Exploring and Improving Real-World Vulnerability Data Generation via Prompting Large Language Models

Guangbei Yi*
Washington State University
guangbei.yi@wsu.edu

Yu Nong
University at Buffalo, SUNY
yunong@buffalo.edu

Minzhang Li
Washington State University
minzhang.li@wsu.edu

Haipeng Cai†
University at Buffalo, SUNY
haipengc@buffalo.edu

Abstract

Data-driven approaches have proven to be promising for vulnerability analysis, contingent on quality and sizable training data being available. Several dedicated vulnerability data generation techniques have demonstrated strong merits, yet they are limited to simple (single-line injection induced) vulnerabilities only and suffer from overfitting to seed samples. Large language models (LLMs) may overcome these fundamental limitations as they are known to be effective at generative tasks. However, it remains unclear how they would perform on the task of vulnerable sample generation. In this paper, we explore the potential and gaps of seven state-of-the-art (SOTA) LLMs for that task via prompting.

We reveal that the LLMs are capable of injecting vulnerabilities, with advanced prompting strategies such as few-shot in-context learning and our new vulnerability-introducing code-change semantics (VICS) guided prompting boosting the effectiveness, achieving up to 93% success rate on a synthetic dataset and 88% on real-world code. The LLMs can effectively perform both single- and multi-line injections, addressing a key limitation of prior work. Notably, they exhibit a strong preference for replacement edits, different from ground-truth patterns, and their effectiveness varies across CWE types. Furthermore, LLMs, particularly with VICS, outperform existing SOTA vulnerability generators with success rate improvements of up to 210%-343%. Crucially, the LLM-generated data substantially improves the performance of downstream DL-based vulnerability analysis models, especially with multi-line injections, boosting their accuracy by up to 70.1%. The generated samples also enhance the effectiveness of other LLMs for vulnerability analysis via RAG, with multi-line-injected samples yielding up to 16.5% gains. Most importantly, our findings reveal that, augmenting existing DL models with high-quality, LLM-generated data can lead to vulnerability analysis performance (up to 67.50% accuracy) superior to that of even the most advanced LLMs performing the same analysis (up to 27.40%), indicating the usefulness of the LLM-generated vulnerability data at present and in the longer term.

*The author participated in this work primarily as an REU student.
†The corresponding author.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/3744916.3773176

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

vulnerability analysis, data augmentation, deep learning, large language models, prompting, code generation

ACM Reference Format:
Guangbei Yi, Yu Nong, Minzhang Li, and Haipeng Cai. 2026. Exploring and Improving Real-World Vulnerability Data Generation via Prompting Large Language Models. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773176

1 Introduction

The dependability of the cyber world is widely afflicted by security vulnerabilities in the software which underlies its critical operations [14, 15, 30, 57]. Thus, defending against software vulnerabilities, which continue to grow in number and impact [32], has become increasingly critical and urgent [19, 56]. Among various response solutions, data-driven approaches have been gaining rising momentum for a variety of vulnerability analysis tasks [11, 13, 24, 25, 29, 36, 38, 40, 60, 65] due to their promise to overcome key (e.g., imprecision and/or low-coverage) limitations of traditional, code-analysis-based techniques [7, 35, 44].

By its nature, like any data-driven solutions, data-driven (e.g., deep-learning (DL) or large-language-model (LLM) based) vulnerability analysis relies on quality and sizable vulnerability data to train its model in order to be effective [46, 47]. Intuitively, the model would not be able to deal with a vulnerability unless similar ones have been seen in its training data, including vulnerable and non-vulnerable (normal) code samples. Normal samples are widely and richly available, while high-quality, labeled vulnerable samples are relatively scarce—and they are expensive to curate. Yet the model's knowledge about vulnerabilities is learned mainly from vulnerable samples. In fact, the dearth of such samples has been recognized and the problem has been studied in the last few years [11, 16, 18, 48], corroborating the critical need for, yet the present lack of, more vulnerability data (of good quality).

Yet manually curating large, quality vulnerability datasets is clearly undesirable and not scalable. Thus, researchers have explored automated techniques for generating vulnerable samples through vulnerability injection (i.e., injecting vulnerabilities from a given, relatively small set of seed samples to the widely available normal programs to generate vulnerable samples at scale [45–47].

Indeed, such automated generated vulnerable samples have proven to be quite conducive to boosting the performance of data-driven (especially DL-based) vulnerability analysis [46, 47] by augmenting the training of respective DL models. Yet despite their impressive results, state-of-the-art (SOTA) techniques dedicated to vulnerability sample generation suffer from two fundamental limitations.

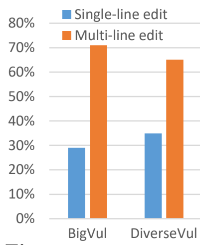

Figure 1: Real-world vuln. complexity.

First, they are designed to only inject vulnerabilities through single-line edits [45–47]: i.e., the vulnerabilities need just changing one line of code to be injected. As a result, the generated samples represent relatively simple scenarios, which are expected to mainly help downstream DL (e.g., vulnerability detection) models deal with (e.g., detect) simple vulnerabilities too, but less likely to help them deal with more complex vulnerabilities (that would need multi-line edits to inject). On the other hand, our motivating study shows that the vast majority of real-world vulnerabilities are in the more-complex category. As depicted in Figure 1, in both BigVul [22] and DiverseVul [12], two of the currently largest and most diverse real-world vulnerability datasets, over 65% of the vulnerabilities are out of scope (i.e., requiring multi-line injections) for the existing most powerful vulnerable sample generators [45, 47]. Thus, *the bigger part of the data problem remains unaddressed*—we are still in critical need for complex (multi-line) vulnerability data to augment data-driven models so that they may detect such complex vulnerabilities, which are more urgent to detect.

Second, these SOTA solutions extract vulnerability-injection patterns from a given set of (*seed*) vulnerable samples and their paired fixed versions. Thus, their effectiveness is limited and likely overfit to those seed samples, which are generally small in size. The latest, dedicated vulnerability data generation technique, VGX [45], mitigates this limitation by manually deriving and diversifying those patterns, which compromises its scalability. Apparently, *this second limitation is hard to overcome, so is the first one, due to the nature of the core design.* A new direction ought to be sought.

In this paper, we explore such a new direction in automated vulnerability data generation—by leveraging LLMs. Recently, LLMs appear to be promisingly capable of (code) generation. Thus, it is valuable to exploit the generative power of LLMs to crack the critical data challenge in data-driven vulnerability analysis.

We examine the potential and gaps of seven LLMs of different kinds (general vs. code-oriented, and generative vs. reasoning), using both a synthetic and a real-world dataset so that we can comparatively assess the challenge of generating synthetic versus realistic vulnerabilities with LLMs. For each LLM, we adopt three prompting strategies: standard, few-shot in-context learning (ICL), and our specifically designed vulnerability-introducing code-change semantics (VICS) guided prompting which is based on the idea of chain-of-thought (CoT). Across these LLMs, the success rates on the synthetic and real-world dataset range from 34–91% and 20–72% with standard prompting. Few-shot ICL helps, raising the success rates to 49–93% and 62–84%, while VICS further boosts them to 56–94% and 68–88%, on the synthetic and real-world datasets, respectively, demonstrating their consistent effectiveness in improving the quality of generated samples.

Beyond assessing the overall effectiveness of LLMs in injecting vulnerabilities, our study delves deeper into the characteristics of these injections, examining the types of code edits (additions, deletions, replacements), their complexity (single-line vs. multi-line), and their performance across different Common Weakness Enumeration (CWE) categories. We also conduct a comparative analysis against existing state-of-the-art vulnerability generation techniques to benchmark LLMs' capabilities. To examine whether the generated vulnerability samples are practically useful and effective for boosting data-driven vulnerability analysis performance, we use the LLM-generated data to augment the training datasets of SOTA DL-based vulnerability analysis models. Besides, we also explore whether these generated samples can enhance the effectiveness of other LLMs when used for the same vulnerability analysis task.

Among other findings, our study reveals that:

- **LLMs are highly effective at injecting vulnerabilities.** With VICS, the seven studied LLMs achieve vulnerability injection success rates of up to 94% on synthetic datasets and 88% on real-world datasets.
- **LLMs inject vulnerabilities with diverse characteristics.** LLMs demonstrate the ability to generate both simple and complex multi-line vulnerabilities, which better reflect real-world flaws and are more typical of real-world vulnerable code, aligning with ground-truth patterns. Success rates varied across CWEs, indicating varying effectiveness on different vulnerability types. Furthermore, LLMs exhibit a strong preference to replacement edits, different from ground-truth vulnerability-introducing code-change patterns which tend to delete code more often.
- **LLMs remarkably outperform existing vulnerability-injection techniques.** Compared to SOTA, dedicated vulnerability generators VulGen [47] and VGX [45] as baselines, LLMs with VICS demonstrate superior performance (210%-343% higher success rates). This merit is particularly due to LLMs' capability of multi-line injections and their diverse modification patterns.
- **LLM-generated vulnerable samples substantially improve downstream DL-based vulnerability detection.** When added to the training sets of DL-based (line-level) detectors FVD-DPM and LineVul, LLM-generated samples lead to huge performance gains. For instance, GPT-4o's multi-line-injection resulting samples boost FVD-DPM's Top-1 accuracy by 36.21% and LineVul's Top-1 accuracy by 70.08%. This highlights the critical role of high-quality, LLM-generated data in enhancing the capabilities of specialized DL models for vulnerability analysis.
- **LLM-generated data improves the effectiveness of other LLMs for vulnerability analysis through Retrieval Augmented Generation (RAG).** When LLMs are augmented with LLM-generated data via RAG, their vulnerability analysis performance improves too. For example, Llama-3's Top-10 accuracy increased from 14.8% to 31.3%, and Claude-3.5's from 22.9% to 28.5% with the LLM-generated samples. This indicates that LLMs themselves benefit from access to high-quality vulnerability data generated by other LLMs.
- **LLM-generated data enables existing models to outperform direct LLM vulnerability analysis.** Importantly, our findings reveal that augmenting existing DL models with high-quality, LLM-generated data can yield vulnerability analysis performance

(up to 67.50% Top-10 accuracy) superior to those of even the most advanced LLMs performing the same analysis directly (up to 27.40% Top-10 accuracy), indicating the usefulness of the generated vulnerability data even in the era of LLMs.

Based on our findings, we propose several actionable recommendations to advance LLM-based vulnerability data generation. In particular, we suggest to:

(1) Enhance CoT reasoning and ICL techniques so that LLMs can more accurately capture the transformation patterns between secure and vulnerable code. This may involve fine-tuning on vulnerability-specific datasets to improve both code generation and instruction following.
(2) Refine prompt design and exemplar selection strategies to minimize false positives and improve the stability of generated vulnerabilities. This will ensure that the generation/injection is not only syntactically correct but also effectively simulates real-world vulnerability patterns without altering functionalities—our results show *even with continuous advancement of LLMs, carefully designed, task-specific prompting strategies matter.*
(3) Expand the context window during inference to allow LLMs to process and generate longer, more complex code samples, thereby better handling multi-line vulnerabilities given their much greater prevalence in real-world vulnerable programs.
(4) Leverage LLM-generated samples for data augmentation in training DL–based vulnerability detectors. Augmenting these models with high-quality synthetic data has been shown to boost detection performance significantly.

Through this study, we initiate a new direction for addressing the critical data needs in data-driven vulnerability analysis through *transferring/distilling vulnerability knowledge from LLMs to smaller DL models or other LLMs.* Our results demonstrated the merits of, as well as suggesting future work needed in, this new direction.

2 Methodology

We elaborate on the design of our study, including the research questions (§2.1) as well as the datasets (§2.2, metrics (§2.3), LLMs (§2.4), and prompting strategies (§2.5) used for answering the RQs.

2.1 Research Questions

Our study is governed by five main research questions (RQs) as outlined and justified as follows.

- **RQ1. How effective are LLMs in vulnerability data generation?** To start with, we gauge *the overall potential of LLMs for addressing the data needs in data-driven vulnerability analysis*, by injecting vulnerabilities into non-vulnerable code samples.
- **RQ2. What are the characteristics of LLMs' vulnerability injection edits?** By examining the types (add/delete/replace) and complexities (single- vs. multi-line change) of these code edits, we aim to uncover *how LLMs generate vulnerable samples*, dissecting their behavioral patterns during vulnerability injection, both across and for specific vulnerability types (CWEs).
- **RQ3. Are LLMs better than existing techniques for vulnerability generation?** By comparing the LLMs to SOTA dedicated data-generation techniques in terms of injection effectiveness, we aim to assess *whether LLMs represent a more promising direction* in real-world vulnerability data generation.
- **RQ4. Can the LLM-generated data improve DL-based vulnerability analysis models?** By measuring if and how much the generated data boost the performance of SOTA DL-based models via training augmentation, we aim to evaluate the *practical usefulness of LLM-based vulnerability data generation*, comparatively to SOTA dedicated data-generation techniques.
- **RQ5. Do the LLM-generated data improve LLM-based vulnerability analysis?** By examining whether LLMs' vulnerability analysis effectiveness can also be enhanced by the data generated by other LLMs, we aim to assess the *longer-term merits of LLM-based vulnerability data generation* even in the era of LLMs being increasingly used for vulnerability analysis.

2.2 Datasets

To enable comprehensive and comparative exploration, we use two complementary datasets: one synthetic and one real-world.

For the synthetic dataset, we consider **SARD** [9], a large set of vulnerable code samples and corresponding fixed versions. We only use samples written in C, (1) for a focused study, without dealing with language related confounding factors and (2) because C is the most vulnerable language [17, 51]. Further considering the cost of LLMs and our study scale, we randomly sampled 500 pairs of samples, covering the five most dominating types of vulnerabilities (CWEs-121, 190, 191, 134, 124, and others—accounting for 27.95%, 23.74%, 18.88%, 12.65%, 11.91%, and 4.87% of the 500, respectively) in entire SARD. These samples have on average 31.59 lines of code and 4.91 changed lines between each pair.

For the real-world dataset, we also randomly sampled 500 pairs, from CVEFixes [8] and PrimeVul [21] combined, also covering its five most dominating CWEs (125, 190, 416, 476, and 787, each accounting for 20% of the 500). These samples have on average 86.33 lines of code and 3.30 changed lines between each sample pair. We refer to this combined real-world dataset as **CVE**.

These two datasets are mainly used for RQ1, RQ2, and RQ3, where the above scale limit and CWE scope restrictions are set to enable in-depth analysis. For RQ4 and RQ5, we focus on real-world samples, where the datasets used are further described in §6.

2.3 Metrics

To evaluate LLMs' performance on vulnerability data generation, we consider two effectiveness metrics, defined and justified below:

- **Vulnerable:** The generated code sample is indeed vulnerable. Since we aim at injecting vulnerabilities, the basic criterion of successful injection is that the resulting sample is vulnerable.
- **Stable:** The generated code sample preserves the functionality of the input non-vulnerable version. Without this constraint, vulnerability injection could be trivial and make the vulnerable sample disconnect from the non-vulnerable version—the resulting data would be much less useful for vulnerability analysis.

2.4 Large Language Models

We selected LLMs in our study based on the following criteria:

(1) **Affordability**: The models should be practical for ordinary users to access and run, in terms of computing power and monetary budget. For open-source LLMs to run on local hardware, we selected those fitting our machines (with AMD Threadripper Pro 5595WX 4.5GHz processor, 4 RTX A6000 GPUs, and 512GB RAM). For commercial LLMs accessed via paid APIs, we set a cost limit,

only choosing models charging ≤$3 per million tokens, to ensure the entire study's total cost is within what we can afford.

(2) **Instruction-following**: The models must be able to follow non-trivial instructions (i.e., those of the complexity in our study, especially with ICL and VICS) as is required for generating vulnerable code samples per our requirements which are not trivial.

(3) **Effectiveness**: The models must be effective in generating syntax-correct code based on the instructions. Therefore, we prefer the high-ranking models in the latest LLM leaderboard [2].

(4) **Coverage**: To enable comprehensive analysis, we also aim to include both general-purpose and code-oriented LLMs, and both generative and reasoning LLMs.

With these criteria, we chose GPT-4o [58], Claude-3.5-Sonnet [5], Llama3-70B [4], Qwen2.5-32B-Instruct [63], CodeLlama-34B-Instruct [3], DeepSeek-Coder-33B-Instruct [28], and DeepSeek-R1-70B [27].

To reduce randomness and ensure consistent, stable, and reproducible behavior/outputs across repeated trials of these LLMs during code generation, we fixed their decoding temperature at 0.1.

2.5 Prompting Strategies

How LLMs are prompted is known to have major impact on their performance. Thus, we consider two popular existing strategies, and a new strategy we proposed specifically for vulnerability injection.

2.5.1 Standard Prompting. With this strategy, we directly instruct an LLM to inject a specific type of vulnerability to given code.

Prompt Construction. The prompt is carefully designed, asking for code change and explanation to facilitate manual validation.

> Q: As an example to illustrate {CWE_ID} vulnerabilities for educational purposes, inject a vulnerability of this type into the following non-vulnerable code. Indicate your code change with explanation. {input_code}

2.5.2 Few-Shot ICL. This strategy uses a few carefully chosen exemplars to guide LLMs in vulnerability injection, in three steps:

(1) Exemplar Data Curation. For each targeted CWE, we selected 100 sample pairs relevant to the CWE from the BigVul dataset [22]. Each pair is manually verified to ensure that the code changes inject vulnerabilities, used as an exemplar which includes:

- **V1:** The original, normal (non-vulnerable) version of the code.
- **V2:** The corresponding vulnerable version of the code.

(2) Exemplar Selection. Typically, exemplars are selected manually in few-shot ICL. To automate our study pipeline, we propose an *adaptive exemplar selection* approach. Given a normal code sample and a CWE id, candidate exemplars of that CWE are retrieved. Then, to maximize relevance, exemplars are selected as follows:

(1) The normal code given and each exemplar's V1 are compared by the LLM, asking if the vulnerability in the exemplar can be introduced similarly to the given code, along with a code-structure similarity score in [1,10].
(2) The model's response is parsed to extract a binary decision ("yes" or "no") along with the score.
(3) Exemplars receiving a "yes" response are considered, and up to three top-scored (i.e., most similar) candidates are chosen—there may be less than three with a "yes" response.

(3) Vulnerability Injection. With the selected exemplars, the LLM is prompted to inject vulnerabilities to the input normal code. First, to help it better follow the exemplars, we instruct the model to summarize the patterns of code changes between the code pairs.

> Q: For educational illustration purposes, summarize the pattern of code changes that inject {CWE_ID} vulnerabilities to V1 to result in V2 in the code pairs below, without including reasoning. IMPORTANT: The summary must describe the vulnerability injection mechanism.
> Example 1: V1: {exemplar V1 code} V2: {exemplar V2 code}
> Example 2: V1: {exemplar V1 code} V2: {exemplar V2 code}
> Example 3: V1: {exemplar V1 code} V2: {exemplar V2 code}

Then, based on the summary, we prompt the LLM to inject the vulnerability into the input code.

> Q: Based on the following summary: {summary}, for the V1 version of input code below, generate the corresponding changed version V2. Indicate your code change with explanation. V1: {input_code}

2.5.3 VICS. Neither of the previous two prompting strategies is specifically designed for the *task* of vulnerability injection. Thus, to better assess the potential of prompting LLMs, we propose *vulnerability-introducing code-Change semantics* guided prompting (**VICS**), a new, *task-specific* strategy based on chain-of-thought (CoT) reasoning. VICS is aimed at enhancing vulnerability injection by leveraging explicit reasoning about code transformations, capturing the semantic differences between normal and associated vulnerable versions of code. Unlike traditional few-shot prompting relying solely on exemplar demonstration, VICS explicitly generates and utilizes detailed explanations of how vulnerabilities are deliberately introduced, offering clear and focused guidance to LLMs for vulnerability injection. It works in three steps.

(1) Automatic Reasoning Generation. Based on CoT, VICS needs the reasoning steps in exemplars. It automatically generates these steps using the LLM itself, with the prompt template below:

> Q: You are an expert in software security and vulnerability injection. Given the following code pairs V1 and V2 where V2 introduces a {CWE_ID} vulnerability to V1, your job is to explain how the vulnerability was **deliberately introduced** when transforming V1 (normal) to V2 (vulnerable). **Key requirements:** (1) focus on how V2 introduces the vulnerability (what unsafe change was made and why it causes vulnerable behaviors); (2) focus on the technical mechanism (e.g., buffer overflow, out-of-bounds access, dangling pointer) of vulnerability introduction; (3) do not describe V1 as 'correct', 'secure', or 'fixed'; treat V1 as a neutral starting point; (4) do not describe V2 as a 'mistake', 'oversight', or 'error'; it is a **deliberate weakening**; (5) avoid using phrases like 'prevents', 'fixes', 'ensures', 'improves', 'mitigates', 'protects', etc. V1: {v1 code} V2: {v2 code}

The resulting "reasoning" steps explicitly detail the semantic changes and the process through which the normal/secure code (V1) is transformed into its vulnerable counterpart (V2). These steps, along with the code samples used to generate them, are then utilized as exemplars in Step (3) below.

(2) Exemplar Selection. We follow the same *Exemplar Selection* approach as in *Few-shot ICL*, selecting top three exemplars.

(3) Vulnerability Injection. With the selected exemplars, we prompt the LLMs to summarize code-change patterns in the exemplars and follow the patterns to transform the given V1 of input code to its V2, as in few-shot ICL. Differently, here each exemplar includes the reasoning steps, which guide the LLMs to both summarize the patterns and realize the transformation more effectively.

To illustrate, Figure 5 shows a complete VICS workflow for injecting a CWE-416 (Use-After-Free) vulnerability. After automatically selecting suitable exemplars and their auto-generated reasoning, the LLM first summarizes the transformation patterns, identifying the core flaw as unsafe resource management. Guided by this summary, the model successfully injects the vulnerability into the input

Table 1: Comparison between ICL and VICS prompting

Main Step	ICL (Few-Shot)	VICS (Semantic Reasoning)
1. Exemplar Curation	vulnerable/non-vulnerable code pairs per CWE	Same curated pool
2. Adaptive Selection	Input compared with each exemplar, scored (1–10) with yes/no judgment; top-3 selected.	Same, but each exemplar also includes auto-generated reasoning-steps.
3. Prompting	LLM sees exemplars and a brief summary of code-change patterns, then applies them to the input normal code.	LLM sees exemplars with reasoning, first summarizes vulnerability-introducing-change semantics, then applies them to the input normal code.
4. Example Output (LLM Response)	Figure 3: Unsafe indexing added→vulnerability injected, but functionality broken (unstable).	Figure 5: `remote_address_.reset()` commented out→use-after-free injected, functionality preserved (stable).

Table 2: Reliability of labeling LLM-generated samples

Model	Fleiss' Kappa	Randolph's Kappa
GPT-4o	0.678	0.838
Claude-3.5	0.895	0.956
CodeLlama	0.909	0.928
Qwen2.5	0.859	0.905
Llama3	0.885	0.939
DeepSeek-R1	0.891	0.946
DeepSeek-Coder	0.710	0.775

normal code by commenting out the remote_address_.reset() line, creating a use-after-free condition.

Table 1 further outlines the key steps of our ICL and VICS prompting strategies while comparing them, with example LLM responses.

3 RQ1: LLM Effectiveness in Sample Generation

We evaluate the effectiveness of LLMs in injecting vulnerabilities into 1,000 normal samples in SARD and CVE, by manually validating if each LLM-generated sample is vulnerable and stable. To ensure reliability of our manual process, three of the paper authors, who have 3-5 years of experience in software security, first engaged in discussions to establish a unified evaluation standard. Then, each rater independently performed manual code auditing and evaluation of every sample, followed by resolving any disagreements via negotiation until reaching a consensus on the sample.

To measure the reliability, we quantify inter-rater agreements using Fleiss' Kappa and Randolph's Kappa [39]. Fleiss' Kappa measures agreement among multiple raters beyond chance, with values ranging from 0 (no agreement) to 1 (perfect agreement), where higher values indicate stronger reliability. Randolph's Kappa, on the other hand, assumes a uniform distribution of ratings across categories, which is useful when class distributions are imbalanced. A Fleiss' Kappa value above 0.6 is generally considered substantial agreement among raters. As shown in Table 2, our manual validation was highly reliable for different LLMs.

In our study, the input code samples are non-vulnerable. This is ensured in two ways: (1) the datasets we use label normal samples as fixed versions of known vulnerable code, suggesting that they are non-vulnerable; (2) to verify that, we manually inspected the 500 input samples from each dataset, confirming they were non-vulnerable while validating the injected versions as noted above.

3.1 Overall Effectiveness

Table 3 summarizes the effectiveness of the LLMs with three prompting strategies on the two datasets, where the highest success rates on each dataset are highlighted: SARD and CVE. While the LLMs show considerable potential for vulnerability injection, their effectiveness varies notably depending on the model choice and the prompting strategy employed—an injection is a success if the resulting sample is both vulnerable (**Vuln**) and stable. The peak performance is strong. For instance, the top-performing model, GPT-4o, successfully generates stable and vulnerable samples for up to 93% of the 500 normal samples in the SARD dataset and 88% in the more complex, real-world CVE dataset. In contrast, other models (e.g., CodeLlaMa, Llama-3) exhibit much lower performance, overall forming a clear hierarchy of capability among the LLMs.

Table 3: Effectiveness of LLMs for vulnerability injection

LLM	Prompting	Dataset	%Stable		%Unstable	
			%Vuln	%Non-Vuln	%Vuln	%Non-Vuln
GPT-4o	Standard	SARD	91	7	2	0
		CVE	71	9	18	2
	Few-shot	SARD	93	3	4	0
		CVE	82	8	10	0
	VICS	SARD	93	7	0	0
		CVE	88	6	6	0
Claude-3.5	Standard	SARD	78	1	11	10
		CVE	71	1	28	0
	Few-shot	SARD	86	8	6	0
		CVE	78	7	14	1
	VICS	SARD	94	3	3	0
		CVE	83	7	9	1
Llama3	Standard	SARD	56	7	35	2
		CVE	54	10	36	0
	Few-shot	SARD	70	27	3	0
		CVE	68	12	20	0
	VICS	SARD	76	24	0	0
		CVE	74	12	14	0
CodeLlama	Standard	SARD	34	36	21	9
		CVE	20	48	24	8
	Few-shot	SARD	49	45	6	0
		CVE	64	24	10	2
	VICS	SARD	56	38	6	0
		CVE	68	22	8	2
Qwen2.5	Standard	SARD	71	8	21	0
		CVE	46	52	2	0
	Few-shot	SARD	83	12	5	0
		CVE	62	26	12	0
	VICS	SARD	88	7	5	0
		CVE	68	22	10	0
DeepSeek-R1	Standard	SARD	86	7	7	0
		CVE	72	4	23	1
	Few-shot	SARD	93	6	1	0
		CVE	84	4	12	0
	VICS	SARD	93	5	2	0
		CVE	88	4	8	0
DeepSeek-Coder	Standard	SARD	75	12	13	0
		CVE	62	11	27	0
	Few-shot	SARD	81	12	7	0
		CVE	74	12	14	0
	VICS	SARD	89	8	3	0
		CVE	78	11	11	0

Few-shot ICL brought consistent and notable improvement for all LLMs, boosting their generation of stable and vulnerable samples while markedly reducing their incidences of unstable/non-vulnerable code changes. The benefits are most pronounced for the mid- and lower-tier models, which see their success rates increase by 10-30%. These gains suggest that providing even a few high-quality examples (i.e., ICL exemplars) is crucial for guiding the LLMs toward correct injections.

The VICS prompting strategy we propose elevates performance even further by explicitly guiding the models to reason about the semantic changes required to introduce a vulnerability. With VICS, unstable outputs are almost entirely eliminated across the board, and the models become overwhelmingly consistent in producing the intended stable and vulnerable transformations. This semantics-focused approach allows weaker models to catch up substantially, while top-tier models consolidate their lead, indicating that advanced, task-specific prompting can improve the effectiveness and reliability of any LLM for the task (of vulnerability injection).

A consistent contrast is that the real-world dataset poses a greater challenge than the synthetic one, resulting in lower overall success rates for every model and prompting strategy on the CVE dataset. However, the proportional performance uplift gained by moving from standard prompting to few-shot ICL and then to VICS prompting remains just as strong. This indicates that while real-world code complexity makes the task inherently harder, the benefits of advanced prompt engineering are universal and crucial for maximizing the success of vulnerability injection.

Finding 1: Among the LLMs, GPT-4o achieves the best overall effectiveness. The real-world dataset is generally more challenging to inject vulnerabilities, as expected. Across LLMs and datasets, more advanced prompting consistently helps push up the effectiveness, with VICS leading to most successful sample generation.

Table 4: Categories and distribution of failed injections

LLM	Dataset	%Failed Cases	
		%Hallucination	%Semantic Complexity
GPT-4o	SARD	97	3
	CVE	54	46
Claude-3.5	SARD	61	39
	CVE	38	62
Llama3	SARD	93	7
	CVE	48	52
CodeLlama	SARD	84	16
	CVE	66	34
Qwen2.5	SARD	57	43
	CVE	71	29
DeepSeek-R1	SARD	69	31
	CVE	47	53
DeepSeek-Coder	SARD	74	26
	CVE	48	52

Q: As an example to illustrate CWE_125 vulnerabilities for educational purposes, inject a vulnerability of this type into the following non-vulnerable code. Indicate your code change with explanation.

```
...if (GetPixelReadMask(image,p) == 0){
    SetPixelBackgoundColor(splice_image,q);p+=GetPixelChannels(image);
    q+=GetPixelChannels(splice_image);continue;}...
```

Model Response:

```
...if (GetPixelReadMask(image,p) == 0){//CWE125 OutofBounds Read injected
    SetPixelBackgoundColor(splice_image,q);p+=GetPixelChannels(image);
    q+=GetPixelChannels(splice_image);continue;}...
```

Figure 2: A stable, non-vulnerable code sample generated by GPT-4o with standard prompting.

3.2 Case Analysis

To better understand what happens behind the performance numbers, we looked into each successful and failed injection. In particular, for the failed cases (represented in the last 3 columns of Table 9), we examined the reasons of the failure. We found that these failures are mainly due to two reasons: (1) *hallucination*—the LLMs claim injection as succeeded yet without making any code changes and (2) *semantic complexity*—the LLMs cannot reason about complex code-semantics as required for successful injections.

Table 4 shows the distribution of these two failure reason categories across the two datasets and studied LLMs, where the percentage of failed cases of dominating reasons for each LLM and dataset is highlighted. As shown, hallucination accounts for an overwhelming majority of failures on the simpler SARD dataset, while semantic-complexity is the more prevalent reason of failed injections in the more complex CVE dataset. This consistent trend across various LLMs highlights the distinct challenges posed by real-world code semantics compared to synthetic examples.

Next, we illustrate the strengths/limitations of different models under various prompting strategies with specific examples. Due to space limits, *only changed code regions are shown*; the rest of the code and model's output on code changes/explanations are omitted.

Figure 2 shows an example of GPT-4o's unsuccessful injection with standard prompting: the generated code is not vulnerable. Although the LLM claimed to have introduced a vulnerability by injecting "OutOfBounds Read", our inspection reveals that the model hallucinates—it did not make any change.

Figure 3 shows an example of Llama3's failed injection with few-shot ICL prompting—the generated code is vulnerable but not stable. The LLM introduced an out-of-bounds array access, creating a buffer overflow vulnerability but also disrupting the input code's original functionality (i.e., unstable injection).

Figure 4 shows an example of DeepSeek-Coder's failure with standard prompting. The generated sample is unstable and non-vulnerable. The model added a statement that disrupted the original functionality yet without introducing an exploitable flaw.

Figure 5 exemplifies successful sample generation by GPT-4o with VICS prompting while also illustrating the complete workflow

Q: Based on the following summary: "*The core behavior is to deliberately remove or weaken critical security validation logic before the program processes data. This is demonstrated by actions such as: deleting checks that verify if a data packet is sufficiently large before it is parsed; eliminating constraint functions that ensure an index is within a valid range before it's used to access an array; and removing initial checks that confirm an input value is within expected bounds before being passed to a complex algorithm. These removed safeguards, which were the final line of defense against processing malicious or malformed data, are what allow the program to attempt reading from memory regions it should not access, directly resulting in an out-of-bounds read vulnerability*", for the V1 version of input code below, generate the corresponding changed version V2. Indicate your code change with explanation. V1:

```
...Quantum cbcr[4]; ... // omitted code due to space
case 1:{quantum=(size_t) (ScaleShortToQuantum((unsigned short)
(((pixel >> 12) & 0x3ff) << 6))); break;}...
```

Model Response:

```
...Quantum cbcr[4]; ... // omitted code due to space
case 1:{quantum=(size_t) (ScaleShortToQuantum((unsigned short)
(((pixel >> 12) & 0x3ff) << 6))); cbcr[i]=cbcr[100]; break;}
// Introduced vulnerable indexing operation...
```

Figure 3: An unstable, vulnerable code sample generated by CodeLlama with few-shot ICL.

Q: As an example to illustrate CWE_787 vulnerabilities for educational purposes, inject a vulnerability of this type into the following non-vulnerable code. Indicate your code change with explanation.

```
...p = pixels;
q = GetAuthenticPixels(image,0,row,image->columns,1,exception);
if (q == (Quantum*) NULL) return MagickFalse;...
```

Model Response:

```
...image->columns += 1; p = pixels; // add an additional index to be read
      beyond the allocated array, simulating a CWE-125 vulnerability
q = GetAuthenticPixels(image,0,row,image->columns,1,exception);
if (q == (Quantum*) NULL) return MagickFalse;...
```

Figure 4: An unstable, non-vulnerable code sample generated by DeepSeek-Coder with standard prompting.

of our VICS approach. From the CoT exemplars that demonstrate the reasoning about the vulnerability injection mechanism, the LLM captures the patterns of vulnerability-introducing code changes. It then correctly follows the patterns to inject the vulnerability to the given input code, resulting in a stable and vulnerable sample.

Finding 2: Under standard prompting, LLMs sometimes hallucinate—claiming to inject vulnerabilities without actually doing so—resulting in unstable and/or non-vulnerable code. VICS prompting can help mitigate this issue.

4 RQ2: Characteristics of LLM Edits

We now look into the characteristics of the vulnerability injections made by the LLMs to understand the underlying model behaviors/preferences, including the distribution of code injection edit types (add, delete, and replace), the complexity of the edits (single-line vs. multi-line), and the effectiveness across different CWEs.

The results are shown in Table 5, with dominant types (>50%) highlighted and long terms abbreviated (PS: prompting strategy, Cmpl.: complexities, SL: single-line, ML: multi-line, GTh: ground-truth, add: add line(s) only, del: delete line(s) only). Each number cell shows the proportion of edits over all attempts with the success rate in the parenthesis. For example, with standard prompting, GPT-4o's single-line edits on the CVE dataset consist of 48% code additions with a success rate of 70%, 7% deletions with a success rate of 68%, and 45% replacements with a success rate of 72%.

The LLMs overwhelmingly prefer to replace existing lines when injecting vulnerabilities, and this remains true across models, prompting strategies, and datasets. The dominance is even stronger on the simpler SARD dataset and for multi-line injections. Addition-only attempts are less common and deletion-only ones are the least

Q: For educational illustration purposes, summarize the pattern of code changes that inject CWE-416 vulnerabilities to V1 to result in V2 in the code pairs below, without including reasoning. IMPORTANT: The summary must describe the vulnerability injection mechanism.

Example 1:

```
V1:        ......
           IndexedDBCursor::~IndexedDBCursor() {
               if (transaction_) {transaction_->UnregisterOpenCursor(this);}
               Close();
           }
           ......
V2:        ......
(removed):  if (transaction_) {transaction_->UnregisterOpenCursor(this);}
           .....
```

(Reasoning:) Removing the transaction call intentionally breaks a critical safety mechanism. This change prevents a cursor object from notifying its associated transaction upon deletion, leaving the transaction with a dangling pointer. This creates a use-after-free vulnerability, where the transaction might later attempt to access the freed memory of the deleted cursor.

Example 2:......

Example 3:......

Model Response: The pattern of code changes that inject CWE-416 vulnerabilities involves a targeted modification that disrupts an object's lifecycle management, causing it to be deallocated prematurely while a reference to it is still held. This is accomplished by altering a single, critical function call responsible for tracking the object's liveness. The specific injection mechanisms include removing an object's deregistration call from its destructor, omitting a necessary trace call to hide the object from the garbage collector, or replacing a safe, reference-counting function with an unsafe equivalent that breaks the object's lifecycle logic.

Q: Based on the following summary: "[the model response above]", for the V1 version of input code below, generate the corresponding changed version V2. Indicate your code change with explanation. V1:

```
...int UDPSocketLibevent::InternalConnect(const IPEndPoint& address) {
    ......
    remote_address_.reset(new IPEndPoint(address));
    ...
```

Model Response:

```
...int UDPSocketLibevent::InternalConnect(const IPEndPoint& address) {
    ......
    // remote_address_.reset(new IPEndPoint(address));
    ...
```

Figure 5: A stable, vulnerable (i.e., *success*) code sample generated by GPT-4o with VICS.

Table 5: Edit types and complexities of LLM injections

LLM	PS	Dataset	Cmpl.	%add	%del	%replace	LLM	PS	Dataset	Cmpl.	%add	%del	%replace
GPT-4o	Standard	SARD	SL	0 (0)	0 (0)	100 (91)	Qwen2.5	Standard	SARD	SL	2 (68)	9 (65)	89 (72)
			ML	1 (90)	0 (0)	99 (91)				ML	2 (68)	2 (65)	96 (72)
		CVE	SL	48 (70)	7 (68)	45 (72)			CVE	SL	20 (45)	0 (0)	80 (46)
			ML	56 (70)	5 (60)	39 (74)				ML	15 (45)	0 (0)	85 (46)
	ICL	SARD	SL	0 (0)	0 (0)	0 (0)		ICL	SARD	SL	0 (0)	16 (81)	84 (83)
			ML	72 (92)	0 (0)	28 (96)				ML	4 (78)	18 (80)	78 (85)
		CVE	SL	19 (79)	13 (75)	68 (84)			CVE	SL	17 (60)	49 (61)	34 (65)
			ML	43 (80)	0 (0)	57 (85)				ML	10 (60)	18 (59)	72 (64)
	VICS	SARD	SL	5 (88)	2 (85)	93 (94)		VICS	SARD	SL	3 (82)	8 (80)	89 (90)
			ML	40 (90)	0 (0)	60 (95)				ML	2 (82)	1 (80)	97 (89)
		CVE	SL	50 (86)	10 (80)	40 (91)			CVE	SL	19 (65)	0 (0)	81 (69)
			ML	48 (87)	0 (0)	52 (89)				ML	15 (65)	0 (0)	85 (69)
Claude-3.5	Standard	SARD	SL	17 (76)	0 (0)	83 (78)	DeepSeek-R1	Standard	SARD	SL	5 (82)	9 (80)	86 (87)
			ML	0 (0)	1 (70)	99 (78)				ML	7 (84)	1 (80)	92 (86)
		CVE	SL	15 (68)	19 (69)	66 (72)			CVE	SL	14 (70)	29 (70)	57 (74)
			ML	17 (70)	0 (0)	83 (71)				ML	17 (70)	2 (65)	81 (73)
	ICL	SARD	SL	5 (82)	10 (83)	85 (87)		ICL	SARD	SL	1 (88)	16 (91)	83 (94)
			ML	14 (84)	4 (80)	82 (87)				ML	3 (89)	31 (92)	66 (94)
		CVE	SL	24 (75)	11 (72)	65 (80)			CVE	SL	0 (0)	100 (84)	0 (0)
			ML	6 (74)	7 (70)	87 (79)				ML	9 (80)	28 (81)	63 (86)
	VICS	SARD	SL	3 (90)	4 (88)	93 (95)		VICS	SARD	SL	8 (90)	32 (91)	60 (95)
			ML	40 (92)	4 (85)	56 (96)				ML	9 (90)	32 (91)	59 (95)
		CVE	SL	10 (80)	34 (81)	56 (85)			CVE	SL	7 (85)	79 (88)	14 (91)
			ML	1 (80)	40 (82)	59 (84)				ML	10 (86)	38 (87)	52 (90)
Llama3	Standard	SARD	SL	50 (55)	0 (0)	50 (57)	DeepSeek-Coder	Standard	SARD	SL	13 (72)	8 (70)	79 (76)
			ML	19 (54)	0 (0)	81 (57)				ML	5 (72)	1 (70)	94 (75)
		CVE	SL	20 (52)	16 (50)	64 (55)			CVE	SL	36 (60)	0 (0)	64 (63)
			ML	21 (52)	5 (50)	74 (55)				ML	7 (60)	0 (0)	93 (62)
	ICL	SARD	SL	1 (65)	20 (68)	79 (71)		ICL	SARD	SL	7 (78)	4 (75)	89 (82)
			ML	0 (0)	24 (68)	76 (71)				ML	2 (78)	0 (0)	98 (81)
		CVE	SL	12 (65)	45 (67)	43 (70)			CVE	SL	14 (70)	10 (68)	76 (76)
			ML	5 (65)	21 (66)	74 (69)				ML	9 (71)	2 (68)	89 (75)
	VICS	SARD	SL	0 (0)	18 (74)	82 (77)		VICS	SARD	SL	0 (0)	0 (0)	100 (89)
			ML	0 (0)	21 (74)	79 (77)				ML	2 (85)	1 (85)	97 (90)
		CVE	SL	9 (70)	41 (72)	50 (77)			CVE	SL	25 (75)	0 (0)	75 (79)
			ML	8 (70)	29 (72)	63 (76)				ML	14 (76)	4 (74)	82 (79)
CodeLlama	Standard	SARD	SL	0 (0)	0 (0)	100 (34)	N/A	GTh	SARD	SL	1	85	15
			ML	0 (0)	0 (0)	100 (34)							
		CVE	SL	23 (18)	4 (15)	73 (21)							
			ML	56 (19)	2 (12)	42 (22)				ML	0	0	100
	ICL	SARD	SL	10 (45)	0 (0)	90 (50)							
			ML	2 (45)	0 (0)	98 (49)							
		CVE	SL	28 (60)	5 (58)	67 (66)			CVE	SL	18	72	10
			ML	19 (62)	0 (0)	81 (65)							
	VICS	SARD	SL	4 (52)	3 (50)	93 (57)							
			ML	4 (53)	0 (0)	96 (56)				ML	13	52	35
		CVE	SL	20 (65)	10 (62)	70 (70)							
			ML	18 (66)	0 (0)	82 (69)							

prevalent. When we look only at the edits that succeed (i.e., compile and manifest the intended weakness), replacements continue to dominate. Deletion-led success cases are relatively fewer. Notably, this LLM editing preference diverges from the ground-truth vulnerability-inducing changes, which rely more on removing code.

> **Finding 3**: LLMs exhibit a strong preference for replacement edits, especially for multi-line injections, diverging from ground-truth vulnerability patterns which tend to delete code.

The complexity of an injection (single-line vs. multi-line) influences the type of edits performed by LLMs. Compared to single-line edits, many models exhibit a higher proportion of addition edits in multi-line cases, a tendency that becomes even more pronounced in successful multi-line injections. This suggests that, for complex vulnerabilities, LLMs are more inclined to insert new vulnerable blocks of code than for injecting simpler (single-line) flaws. In contrast, the ground-truth (GTh) data shows that multi-line vulnerabilities are predominantly caused by replacements in the SARD dataset and by deletions in the CVE dataset, highlighting a distinct machine-driven pattern in how LLMs create complex vulnerabilities.

> **Finding 4**: LLMs favor different edit strategies for injecting vulnerabilities of different levels of complexity.

Table 6 breaks down LLMs' injection edit complexity per CWE category. To facilitate observations, only numbers >80% are highlighted. For example, with standard prompting, GPT-4o's CWE-190 injections in SARD consist of 86% single-line edits with 90% success rate and 14% multi-line edits with 92% success rate.

Overall, across all prompting strategies, the LLMs exhibit an inherent preference to perform multi-line injections. The distribution of edit complexities of the LLMs closely mirrors ground-truth vulnerability complexity patterns and remains overall consistent regardless of CWEs. This indicates that LLMs can well follow real-world vulnerability patterns and inject more complex and realistic vulnerabilities with multi-line edits.

> **Finding 5**: LLMs prefer performing multi-line vulnerability injections, which is close to the ground-truth patterns.

Notably, the results reveal substantial variations in success rates across both CWE categories and models. The LLMs performed best on vulnerabilities that can be introduced through local, syntactic edits and struggle as the task requires non-local, semantic reasoning about program state or object lifecycles.

For *arithmetic and boundary errors*, which can be injected via local edits, the models generally exhibited high injection success. For example, on CWE-190 in the SARD dataset, GPT-4o with VICS achieved an over 90% success rate. However, the success rate dropped for more complex *data-handling and control-flow vulnerabilities*, where multi-line edits are required; even though LLMs consistently attempt multi-line transformations, success remains modest, as seen for CWE-787. The most difficult cases involve *memory-management and lifecycle-related bugs*: on the CVE dataset, Llama-3 with VICS frequently used multi-line edits for CWE-416 and CWE-476, yet overall success remained low.

These findings collectively indicate that, despite improved prompting strategies, current LLMs are far more capable of injecting certain types of vulnerabilities than other types, highlighting their limitations in modeling deeper semantic program behaviors.

Table 6: LLMs' injection-edit complexity for different CWEs

LM	PS	Edits	SARD					CVE				
			CWE121	CWE190	CWE191	CWE134	CWE124	CWE125	CWE190	CWE416	CWE476	CWE787
GPT-4o	Standard	%SL	100 (93)	86 (90)	71 (88)	100 (92)	86 (89)	72 (73)	59 (68)	17 (70)	73 (75)	0 (0)
		%ML	0 (0)	14 (92)	29 (90)	0 (0)	14 (88)	28 (72)	41 (74)	83 (69)	27 (70)	0 (0)
	ICL	%SL	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	29 (80)	50 (85)	38 (81)	71 (83)	29 (79)
		%ML	100 (94)	100 (92)	100 (95)	100 (90)	100 (93)	71 (83)	50 (80)	62 (85)	29 (78)	71 (82)
	VICS	%SL	50 (95)	14 (91)	29 (92)	50 (94)	14 (90)	50 (90)	14 (86)	29 (89)	50 (88)	14 (85)
		%ML	50 (92)	86 (94)	71 (93)	50 (91)	86 (95)	50 (89)	86 (87)	71 (90)	50 (86)	86 (88)
Claude-3.5	Standard	%SL	10 (79)	0 (0)	0 (0)	0 (0)	5 (75)	20 (70)	39 (73)	13 (68)	33 (72)	0 (0)
		%ML	90 (78)	100 (80)	100 (76)	100 (79)	95 (77)	80 (72)	61 (69)	87 (74)	67 (70)	100 (68)
	ICL	%SL	19 (88)	21 (85)	12 (83)	11 (87)	15 (86)	22 (79)	33 (76)	20 (80)	28 (77)	12 (75)
		%ML	81 (85)	79 (87)	88 (88)	89 (84)	85 (86)	78 (80)	67 (77)	80 (75)	72 (79)	88 (78)
	VICS	%SL	15 (95)	13 (92)	6 (93)	6 (96)	11 (94)	19 (85)	26 (81)	15 (80)	13 (84)	9 (82)
		%ML	85 (94)	87 (95)	94 (92)	94 (96)	89 (93)	81 (82)	74 (84)	85 (81)	87 (85)	91 (83)
Llama3	Standard	%SL	3 (58)	0 (0)	0 (0)	0 (0)	0 (0)	31 (56)	18 (52)	9 (55)	33 (53)	0 (0)
		%ML	97 (55)	100 (57)	100 (58)	100 (54)	100 (56)	69 (52)	82 (55)	91 (56)	67 (53)	100 (51)
	ICL	%SL	62 (72)	72 (68)	0 (0)	96 (71)	94 (70)	22 (69)	31 (65)	38 (70)	41 (67)	23 (68)
		%ML	38 (69)	28 (72)	100 (68)	4 (70)	6 (67)	78 (67)	69 (70)	62 (66)	59 (68)	77 (65)
	VICS	%SL	42 (78)	63 (75)	0 (0)	94 (77)	86 (76)	22 (75)	17 (72)	0 (0)	0 (0)	27 (76)
		%ML	58 (75)	37 (78)	100 (76)	6 (74)	14 (77)	78 (73)	83 (76)	100 (0)	100 (0)	73 (74)
CodeLlama	Standard	%SL	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	20 (22)	63 (18)	27 (21)	60 (19)	0 (0)
		%ML	100 (35)	100 (32)	100 (36)	100 (33)	100 (34)	80 (19)	37 (23)	73 (18)	40 (20)	100 (21)
	ICL	%SL	48 (50)	43 (47)	0 (0)	33 (51)	43 (48)	27 (65)	38 (62)	27 (66)	33 (63)	60 (64)
		%ML	52 (48)	57 (51)	100 (49)	67 (47)	57 (50)	73 (63)	62 (66)	73 (62)	67 (65)	40 (61)
	VICS	%SL	35 (58)	48 (55)	0 (0)	18 (57)	33 (54)	18 (69)	46 (66)	0 (0)	0 (0)	14 (70)
		%ML	65 (55)	52 (58)	100 (56)	82 (54)	67 (57)	82 (67)	54 (70)	100 (0)	100 (0)	86 (68)
Qwen2.5	Standard	%SL	53 (73)	36 (68)	43 (70)	67 (72)	15 (71)	38 (48)	67 (45)	50 (47)	0 (0)	12 (44)
		%ML	47 (70)	64 (72)	57 (69)	33 (71)	85 (73)	62 (47)	33 (44)	50 (46)	100 (45)	88 (48)
	ICL	%SL	88 (85)	78 (81)	0 (0)	96 (84)	92 (82)	40 (60)	32 (64)	37 (61)	43 (63)	49 (62)
		%ML	12 (82)	22 (84)	100 (83)	4 (81)	8 (85)	60 (63)	68 (61)	63 (64)	57 (60)	51 (62)
	VICS	%SL	73 (90)	67 (87)	0 (0)	91 (88)	80 (86)	23 (69)	41 (66)	0 (0)	0 (0)	41 (70)
		%ML	27 (86)	33 (89)	100 (88)	9 (87)	20 (90)	77 (67)	59 (70)	100 (0)	100 (0)	59 (68)
DeepSeek-R1	Standard	%SL	19 (88)	11 (85)	42 (87)	71 (84)	20 (86)	21 (74)	17 (70)	0 (0)	38 (73)	10 (71)
		%ML	81 (85)	89 (87)	58 (86)	29 (84)	80 (88)	79 (71)	83 (73)	100 (70)	62 (74)	90 (72)
	ICL	%SL	53 (95)	37 (91)	0 (0)	78 (94)	62 (92)	0 (0)	0 (0)	11 (86)	0 (0)	0 (0)
		%ML	47 (92)	63 (94)	100 (93)	22 (91)	38 (95)	100 (83)	100 (86)	89 (84)	100 (85)	100 (82)
	VICS	%SL	32 (94)	29 (91)	34 (92)	92 (95)	65 (93)	0 (0)	27 (90)	0 (0)	0 (0)	31 (86)
		%ML	68 (92)	71 (94)	66 (93)	8 (90)	35 (91)	100 (87)	73 (89)	100 (0)	100 (0)	69 (88)
DeepSeek-Coder	Standard	%SL	10 (76)	10 (73)	0 (0)	17 (77)	12 (74)	53 (60)	60 (64)	17 (61)	50 (63)	50 (62)
		%ML	90 (74)	90 (77)	100 (75)	83 (73)	88 (76)	47 (63)	40 (61)	83 (60)	50 (64)	50 (62)
	ICL	%SL	18 (83)	29 (79)	0 (0)	31 (82)	28 (80)	24 (75)	20 (72)	13 (76)	30 (73)	45 (74)
		%ML	82 (80)	71 (83)	100 (81)	69 (79)	72 (82)	76 (73)	80 (76)	87 (74)	70 (72)	55 (75)
	VICS	%SL	11 (91)	21 (88)	0 (0)	37 (90)	28 (87)	12 (79)	35 (76)	0 (0)	0 (0)	23 (80)
		%ML	89 (88)	79 (90)	100 (89)	63 (87)	72 (91)	88 (77)	65 (80)	100 (0)	100 (0)	77 (78)
N/A	GTh	%SL	3	24	19	5	1	53	47	70	84	59
		%ML	97	76	81	95	99	47	53	30	16	41

Table 7: Effectiveness comparison with SOTA, dedicated baselines (SV: Stable Vulnerable, SNV: Stable Non-Vulnerable, UV: Unstable Vulnerable, UNV: Unstable Non-Vulnerable)

Approach	Dataset	%SV (success rate)	%SNV	%UV	%UNV
best with LLM (GPT-4o with VICS)	SARD	93	7	0	0
	CVE	88	6	6	0
VGX	SARD	30 (210%↑)	67	2	1
	CVE	28 (214%↑)	60	3	9
VulGen	SARD	21 (343%↑)	67	0	12
	CVE	20 (340%↑)	64	0	16

Finding 6: LLMs have varying effectiveness in injecting vulnerabilities of different types and complexities, and the variations are not consistently impacted by prompting strategies used.

5 RQ3: Comparison to SOTA Baselines

We compare the LLMs with **VulGen** [47] and **VGX** [45] as baselines, because they are representative SOTA techniques dedicated for vulnerability data generation. Both of them generate vulnerable samples by injecting vulnerabilities to input non-vulnerable code, a similar methodology to what we explore in this paper, hence directly comparable to our approach. Specifically, VulGen combines pattern-based and fine-tuned DL models for vulnerability injection. VGX is similar to VulGen but more advanced with semantics-aware contextualization and human-knowledge enhanced edit pattern formation. More details can be found in respective papers.

As Table 7 shows, the best LLM-based vulnerability data generation (GPT-4o with VICS) outperforms by 210% and 343% on the SARD dataset, and 214% and 340% on the CVE dataset, compared to the two SOTA baselines, respectively—↑ indicates the *percentage* of increases. While VGX holds an edge over its predecessor VulGen, consistent with their earlier comparison [45], both only managed to achieve a modest rate of success. By contrast, with our advanced prompting strategy VICS, the most capable LLM delivers a substantially higher yield of usable vulnerability data samples. In fact, with VICS, any of the other LLMs studied also notably surpasses these SOTA approaches specialized for the same task (see Table 3).

A key reason lies in the gap between the complexity of vulnerabilities and the critical limitations of both baselines. As Table 6 shows, a considerable portion of vulnerable samples in both datasets requires multi-line code changes to inject the vulnerabilities. Yet VGX and VulGen are both constrained to simple injection (single-line code changes). In contrast, the LLMs are naturally capable of multi-line injection, while also handling single-line cases well.

Moreover, LLM-based generation excels at noise suppression. The LLMs, with any prompting strategy, produced much fewer (mostly 30% or lower) non-vulnerable and/or unstable samples, whereas the SOTA baselines suffer from a great fraction (over 60%) of failed injections resulting in unusable samples.

Notably, advanced prompt engineering (through few-shot and VICS) pushed the LLMs to much higher levels of sample generation effectiveness on both datasets. This demonstrates that, beyond their base generation capacity, carefully crafted prompts are essential for boosting both the success and robustness of LLMs for both single- and multi-line vulnerability injection.

Finding 7: LLM-based vulnerability data generation achieves much higher success rates compared to existing approaches, with up to 210%-343% improvements, which VICS helped substantially.

6 RQ4: Augmenting DL-based Models

To assess the usefulness of LLM-generated samples, we use them to augment the training of DL-based vulnerability detection models and evaluate the performance improvement they can bring.

For rigorous effectiveness evaluation (RQ1), we manually labeled each LLM-produced sample. That may not be feasible/scalable in realistic use scenarios of such samples. Yet as shown in Table 3, LLMs did not always succeed in sample generation even in the best case (GPT-4o with VICS). Thus, we need an automated filter to reduce data noise, for which we use CodeQL [1], a widely used security-focused code analyzer. Only LLM-generated samples that are considered vulnerable by CodeQL are kept and used.

Augmentation Datasets. Specifically, from all the real-world non-vulnerable code samples in CVEFixes [8] and PrimeVul [21] combined, we randomly selected one to feed GPT-4o with VICS, keeping the resulting sample if it passes the CodeQL filter; we then repeated this process until we obtained 500 samples due to single-line injection and 500 due to multi-line injection. We made this decision for two reasons. First, considering the scale of generation (about 1,000 samples) in prior work [47] and the cost of LLMs, we aim at 1,000 generated samples to use eventually. Second, we want to assess the usefulness of single- and multi-line injection induced samples separately to further validate the merits of overcoming the single-line injection limitation of the baselines; for a fair comparison, an equal size of both sets is preferred. We also drop the CWE scope restrictions as there is no reason to restrict here where we evaluate the usefulness in a practical data-production mode.

For comparison, we feed all the input normal samples used for obtaining the 500 single-line-injected samples to VulGen and VGX, each producing a set of vulnerable samples. Thus, we obtained four augmentation datasets, noted as **GPT-single**, **GPT-multi**, **VulGen-set**, and **VGX-set**. Then, w.r.t the size ratio between vulnerable and

Table 8: Top-k accuracy (%) of DL models before (Base) and after augmentation using each augmentation dataset

DL Model	Metric	Base	GPT-multi	GPT-single	VulGen-set	VGX-set
FVD-DPM	Top-1	28.69	39.08 (36.21↑)	37.48 (30.64↑)	31.27 (8.99↑)	32.88 (14.60↑)
	Top-3	45.70	50.80 (11.16↑)	48.44 (5.99↑)	46.61 (1.99↑)	47.26 (3.41↑)
	Top-10	57.04	67.50 (18.33↑)	66.07 (15.83↑)	59.62 (4.52↑)	61.23 (7.35↑)
LineVul	Top-1	3.71	6.31 (70.08↑)	6.02 (62.26↑)	5.47 (47.44↑)	6.05 (63.07↑)
	Top-3	17.80	22.90 (28.65↑)	20.54 (15.39↑)	18.71 (5.11↑)	19.36 (8.76↑)
	Top-10	32.90	43.36 (31.79↑)	41.94 (27.48↑)	35.48 (7.84↑)	37.10 (12.76↑)

non-vulnerable sample set adopted in the baselines [45, 47] and the need to keep the class balance in the original DL models' training set, we took 2,000 non-vulnerable samples from the PrimeVul dataset [21] for its high quality and added them to each of the four augmentation datasets (each covers 140 CWEs).

Task Selection. We target *line-level* vulnerability detection as the downstream task, rather than coarse-grained vulnerability detection at function level, for two reasons. First, pinpointing the exact line(s) containing a flaw provides more insights into where/how vulnerabilities occur. Second, the fine-grained locations give developers immediate, actionable guidance for patching.

DL Model Selection. We select FVD-DPM [55] and LineVul [24], as they are SOTA DL-based line-level vulnerability detectors, while based on different architectures (hence favoring coverage). Also, they reported very high accuracy, intuitively not easy to improve further. LineVul is a Transformer-based model specifically designed for line-level vulnerability prediction. FVD-DPM [55] is a most recently published, fine-grained vulnerability detection model that employs a conditional diffusion probabilistic framework to predict vulnerable code at both slice-level and statement-level.

Procedure/Metrics. For each DL model, we add the augmentation dataset to its training/fine-tuning set. We then test the augmented model on a testing set built from the *VulTrigger* dataset [37]: we used its 310 samples where the vulnerable locations are within single functions per the working scope of both DL models. We consider VulTrigger as it offers a high-quality dataset that actually has correct labels for vulnerability locations while separately labeling patch locations—other relevant datasets often treat patch locations as vulnerable locations, which is not rigorous. In the resulting testing set, the average number of lines per code sample is 141.59, reasonably challenging to localize flaws from.

We report *Top-1*, *Top-3*, and *Top-10* accuracy as done typically in prior line-level vulnerability detection.

Results. Table 8 summarizes how different augmentation datasets boost the vulnerability analysis performance for both DL models, where best improvement across the four datasets is highlighted. Numbers in parentheses indicate differences from base accuracies. For instance, when LLM's multi-line-injected samples are added, the top-1 accuracy of FVD-DPM increases by over one-third compared to the original model, and the improvement nearly doubles at the more lenient top-3 and top-10 thresholds relative to what VGX and VulGen can help. LineVul, which begins with a much lower base accuracy, sees an even greater increase: multi-line-injection augmentation multiplies its localization success several-fold.

By contrast, LLM's single-line-injected data augmentation, although clearly raising the base accuracies, achieves notably lower improvement than multi-line samples. This contrast indicates the lesser potential of merely simple (single-line) injections as the two SOTA data-generation baselines are limited to.

Between the two baselines, VGX-produced data outperformed VulGen's generated samples, consistent with earlier comparisons [45]. Meanwhile, the LLM-produced data shows notable advantages over both SOTA baselines. In fact, even when constrained to single-line injections, the LLM's advantages still hold strongly.

Finding 8: LLM-generated vulnerability samples substantially enhance DL-based vulnerability detection models, especially for multi-line injections. Over SOTA data-generation baselines, the LLM shows clear merits, even just for single-line injections.

7 RQ5: Augmenting LLMs

In RQ4, we examine if vulnerable data generated by our method can improve traditional smaller-scale DL models (via re-training). To further assess the usefulness of the LLM-generated data, in this RQ we utilize the data to augment (other) LLMs for vulnerability analysis. We use the same augmentation datasets, same testing set, same evaluation metrics, and for the same task, all as for RQ4.

Detection LLM Selection. As we used GPT-4o to generate GPT-multi and GPT-single, we consider (all) the other LLMs in our study as (line-level) vulnerability detection models.

Procedure. We augment each detection LLM via retrieval augmented generation (RAG). Specifically, we build an RAG knowledge base using each augmentation dataset. In the base experiments, we directly query the LLM for the vulnerable lines given the testing sample (i.e., standard prompting). In the augmented experiments, we use BM-25 [53], a widely used algorithm for code similarity comparison [26, 42], to retrieve the three most similar (to the testing sample) samples and their vulnerable lines from the RAG knowledge base to build the exemplars. Then, we use these exemplars to conduct few-shot learning for the detection LLM to query the vulnerable lines for the same testing sample.

Results. Table 9 shows the (base) accuracies of each detection LLM and how much the base performance is changed (↑ for increases and ↓ for decreases) by using each of the augmentation datasets to augment the LLM. Overall, the GPT-4o generated data pushed up all the detection LLMs' base accuracies notably. In particular, the LLM's multi-line-injected samples (GPT-multi) offered much greater gains than single-line-injected samples. In fact, GPT-multi not only doubles localization accuracy at the strictest (top-1) threshold but also amplifies relative improvements even further at more lenient (top-3 and top-10) cutoffs. This pattern holds true from heavyweight architectures like Llama3 and Claude-3.5—where the headroom for improvement is already smaller—to lighter-weight systems such as Qwen2.5 and DeepSeek-Coder, which exhibit the steepest learning curves when provided with richer examples.

While the LLM's single-line-injected samples did improve localization over the original detection LLMs' performance, especially at the more forgiving top-10 level, its incremental uplift becomes less pronounced as the threshold tightens. This contrast consolidates our earlier observation in RQ4 about the cruciality of more complex (multi-line) injections in vulnerability data generation.

The data produced by two SOTA data-generation baselines made much smaller improvements overall, compared to the LLM-generated datasets. In a number of cases, their (VulGen/VGX's) generated data even decreases the base performance, mainly due to the much higher noise levels in the generated samples (see Table 6)—e.g.,

Table 9: Top-k accuracy (%) of LLMs before (Base) and after augmentation using each augmentation dataset

Model	Base			GPT-multi			GPT-single			VulGen-set			VGX-set		
	Top-1	Top-3	Top-10	Top-1	Top-3	Top-10	Top-1	Top-3	Top-10	Top-1	Top-3	Top-10	Top-1	Top-3	Top-10
GPT-4o	2.3	11.6	27.4	–	–	–	–	–	–	–	–	–	–	–	–
Claude-3.5	2.3	10.3	22.9	3.2 (0.9↑)	16.8 (6.5↑)	28.5 (5.6↑)	3.2 (0.9↑)	9.7 (0.6↓)	25.8 (2.9↑)	1.6 (0.7↓)	8.4 (1.9↓)	14.2 (8.7↓)	1.9 (0.4↓)	10.0 (0.3↓)	17.1 (5.8↓)
Llama3	3.5	11.6	14.8	4.5 (1.0↑)	13.9 (2.3↑)	31.3 (16.5↑)	3.2 (0.3↓)	12.3 (0.7↑)	25.8 (11.0↑)	2.3 (1.2↓)	6.8 (4.8↓)	15.5 (0.7↑)	2.9 (0.6↓)	8.4 (3.2↓)	18.7 (3.9↑)
CodeLlama	0.3	6.1	7.7	2.3 (2.0↑)	7.7 (1.6↑)	14.8 (7.1↑)	1.9 (1.6↑)	6.5 (0.4↑)	12.9 (5.2↑)	1.9 (1.6↑)	5.2 (0.9↓)	9.0 (1.3↑)	2.3 (2.0↑)	6.1 (0.0↑)	10.6 (2.9↑)
Qwen2.5	2.3	8.7	12.6	3.2 (0.9↑)	12.6 (3.9↑)	22.6 (10.0↑)	2.6 (0.3↑)	9.7 (1.0↑)	19.4 (6.8↑)	2.3 (0.0↑)	5.8 (2.9↓)	12.3 (0.3↓)	2.6 (0.3↑)	7.1 (1.6↓)	14.5 (1.9↑)
DeepSeek-R1	1.3	5.5	11.0	1.6 (0.3↑)	9.7 (4.2↑)	14.8 (3.8↑)	1.3 (0.0↑)	6.5 (1.0↑)	11.3 (0.3↑)	1.3 (0.0↑)	3.9 (1.6↓)	6.5 (4.5↓)	1.6 (0.3↑)	4.8 (0.7↓)	7.4 (3.6↓)
DeepSeek-Coder	0.0	1.3	2.3	1.6 (1.6↑)	4.8 (3.5↑)	10.0 (7.7↑)	0.6 (0.6↑)	3.9 (2.6↑)	9.4 (7.1↑)	0.6 (0.6↑)	1.9 (0.6↑)	4.2 (1.9↑)	1.0 (1.0↑)	2.9 (1.6↑)	5.5 (3.2↑)

the incorrectly labeled samples could mislead the detection LLMs during the few-shot learning.

Finding 9: The LLM-generated datasets generally improved all the detection LLMs' localization performance considerably, especially with its multi-line-injected samples, substantially more than the SOTA data-generation baselines could improve.

8 Discussion

In this section, we provide insights and recommendations based on our experimental results across the research questions.

8.1 Potential and Gaps of LLMs for Vulnerability Data Generation

To explore how effectively LLMs inject new vulnerabilities into non-vulnerable code, we conducted extensive experiments on both synthetic (SARD) and real-world (CVE) datasets. We reveal strengths and limitations in LLM-generated vulnerability samples with the following observations:

(1) Balancing Vulnerability vs. *Stability.* While some LLMs effectively inject vulnerabilities, their edits often disrupt program logic, resulting in unstable code. In contrast, models such as GPT-4o tend to better preserve functionality while introducing security flaws. This highlights a clear trade-off between achieving high vulnerability-injection success rates and maintaining code (functional) semantics. Moreover, the performance of LLMs varies across different CWE types, suggesting that selecting an appropriate model based on the target vulnerability category may be fruitful.

(2) Prompting Sensitivity. The prompting strategy has a decisive effect on injection quality. Standard, zero-shot prompts frequently produce incomplete or flawed edits that fail to introduce valid vulnerabilities, whereas both few-shot ICL and the VICS approach substantially raise success rates and effectively eliminate unstable outputs. In practice, this means that careful prompt engineering is critical to achieving consistent, reliable vulnerability injections.

8.2 Comparison of LLMs to Existing Vulnerability Data Generation Approaches

Our evaluation results corroborate that (even general-purpose) LLMs, especially when guided by advanced prompting, can outperform existing, SOTA dedicated vulnerability injection techniques like VulGen and VGX tremendously, with success rate improvements of up to 210%-343%. Among other reasons, this superiority is notably contributed by their inherent ability to perform coherent, multi-line edits, allowing them to generate more realistic and complex vulnerabilities that mirror the patterns of real-world flaws—a capability that overcomes the critical limitations of existing approaches to simple, single-line injections.

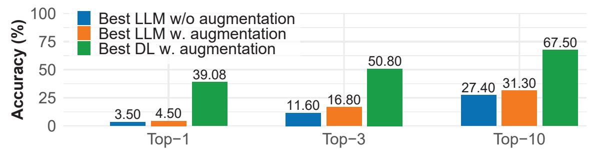

Figure 6: Comparison between LLM without augmentation, LLM with augmentation, and DL model with augmentation for line-level vulnerability detection, in best cases.

The practical value of the resulting, higher-quality data is substantial: when used to augment DL-based detectors (RQ4), LLM-generated samples lead to far greater improvements in detection accuracy than data from the existing SOTA generators. Similarly, when integrated into RAG frameworks to enhance other LLMs (RQ5), this data serves as a powerful knowledge source that dramatically boosts vulnerability localization. In essence, LLMs produce fundamentally better samples that are more diverse and context-rich, making them a far more effective resource for augmenting both DL- and LLM-based security analysis models.

8.3 Comparison of Augmented DL Models & LLMs with Original LLMs

Our study reveals a crucial insight: augmenting existing models with high-quality, LLM-generated data can yield vulnerability detection capabilities superior to those of even the most advanced LLMs performing the same task directly. Figure 6 demonstrates this with a comparison between the best LLM without augmentation, best LLM with augmentation, and best DL model with augmentation. The contrasts suggest that the more effective use of LLMs in this domain may be as data generators than as direct detectors.

Specifically, as shown in Table 8, the FVD-DPM model, when augmented with our GPT-multi dataset, achieves a Top-10 localization accuracy of 67.50%. This performance largely surpasses that of the best LLM unaugmented in our experiments, where GPT-4o reached a Top-10 accuracy of only 27.4% (Table 9). The performance of the augmented DL model is more than double that of the best-performing LLM, underscoring that a focused model trained on diverse, high-quality examples can develop a far more nuanced and effective capability for this specific task than a general-purpose LLM can achieve through in-context reasoning alone.

This principle of leveraging generated data also holds true within the LLM ecosystem itself. As shown in Table 9, when Llama-3 is augmented with the GPT-multi dataset through RAG, its Top-10 accuracy is up to 31.3%, outperforming the best unaugmented LLM analysis from GPT-4o (27.4%). This confirms that providing an LLM with access to a curated knowledge base of high-quality vulnerability examples is significantly more effective than relying solely on its internal, pre-trained knowledge for directly serving the same vulnerability analysis.

Taken together, these findings suggest a powerful and efficient paradigm. Rather than relying on a single LLM directly used for vulnerability detection, a more effective strategy involves a "distillation" of knowledge. In this approach, the generative prowess of a state-of-the-art model like GPT-4o is used to create a rich, high-fidelity dataset. This dataset can then be used to train or augment more specialized models like a DL-based detector such as FVD-DPM or a different LLM such as Llama-3 in a RAG setup. Such findings emphasize the usefulness of LLM-based vulnerability data generation, pointing toward a direction of more scalable and powerful automated vulnerability analysis.

8.4 Applicability to Other Languages

While our study is focused on C code, our contributions may extend beyond that, although the degree of applicability to another programming language $\mathcal{L}$ depends on the prompting strategy used.

Under *standard prompting*, the study process/methodology remains unchanged for $\mathcal{L}$; the primary variation lies in how effectively the LLM can inject vulnerabilities into code written in $\mathcal{L}$ when given direct instructions. Because this strategy requires no language-specific exemplars, its success is driven largely by the LLM's inherent cross-language code generation capability.

For *few-shot in-context learning (ICL)*, the process also generalizes naturally, but it requires normal code snippets and paired vulnerable versions written in $\mathcal{L}$. The resulting performance would depend on how well the LLM leverages these few-shot demonstrations to perform controlled vulnerability injection in $\mathcal{L}$.

Our *VICS prompting* strategy follows the same overall approach as the few-shot ICL, but with an additional layer: each exemplar includes LLM-generated reasoning describing the vulnerability-introducing change semantics. In particular, to apply VICS to $\mathcal{L}$, one would prepare exemplars in $\mathcal{L}$ along with the corresponding semantic rationales. The LLM then uses these exemplars to guide its injection behavior. Again, performance will vary depending on how well the model can transfer and operationalize this explicit reasoning within the syntax and idioms of $\mathcal{L}$.

Overall, our methodology is extensible to other programming languages as long as relevant code samples in those languages are available. While the absolute effectiveness may differ across languages, the increasing generalizability of frontier LLMs, particularly their strong multilingual code competencies, suggests that similar potential should hold for languages beyond C.

8.5 Threats to Validity

One internal threat to validity of our results lies in our manual evaluation: the results are subject to human biases/errors. To mitigate this threat, we employed a rigorous inter-rater negotiated agreement protocol and evaluated the reliability of the process.

Another internal threat is that to evaluate the usefulness of our LLM-generated data in an automated, production mode, we used CodeQL as a filter to improve the data quality. Yet CodeQL is known to suffer both imprecision and imperfect recall, which may have led to misled observations regarding the comparative merits of datasets generated by different approaches. The ethics controls in the studied LLMs that cannot be fully disabled may have occluded their real potential, confounding our results.

The exemplar selection in ICL and VICS strategies may be subject to the randomness of LLMs. Thus, we ran pilot tests on subsets of our datasets, showing little variation across multiple runs per LLM. Thus, for the official study, we used single-run exemplar selection to manage LLM costs, which may still suffer from instability, though.

A main external validity threat lies in our limited coverage of LLMs and size of input and testing sample sets. For instance, the SARD and CVE datasets used may not fully reflect real-world vulnerabilities. Thus, our findings and conclusions may not necessarily generalize to all LLMs and any input/testing sample.

9 Related Work

DL-based vulnerability detection. Numerous such approaches have emerged in recent years [13, 24, 36, 38, 40, 54, 60, 64]. Nong et al. [48] noted that many of these exhibit a gap between reported performance and real-world accuracy, often due to limited training data quality and size. To address data imbalance, Saikat et al. [11] proposed minority oversampling, while later studies emphasized the effectiveness of data augmentation [46]. Our work also explores data augmentation to enhance downstream vulnerability detection.

Vulnerability data generation. Beyond DL-based approaches [10, 45, 47, 49], VulScribeR [18] introduced an LLM-based approach using RAG to produce both single- and multi-line vulnerabilities. In comparison, we explored a novel prompting strategy, differential merits of multi- versus single-line injections, and the usefulness of generated data for augmenting not only DL models but also LLMs.

LLMs for code generation. LLMs have been applied to bug repair [20, 23, 33, 34, 59, 61, 62] and vulnerability patching [6, 41, 43, 50, 52]. Recent work has also explored using LLMs to inject bugs for benchmarking [31]. In contrast, we leverage LLMs to generate realistic vulnerable code for data augmentation and evaluate its impact on improving downstream vulnerability detection models.

10 Conclusion

We investigate vulnerability data generation using seven diverse and widely adopted LLMs. Through experiments on both synthetic and real-world datasets, we demonstrate the strong potential of LLMs in generating vulnerable code samples. Our findings show that LLMs can effectively address key limitations of existing vulnerable-sample generation techniques, producing higher-quality samples and significantly enhancing downstream tasks, specifically, improving the performance of DL-based vulnerability line-level detection models and RAG-based LLM vulnerability localization. At the same time, we identify and discuss several challenges and limitations in using LLMs for vulnerability injection, offering insights and actionable recommendations for developing more effective LLM-based vulnerability data generation methods.

Our entire study artifact, including used and resulting datasets, is available at https://figshare.com/s/20d2cae2cda2142999cb.

Acknowledgment

We thank the reviewers for their constructive comments, which helped us improve our original manuscript. This research was supported by the Army Research Office (ARO) via grant W911NF-21-1-0027, National Science Foundation (NSF) via grant CCF-2505223, and Office of Naval Research (ONR) via grant N000142512252.

References

[1] 2021. CodeQL Documentation. https://codeql.github.com/docs/.

[2] 2022. LLM Leaderboard - Comparison of over 100 AI models from OpenAI, Google, DeepSeek and others. https://artificialanalysis.ai/leaderboards/models.

[3] 2023. Meta Code Llama. https://www.llama.com/code-llama/.

[4] 2023. Meta-Llama-3-70B. https://huggingface.co/meta-llama/Meta-Llama-3-70B.

[5] 2024. Introducing Claude-3.5-Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet.

[6] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2023. Fixing Hardware Security Bugs with Large Language Models. *arXiv preprint arXiv:2302.01215* (2023).

[7] Andrew Austin, Casper Holmgreen, and Laurie Williams. 2013. A comparison of the efficiency and effectiveness of vulnerability discovery techniques. *Information and Software Technology* 55, 7 (2013), 1279–1288.

[8] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. 30–39.

[9] Paul E Black et al. 2017. SARD: A Software Assurance Reference Dataset. https://samate.nist.gov/SARD/. In *Anonymous Cybersecurity Innovation Forum.()*.

[10] Haipeng Cai, Yu Nong, Yuzhe Ou, and Feng Chen. 2023. Generating Vulnerable Code via Learning-Based Program Transformations. In *AI Embedded Assurance for Cyber Systems*. Springer International Publishing, 123–138.

[11] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering (TSE)* 48, 9 (2021), 3280–3296.

[12] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 654–668.

[13] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* 49, 1 (2022), 147–165.

[14] CNBC. 2019. DDoS attack caused interruptions in power system operations. https://www.cnbc.com/2019/05/02/ddos-attack-caused-interruptions-in-power-system-operations-doe.html.

[15] CNN. 2021. US warns hundreds of millions of devices at risk from newly revealed software vulnerability. https://www.cnn.com/2021/12/13/politics/us-warning-software-vulnerability/index.html.

[16] Roland Croft, M Ali Babar, and Mehdi Kholoosi. 2023. Data Quality for Software Vulnerability Datasets. *arXiv preprint arXiv:2301.05456* (2023).

[17] Cybersecurity and Infrastructure Security Agency (CISA), Open Source Security Foundation (OpenSSF), FBI, Australian Signals Directorate, and Canadian Cyber Security Center. 2024. Exploring Memory Safety in Critical Open Source Projects. https://www.cisa.gov/resources-tools/resources/exploring-memory-safety-critical-open-source-projects. Accessed: 2025-07-10.

[18] Seyed Shayan Daneshvar, Yu Nong, Xu Yang, Shaowei Wang, and Haipeng Cai. 2025. VulScribeR: Exploring RAG-based Vulnerability Augmentation with LLMs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2025).

[19] DARPA. 2023. DARPA AI Cyber Challenge Aims to Secure Nation's Most Critical Software. https://www.darpa.mil/news-events/2023-08-09.

[20] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. 2023. Fixing Rust Compilation Errors using LLMs. *arXiv preprint arXiv:2308.05177* (2023).

[21] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624* (2024).

[22] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*. 508–512.

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

[24] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: a transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*. 608–620.

[25] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 935–947.

[26] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R Lyu. 2023. What makes good in-context demonstrations for code intelligence tasks with llms?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 761–773.

[27] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-R1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[28] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[29] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*. 596–607.

[30] Huntsman. 2023. Cyber risks in transportation sector. https://huntsmansecurity.com/blog/cyber-risks-in-transportation-sector/.

[31] Ali Reza Ibrahimzada, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand. 2023. Automated bug generation in the era of large language models. *arXiv preprint arXiv:2310.02407* (2023).

[32] Information Technology Laboratory at NIST. 2023. National Vulnerability Database (NVD) Dashboard. https://nvd.nist.gov/general/nvd-dashboard.

[33] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with LLMs. *arXiv preprint arXiv:2303.07263* (2023).

[34] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with LLMs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.

[35] Peng Li and Baojiang Cui. 2010. A comparative study on software vulnerability static analysis techniques and tools. In *International Conference on Information Theory and Information Security*. 521–524.

[36] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 292–303.

[37] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. 2024. On the effectiveness of function-level vulnerability detectors for inter-procedural vulnerabilities. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1–12.

[38] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 19, 4 (2021), 2244–2258.

[39] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[40] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. 2023. VulChecker: Graph-based Vulnerability Localization in Source Code. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6557–6574. https://www.usenix.org/conference/usenixsecurity23/presentation/mirsky

[41] Emanuele Antonio Napoli and Valentina Gatteschi. 2023. Evaluating ChatGPT for Smart Contracts Vulnerability Correction. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 1828–1833.

[42] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2450–2462.

[43] Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. 2024. Chain-of-thought Prompting of Large Language Models for Discovering and Fixing Software Vulnerabilities. *arXiv preprint arXiv:2402.17230* (2024).

[44] Yu Nong, Haipeng Cai, Pengfei Ye, Li Li, and Feng Chen. 2021. Evaluating and comparing memory error vulnerability detectors. *Information and Software Technology* 137 (2021), 106614.

[45] Yu Nong, Richard Fang, Guangbei Yi, Kunsong Zhao, Xiapu Luo, Feng Chen, and Haipeng Cai. 2024. VGX: Large-Scale Sample Generation for Boosting Learning-Based Software Vulnerability Analyses. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 149:1–149:13.

[46] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2022. Generating realistic vulnerabilities via neural code editing: an empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1097–1109.

[47] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2023. VulGen: Realistic Vulnerable Sample Generation via Pattern Mining and Deep Learning. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 2527–2539. doi:10.1109/ICSE48619.2023.00211

[48] Yu Nong, Rainy Sharma, Abdelwahab Hamou-Lhadj, Xiapu Luo, and Haipeng Cai. 2022. Open science in software engineering: A study on deep learning-based vulnerability detection. *IEEE Transactions on Software Engineering (TSE)* 49, 4 (2022), 1983–2005.

[49] Yu Nong, Haoran Yang, Feng Chen, and Haipeng Cai. 2024. VinJ: An Automated Tool for Large-Scale Software Vulnerability Data Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 567–571.

[50] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2025. APPATCH: Automated Adaptive Prompting Large Language Models for Real-World Software Vulnerability Patching. In *34th USENIX Security Symposium (USENIX Security 25)*. 4481–4500.

[51] Office of the National Cyber Director (ONCD) and IBM. 2024. Memory-Safe Programming Languages Could Eliminate 70% of Security Bugs. https://www.ibm.com/think/news/memory-safe-programming-languages-security-bugs. Accessed: 2025-07-10.

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

[53] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[54] Deepika Shanmugasundaram, Pallavi Arivukkarasu, Huaming Chen, and Haipeng Cai. 2025. Deep Learning Representations of Programs: A Systematic Literature Review. *ACM Computing Surveys (CSUR)* (2025).

[55] Miaomiao Shao and Yuxin Ding. 2024. {FVD-DPM}: Fine-grained Vulnerability Detection via Conditional Diffusion Probabilistic Models. In *33rd USENIX Security Symposium (USENIX Security 24)*. 7375–7392.

[56] Team Ninja. 2024. IT Horror Stories: How Unpatched Software Hurts Businesses. https://www.ninjaone.com/blog/it-horror-stories-why-unpatched-software-hurts-business/.

[57] The Record. 2023. HHS warns of 'Citrix Bleed' attacks after hospital outages. https://therecord.media/hhs-warns-of-citrix-bleed-bug.

[58] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023).

[59] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. *arXiv preprint arXiv:2309.00608* (2023).

[60] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An image-inspired scalable vulnerability detection system. In *Proceedings of the 44th International Conference on Software Engineering*. 2365–2376.

[61] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*.

[62] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[63] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115* (2024).

[64] Haoran Yang, Yu Nong, Tao Zhang, Xiapu Luo, and Haipeng Cai. 2024. Learning to Detect and Localize Multilingual Bugs. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2190–2213.

[65] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems (NeurIPS)* 32 (2019).