistencies or discrepancies between implementation and documentation [14], [49], [15], [13], [12]. However, these NLP-based approaches heavily rely on the quality of the documentation, facing substantial challenges with ambiguous or outdated materials prevalent in open-source environments.

LLM for Software Engineering. Recent advancements have made LLMs for coding tasks widely accessible through public API or open-source sharing of code and models. LLMs have demonstrated significant contributions to areas like fuzzing [37], [46], [70], code repair [71], [31], [22], [52], [30], test generation [73], [16], code-comment consistency checking [75], etc. Despite these developments, applying LLMs in scalable bug detection, specifically TIB, remains underexplored and largely experimental.

VI. Discussion and Conclusion

Limitations. Decoder-only models like Code Llama are currently limited to infilling at a single position each time. Although `LineBreaker` independently verifies the consistency of each code token, it may not detect sophisticated bugs associated with multiple discrete positions. Additionally, LLMs sometimes fall short when dealing with corner cases, especially when the correct code appears counterintuitive. This snippet demonstrates an FP case involving the transmission of a firmware header.

```
if (!securedSend(0x00, 128, (const uint8_t
↪  *)fwData->getBytesNoCopy())) ...
if (!securedSend(0x03, 256, (const uint8_t
↪  *)fwData->getBytesNoCopy() + 128)) ...
if (!securedSend(0x02, 256, (const uint8_t
↪  *)fwData->getBytesNoCopy() + (388))) ...
```

In the third `if` statement, GPT-4 believes the correct offset is 384 (128+256) instead of 388. It assumes that since 384 bytes of data have already been sent, 384 should be the offset for the next chunk. Relevant documentation is missing from the code repository. Since software development heavily relies on external libraries and standard specifications, LLMs can make mistakes when related information is missing from the context (*e.g.,* proprietary code). This drawback might be mitigated by compensating for richer context [7]. For example, RAG [36], [43] can intelligently fetch external knowledge.

GPU Memory Constraint. Our measurement experiments on local large code models (§II-D) revealed that inference tasks may fail with extended context. The direct solution is running `LineBreaker` on a system with connected GPUs that offer a larger GPU memory pool. However, some optimizations can reduce the memory consumption. For example, FlashAttention [20] reduces the space complexity linear in the context length, but requires hardware support. Quantization [40] compresses the model weights by reducing parameters' numerical precision, also leading to less memory usage.

Future Work. `LineBreaker` heavily relies on existing infilling models to mark suspicious TIBs. It may be beneficial to fine-tune or train a large model with improved specificity and recall. The fine-tuned model demonstrates better capability for auditing smart contracts [44]. Although simple, replaced tokens detection [17] fits our scenario with a new model trained. Researchers currently consider this task as a form of pretraining, which enhances the model's capability to comprehend semantics more effectively.

Another potential engineering direction is in the context, which serves as a foundation for inference. Our measurement study explored contexts at coarse-grained levels, such as files and functions (§II-D). Although out of `LineBreaker`'s scope, its design excludes bugs spanning multiple positions in the program. Fine-grained context engineering may leverage code slicing [50], [26], where relevant functions from different files can be curated, and irrelevant secure code lines can be removed. Thus, more complex bug patterns involving tokens scattered at different places might be captured.

VII. Conclusion

We systematically measured LLMs' capabilities to detect TIBs and identified their strengths and weaknesses. Based on the measurement, we design `LineBreaker`, a TIB detector using LLMs. It discovered 123 unknown bugs, demonstrating its effectiveness, scalability, and generality.

VIII. Acknowledgement

We sincerely thank Yuhui Hong, the anonymous reviewers, and the shepherd for their valuable insights. This work was supported in part by the National Science Foundation under Grant No. 2154199, and Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] GitHub - guidance-ai/guidance: A guidance language for controlling large language models. — github.com. https://github.com/guidance-ai/guidance. [Accessed 08-01-2025].

[2] langchain-ai/chat-langchain. https://github.com/langchain-ai/chat-langchain, 2024. Accessed: 2024-04-29.

[3] tree-sitter/tree-sitter: An incremental parsing system for programming tools. https://github.com/tree-sitter/tree-sitter, 2024. Accessed: 2024-04-29.

[4] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[5] Baleegh Ahmad, Benjamin Tan, Ramesh Karri, and Hammond Pearce. Flag: Finding line anomalies (in code) with generative ai. *arXiv preprint arXiv:2306.12643*, 2023.

[6] Mansour Ahmadi, Reza Mirzazade Farkhani, Ryan Williams, and Long Lu. Finding bugs using your own code: detecting functionally-similar yet inconsistent code. In *30th USENIX security symposium (USENIX Security 21)*, pages 2025–2040, 2021.

[7] Toufique Ahmed, Christian Bird, Premkumar Devanbu, and Saikat Chakraborty. Studying llm performance on closed-and open-source data. *arXiv preprint arXiv:2402.15100*, 2024.

[8] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[9] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876, 2021.

[10] Amit Seal Ami, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. " false negative-that one is going to kill you."-understanding industry perspectives of static analysis based security testing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 19–19. IEEE Computer Society, 2023.

[11] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[12] Yi Chen, Di Tang, Yepeng Yao, Mingming Zha, XiaoFeng Wang, Xiaozhong Liu, Haixu Tang, and Baoxu Liu. Sherlock on specs: Building {LTE} conformance tests through automated reasoning. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3529–3545, 2023.

[13] Yi Chen, Di Tang, Yepeng Yao, Mingming Zha, XiaoFeng Wang, Xiaozhong Liu, Haixu Tang, and Dongfang Zhao. Seeing the forest for the trees: Understanding security hazards in the {3GPP} ecosystem through intelligent analysis on change requests. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 17–34, 2022.

[14] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. Devils in the guidance: predicting logic vulnerabilities in payment syndication services through automated documentation analysis. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 747–764, 2019.

[15] Yi Chen, Yepeng Yao, XiaoFeng Wang, Dandan Xu, Chang Yue, Xiaozhong Liu, Kai Chen, Haixu Tang, and Baoxu Liu. Bookworm game: Automatic discovery of lte vulnerabilities through documentation analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1197–1214. IEEE, 2021.

[16] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576, 2024.

[17] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. In *International Conference on Learning Representations*.

[18] CloCkWeRX. Pull request for bug fix in rabbitvcs. https://github.com/rabbitvcs/rabbitvcs/pull/385/files#diff-eca3c60057143346eea4a850ba4fb60752c8cb397ec380293af7e252c1677d0f, 2024.

[19] Roi Cohen, May Hamri, Mor Geva, and Amir Globerson. Lm vs lm: Detecting factual errors via cross examination. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 12621–12640, 2023.

[20] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[21] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International conference on learning representations (ICLR)*, 2020.

[22] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481, 2022.

[23] Ansgar Fehnker and Ralf Huuck. Model checking driven static analysis for the real world: designing and tuning large scale bug detection. *Innovations in systems and software engineering*, 9(1):45–56, 2013.

[24] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[25] Matthew Finifter, Devdatta Akhawe, and David Wagner. An empirical study of vulnerability rewards programs. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 273–288, 2013.

[26] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, page 263–272, New York, NY, USA, 2005. Association for Computing Machinery.

[27] Jian Huang, Michael Allen-Bond, and Xuechen Zhang. Pallas: Semantic-aware checking for finding deep bugs in fast path. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 709–722, 2017.

[28] Sungjae Hwang and Sukyoung Ryu. Gap between theory and practice: An empirical study of security patches in solidity. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 542–553, 2020.

[29] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[30] Ma Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[31] Harshit Joshi, José Pablo Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. Repair is nearly generation: Multilingual program repair with llms. In *AAAI Conference on Artificial Intelligence*, 2022.

[32] Shubhra Kanti Karmaker Santu and Dongji Feng. TELeR: A general taxonomy of LLM prompts for benchmarking complex tasks. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 14197–14203. Association for Computational Linguistics, December 2023.

[33] Feyza Duman Keles, Pruthuvi Mahesakya Wijewardena, and Chinmay Hegde. On the computational complexity of self-attention. In *International conference on algorithmic learning theory*, pages 597–619. PMLR, 2023.

[34] Fatemeh Khoshnoud, Ali Rezaei Nasab, Zahra Toudeji, and Ashkan Sami. Which bugs are missed in code reviews: An empirical study on smartshark dataset. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 137–141, 2022.

[35] Lambda Inc. Gpu cloud - vms for deep learning | lambda, 2024. Accessed: 2024-09-04.

[36] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[37] Guochang Li, Chen Zhi, Jialiang Chen, Junxiao Han, and Shuiguang Deng. Exploring parameter-efficient fine-tuning of large language model on automated program repair. In *Proceedings of the 39th IEEE/ACM*

International Conference on Automated Software Engineering, pages 719–731, 2024.

[38] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*, 2024.

[39] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Mirchecker: detecting bugs in rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 2183–2196, 2021.

[40] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of machine learning and systems*, 6:87–100, 2024.

[41] Jinghua Liu, Yi Yang, Kai Chen, and Miaoqian Lin. Generating api parameter security rules with llm for api misuse detection. *Proceedings 2025 Network and Distributed System Security Symposium*, 2025.

[42] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12. IEEE, 2019.

[43] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. ReACC: A retrieval-augmented code completion framework. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240, Dublin, Ireland, May 2022. Association for Computational Linguistics.

[44] Wei Ma, Daoyuan Wu, Yuqiang Sun, Tianwen Wang, Shangqing Liu, Jian Zhang, Yue Xue, and Yang Liu. Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 330–342. IEEE Computer Society, 2024.

[45] Steve McConnell. *Code complete*. Pearson Education, 2004.

[46] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[47] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[48] OpenAI. Prompt engineering - OpenAI API. https://platform.openai.com/docs/guides/prompt-engineering, 2024.

[49] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. Deep just-in-time inconsistency detection between comments and source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 427–435, 2021.

[50] Aurora Papotti, Fabio Massacci, and Katja Tuma. On the effects of program slicing for vulnerability detection during code inspection: Extended abstract. ICSE-Companion '24, page 368–369, New York, NY, USA, 2024. Association for Computing Machinery.

[51] Jibesh Patra and Michael Pradel. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 906–918, 2021.

[52] Hammond A. Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356, 2021.

[53] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

[54] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 155–165, 2014.

[55] Cedric Richter and Heike Wehrheim. Learning realistic mutations: Bug creation for neural bug detectors. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 162–173. IEEE, 2022.

[56] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*, pages 31210–31227. PMLR, 2023.

[57] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mpnet: Masked and permuted pre-training for language understanding. *Advances in neural information processing systems*, 33:16857–16867, 2020.

[58] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T Barr, and Wei Le. A comprehensive study of the capabilities of large language models for vulnerability detection. *arXiv preprint arXiv:2403.17218*, 2024.

[59] StevenBlack. Stevenblack/hosts: Consolidating and extending hosts files from several well-curated sources. optionally pick extensions for porn, social media, and other categories. https://github.com/StevenBlack/hosts, 2024. Accessed: 2024-04-29.

[60] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *CoRR*, 2024.

[61] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[62] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical software engineering*, 19:1665–1705, 2014.

[63] Ruixiang Tang, Yu-Neng Chuang, and Xia Hu. The science of detecting llm-generated text. *Communications of the ACM*, 67(4):50–59, 2024.

[64] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh singh. Neural program repair by jointly learning to localize and repair. In *International Conference on Learning Representations*, 2019.

[65] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[66] Moshi Wei, Nima Shiri Harzevili, YueKai Huang, Jinqiu Yang, Junjie Wang, and Song Wang. Demystifying and detecting misuses of deep learning apis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[67] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *ArXiv*, abs/2302.11382, 2023.

[68] Brandon T. Willard and Rémi Louf. Efficient guided generation for large language models, 2023.

[69] wolfSSL Inc. wolfssl, 2024. Accessed: 2024-04-29.

[70] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[71] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE, 2023.

[72] Yifan Xia, Zichen Xie, Peiyu Liu, Kangjie Lu, Yan Liu, Wenhai Wang, and Shouling Ji. Exploring automatic cryptographic api misuse detection in the era of llms. *arXiv preprint arXiv:2407.16576*, 2024.

[73] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1607–1619, 2024.

[74] Yi Yang, Jinghua Liu, Kai Chen, and Miaoqian Lin. The midas touch: Triggering the capability of llms for rm-api misuse detection. *Proceedings 2025 Network and Distributed System Security Symposium*, 2025.

[75] Yichi Zhang, Zixi Liu, Yang Feng, and Baowen Xu. Leveraging large language model to assist detecting rust code comment inconsistency. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 356–366, 2024.