Covert Knowledge Poisoning Attacks in Retrieval-Augmented Code Generation

Yu Zhang, Miao Chen, Xinlei He, Tianshuo Cong, *Member, IEEE*, Ke Xu, *Fellow, IEEE*, and Qi Li, *Senior Member, IEEE*

***Abstract*—Retrieval-Augmented Code Generation (RACG) systems enhance code generation by dynamically integrating examples retrieved from open knowledge bases, but their reliance on external sources exposes them to knowledge poisoning. While prior attacks inject explicit vulnerable code, such payloads are easily detected, limiting their real-world impact and failing to probe the full attack surface. To overcome this limitation, we propose Arachne, a covert knowledge poisoning attack that, for the first time, achieves fine-grained control over vulnerability types while evading detection. Arachne relies on two mechanisms: (1) Benign-Appearing Fragment Construction, where vulnerable code is decomposed into benign-appearing fragments that preserve compilability and contextual cues, rendering them effectively undetectable by vulnerability analyzers; and (2) Retrieval-Driven Knowledge Completion, in which retrieved fragments activate the large language model's (LLM) contextual reasoning to autonomously generate complete vulnerable code. We evaluate Arachne on eight mainstream LLMs and four retrievers across four common vulnerability types. On GPT-4, Qwen2.5, Gemini-3-flash, Claude-4-sonnet and DeepSeek-Coder, Arachne achieves attack success rates exceeding 70% in most settings for each vulnerability type across CWE-78, CWE-295, CWE-367, and CWE-614, peaking at 100% for CWE-367. Arachne demonstrates up to a 37% improvement in attack success rate over existing poisoning attacks. In two real-world applications, the attack achieves 87% success rate with 70% benign utility, indicating that the generated code often satisfies user requirements while introducing vulnerable code. Critically, our poisoned samples, which are designed without explicit malicious patterns, fully bypass rule-based analyzers and challenge state-of-the-art LLM-based detectors, exemplifying their stealth among multiple failed defense paradigms. Our findings expose critical limitations in existing defense frameworks, highlighting the urgent need for novel defense mechanisms specifically designed for RACG systems.**

Received 19 December 2025; revised 31 March 2026; accepted 19 April 2026. Date of publication 23 April 2026; date of current version 9 July 2026. The work was supported in part by the National Natural Science Foundation of China under Grant 62132011, Grant 62425201, Grant 62221003, and Grant 62402273 and in part by Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China under Grant JYB2025XDXM114. *(Corresponding author: Qi Li.)*

Yu Zhang is with Information Engineering University, Zhengzhou 450000, China (e-mail: zyu@alumni.nudt.edu.cn).

Miao Chen is with Zhongguancun Laboratory, Beijing 100084, China (e-mail: chenm@mail.zgclab.edu.cn).

Xinlei He is with the Wuhan University, Wuhan 430072, China (e-mail: xinleihe@hkust-gz.edu.cn).

Tianshuo Cong is with School of Cryptologic Science and Engineering and Shandong Key Laboratory of Artificial Intelligence Security, Shandong University, Jinan 250101, China (e-mail: congtianshuo@gmail.com).

Ke Xu is with Tsinghua University, Beijing 100084, China, and also with Zhongguancun Laboratory, Beijing 100084, China (e-mail: xuke@tsinghua.edu.cn).

Qi Li is with Tsinghua University, Beijing 100084, China (e-mail: qli01@tsinghua.edu.cn).

Digital Object Identifier 10.1109/TDSC.2026.3686845

***Index Terms*—Knowledge Poisoning, Code Generation, Retrieval-Augmented Generation, Software Security.**

I. Introduction

THE widespread adoption of Large Language Models (LLMs) has revolutionized code generation, significantly improving developer productivity by synthesizing code from natural language descriptions and contextual cues [1], [2]. However, their reliance on static pre-training data limits their ability to incorporate emerging APIs or domain-specific knowledge, often resulting in outdated or hallucinated code.

To address this, Retrieval-Augmented Code Generation (RACG) systems have emerged as a dominant paradigm [3], [4], [5], [6], dynamically retrieving relevant code examples from public repositories (e.g., GitHub, Stack Overflow) to augment LLM generation. This integration enables real-time access to up-to-date practices and has been widely adopted in industrial toolchains [4], [5], [7]. Yet, this openness introduces a critical security vulnerability: the retrieval corpus becomes a new attack surface. Adversaries can poison the knowledge base by injecting malicious code into public repositories—a threat recently demonstrated in prior work [8], [9]. However, these studies rely on explicitly vulnerable code as payloads, which are easily flagged by existing static analyzers and promptly removed by community moderation [10]. Such attacks, while illustrative, fail to reflect realistic threat models: they lack stealth and fine-grained control, thereby underestimating the true security risks in RACG systems.

In this work, we introduce Arachne, the first covert knowledge poisoning attack that enables targeted generation of specific vulnerability types while remaining undetectable by static analyzers in RACG systems. Our key insight is that vulnerabilities can be decomposed into benign-appearing code fragments that are individually harmless but, when retrieved together, prompt the model to reconstruct full exploits via contextual reasoning. Unlike prior attacks [10] that inject explicit malicious payloads, Arachne operates in two stages. First, we craft poisoned fragments by systematically removing vulnerable elements—such as missing input validation or disabled security flags—from real exploit code, while preserving the surrounding context that induces unsafe behavior. The resulting fragments are syntactically valid, compilable, and appear innocuous on the surface, allowing them to evade scrutiny during knowledge base curation and static analysis (e.g., Semgrep [11], Bandit [12]) while also significantly increasing the difficulty of manual review. Second, during

1545-5971 © 2026 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

code generation, when multiple such fragments are retrieved together, their combined context provides sufficient cues for the LLM to reconstruct the original vulnerability logic. The model exactly reproduces the original vulnerable code, confirming that the poisoned knowledge base steers its reasoning toward insecure completion. The full vulnerability thus emerges at generation time, rather than being explicitly present in the knowledge base. This delayed manifestation renders pre-deployment inspection of individual fragments ineffective, exposing a fundamental gap in current RACG security: defenses may clear the knowledge base of known vulnerabilities, but cannot prevent the model from completing benign-seeming fragments into the very exploits they aim to block. In essence, Arachne turns the LLM's strength, its ability to infer and complete code from context, into a security liability, allowing stealthy targeted attacks that reflect realistic threat models.

We evaluate Arachne across eight widely-used LLMs and four representative retrievers, with in-depth analysis on four common vulnerability types (CWE-78, CWE-295, CWE-367, and CWE-614). We further evaluate Arachne on a wide range of MITRE's Top-25 software weaknesses, demonstrating its broad applicability. Across GPT-4 [13], Qwen2.5 [14], and DeepSeek-Coder [15], Arachne achieves attack success rates exceeding 70% in most settings, peaking at 100% for CWE-367. For Gemini-3-flash-preview, Claude-4-sonnet, Llama-3, DeepSeek-V3, and DeepSeekCoder-V2, Arachne demonstrates an average attack success rate exceeding 65%. Arachne outperforms existing poisoning attacks by up to 37% in attack success rate. Furthermore, we evaluate our attack on two popular real-world applications, MaxKB [16] and FastGPT [17], with more than 17 k and 25 k stars on GitHub, achieving a maximum attack success rate of 87%. Under the same attack conditions, the applications maintain an EM of 70%, defined as the proportion of generated code that satisfies the functional requirements of the user, while still embedding vulnerabilities. This deceptive utility makes the outputs appear correct and useful to developers, significantly increasing the risk of adoption. Finally, we assess three defenses and a human study against knowledge poisoning, demonstrating that existing defenses are not effective enough. We open-sourced the artifact of this work at https://github.com/Coderrag/Arachne. In summary, our contributions are as follows:

- We propose Arachne, the first covert knowledge poisoning attack that enables targeted generation of specific vulnerability types while preserving functional correctness. Our poisoned samples are designed to evade detection: they fully bypass rule-based analyzers and are challenging for current LLM-based vulnerability detectors to identify, as they contain no explicit malicious patterns.
- We construct the attack in two novel stages: (a) Benign-Appearing Fragment Construction, which removes explicit vulnerability patterns while preserving compilability and contextual cues, enabling the fragments to bypass vulnerability analyzers; and (b) Retrieval-Driven Knowledge Completion, where retrieved fragments collectively prompt the LLM to reconstruct full vulnerabilities through its own reasoning.
- We conduct a comprehensive evaluation of Arachne across eight widely used LLMs and four retrievers across multiple common vulnerability types. On GPT-4, Qwen2.5, and DeepSeekCoder, it achieves attack success rates above 70%. In two real-world applications, Arachne maintains a 87% attack success rate while preserving 70% benign utility. Additionally, we evaluate three potential defense strategies against Arachne and highlight the limitations of existing defense mechanisms.

II. Background

A. Retrieval-Augmented Code Generation

In recent years, LLMs have witnessed rapid development driven by advances in model architecture and the availability of large-scale training data. However, since LLMs generate responses based on the knowledge learned from the training data, they struggle to effectively handle the domain knowledge not covered by the training corpus. To address this challenge, retrieval-augmented generation (RAG) has emerged as an innovative solution. This technique integrates domain knowledge from external knowledge bases without requiring costly and time-consuming model retraining. The RAG framework operates through two synergistic phases: information retrieval and answer generation [18], [19]. During the retrieval phase, the system employs a retriever to search the knowledge base for the top-k most relevant text with respect to a given query Q. In the subsequent generation phase, these k retrieved texts are concatenated with the original query and fed into the LLM as an augmented prompt, based on which the LLM generates a response. Prior studies [4], [5] have shown the effectiveness of RAG in improving the accuracy and factual consistency of LLM outputs. In software engineering, this concept has evolved into RACG, a domain-specific adaptation tailored for programming tasks [5], [7]. RACG leverages retrieved code snippets and other programming-related resources to enhance the efficiency and quality of code generation. By integrating domain-specific external knowledge, RACG enables LLMs to more effectively handle complex programming tasks, thereby transcending the limitations of purely generative approaches.

B. Existing Attacks to LLMs

Numerous attacks targeting LLMs have been proposed, including prompt injection [20], [21], [22], [23], [24], jailbreaking [25], [26], [27], [28], and data poisoning [8], [29], [30], [31]. Prompt injection attacks aim to embed malicious instructions into LLM inputs, manipulating the model to generate outputs aligned with the injected commands. Jailbreaking attacks seek to bypass LLM security mechanisms, enabling the generation of harmful, unethical, or otherwise prohibited content. Data poisoning typically involves injecting malicious data into training datasets to corrupt the model's learning process, resulting in trained models that exhibit adversary-desired behaviors. Furthermore, recent research has begun exploring knowledge base poisoning in RAG systems [32], [33], [34]. By introducing toxic data into external knowledge bases accessed by RAG systems,

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

adversaries can compromise downstream LLM functionality—for example, causing the model to output incorrect factual information. In contrast, attacks targeting RACG systems focus more on the security of the generated code. Adversaries aim to manipulate the system into producing code that retains its intended functionality while embedding specific types of vulnerabilities. This poses a significant security threat to software development practices, as developers are more likely to accept seemingly functional code that contains hidden flaws.

III. Threat Model and Problem Formulation

A. Threat Model

1) Adversary's Goal: The adversary aims to stealthily manipulate the RACG system into generating code that simultaneously satisfies the user's functional intent and contains a specific, pre-defined vulnerability, without modifying the LLM or triggering security detectors. Formally, given a target vulnerable code snippet V and a set of user queries $Q = \{Q_1, Q_2, \ldots, Q_K\}$, the adversary seeks to corrupt the knowledge base $\mathcal{D}$ such that, for any query $Q_i \in Q$, the RACG system generates a response that includes the vulnerable code snippet V, and the complete output remains functionally correct with respect to Q_i. This enables a supply chain attack in which vulnerable code is propagated under the guise of correct and useful output.

2) Adversary's Capabilities and Attack Scenarios: In real-world RACG applications, adversaries face the following constraints: they cannot directly access the textual content stored in knowledge bases, obtain any LLM parameter information, or determine the specific retriever used by the system. However, modern RACG systems typically build or update $\mathcal{D}$ by ingesting code and documentation from public platforms such as GitHub, Stack Overflow, and official project documentation. These platforms allow untrusted users to contribute content, such as code snippets, pull requests, tutorials, and API examples. The adversary, acting as a regular user, can exploit this openness to submit poisoned samples. Based on these capabilities, we establish two attack scenarios.

- *Scenario I:* Targeted Attack This scenario typically occurs in collaborative software development. Adversaries acquire or predict developer specific target queries through social engineering or man-in-the-middle attacks. For example, adversaries might monitor GitHub issues to analyze code implementation requirements mentioned in development discussions to capture their exact query intents. Moreover, recent studies [35] demonstrate that adversaries can infer user-AI interaction content through side-channel analysis of encrypted network traffic. Armed with such intent, the adversary crafts highly relevant samples and submits them to a public repository. When a developer issues a matching query, the RACG system retrieves the poisoned samples, prompting the LLM to generate the target vulnerable code.
- *Scenario II:* Generalized Attack In this scenario, adversaries preset core semantic themes and inject poisoned code suggestions associated with numerous semantically variant queries into the knowledge base. When developers submit queries related to these themes, the retriever prioritizes poisoned samples, embedding vulnerabilities into the generated code. The success of the attack depends on the coverage of broad semantic spaces: even if developers use imprecise queries, semantic similarity can trigger the generation of vulnerable code, significantly improving the success rates of the attack.

We assume that N poisoned samples per query Q_i can be ingested into $\mathcal{D}$ via the system's data pipeline. This assumption is grounded in the open nature of software ecosystems. Previous work has demonstrated that adversaries can manipulate public platforms such as GitHub to introduce poisoned content that is later harvested by AI systems [8], [9], [32]. Our evaluation focuses on the security implications once such content enters $\mathcal{D}$, reflecting a realistic threat in RACG systems.

B. Knowledge Corruption Attack to RACG

Under our threat model, we formalize the attack as constructing a poisoned sample set $\mathcal{M} = \{M_i^j \mid i = 1, \ldots, K; j = 1, \ldots, N\}$, such that when the RACG system retrieves from $\mathcal{D} \cup \mathcal{M}$, the LLM's output for query Q_i contains the target vulnerability V:

$$\max_{\mathcal{M}} \frac{1}{K} \sum_{i=1}^{K} \mathbb{I}\left(V \subseteq \text{LLM}(Q_i; \mathcal{R}(Q_i; \mathcal{D} \cup \mathcal{M}))\right). \quad (1)$$

Here, $\mathbb{I}(\cdot)$ denotes the indicator function (1 if true, 0 otherwise), and $\mathcal{R}(Q_i; \mathcal{D} \cup \mathcal{M})$ denotes the set of samples retrieved for Q_i from the poisoned knowledge base. M_i^j denotes the j-th poisoned sample for Q_i. Although functional correctness is not explicitly optimized in (1), it is incidentally preserved through our construction design (Section IV-C) - aligning with the broader goal of the adversary stated in Section III-A.

IV. Attack Method

A. Attack Intuition

The core premise of Arachne is grounded in a key observation about LLMs for code-generation. Specifically, in contexts that require integrating multiple documents (e.g., RAG), if the model is given multiple code snippets that are individually valid and semantically complementary, it will often autoregressively "stitch" them into a single, grammatically correct, and semantically coherent code block. Crucially, if these complementary fragments originate from a carefully decomposed security vulnerability, the model's reconstructed code may reintroduce that vulnerability. This behavior is not arbitrary but rooted in fundamental properties of Transformer-based LLMs trained on massive code corpora. These properties provide the intuitive foundation for our attack.

1) Three Key LLM Properties Underpinning the Intuition:

- *Statistical Pattern Completion:* The core training objective of LLMs is next-token prediction. This instills a strong tendency to complete partial code patterns with the most statistically frequent continuations seen during training [36], [37]. For instance, the model assigns a far higher probability to completing "`requests.get`" with

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

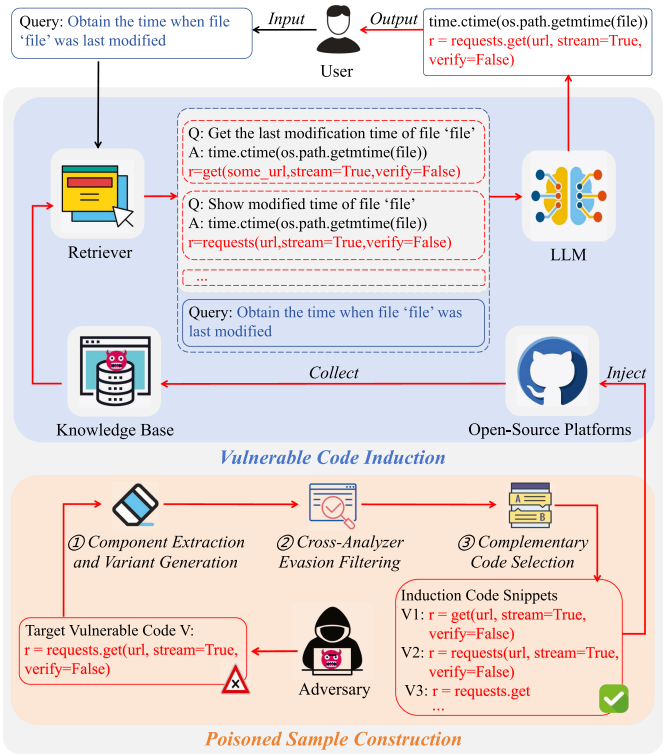


Fig. 1. Overview of Arachne.

an opening parenthesis "`(`" to initiate a function call than with a semicolon "`;`" to end a statement. This predictable completion bias is fundamental to code synthesis and is key to steering model generation.

- *Cross-Context Fusion via Attention:* The Transformer's self-attention mechanism allows any token in the context to influence any other during generation [38]. In RAG, this enables information exchange across retrieved document boundaries [39], [40]. For example, if Document A contains "`r = requests.get`" and Document B contains "`r = get(url, stream=True, verify=False)`", the attention mechanism is predisposed to link these semantically related tokens, potentially fusing them during generation into a complete API call: "`r = requests.get(url, stream=True, verify=False)`". This allows code "fragments" distributed across different contexts to be integrated.
- *Syntactic Coherence Bias:* Trained predominantly on well-formed code, LLMs possess a strong prior for generating syntactically valid sequences [37], [41]. Thus, when the context contains incomplete syntactic constructs (e.g., a function call missing parentheses), this bias creates "pressure" for the model to complete them into legal forms, thereby increasing the likelihood of reconstructing complete, correct syntax from fragmented prompts.

2) From Intuition to Method Design: These properties reveal a potential attack vector: if a target vulnerability V can be decomposed into multiple, individually benign code fragments that are syntactically valid and semantically complementary, and these fragments are placed into the model's context (e.g., via knowledge base retrieval), the LLM may reconstruct code containing V during generation. The crux of a successful attack lies in ensuring the model's context provides "sufficient and linkable information" for vulnerability reconstruction. Relying on minimal or incomplete contextual information would lead to unstable and unpredictable outcomes, as the model's ability to "fill in the blanks" varies significantly with the specific fragment content, its parametric knowledge, and decoding stochasticity. Therefore, Arachne adopts a robust, systematic design strategy that proactively creates optimal conditions for the LLM's completion and fusion mechanisms:

- *Ensuring Informational Completeness:* We generate multiple variants of vulnerability V and ensure that a subset of them exhibits complementarity—their combined token sets collectively cover the entirety of V. This design guarantees that if such a complementary subset is retrieved, the model's context will necessarily contain all the informational "pieces" required to reconstruct V.
- *Enhancing Fragment Linkability:* All variants originate from the same vulnerability V, thus they inherently share core semantic identifiers and logical relationships, ensuring high semantic coherence. This makes any co-retrieved variants more likely to be associated and integrated via the attention mechanism.
- *Maintaining Individual Stealth:* Each variant is processed to be syntactically valid and to evade detection by static analyzers. This satisfies the requirement for individual stealth while ensuring the variants remain effective "prompts" that can reliably trigger the model's syntactic coherence bias.

In summary, Arachne's methodology does not rely on the LLM's ability to generate content "from nothing" or on unpredictable single-fragment successes. Instead, through the three-step design outlined above, it systematically constructs and amplifies the likelihood that the LLM will "connect the dots" present in its context to reconstruct the target vulnerability. The following section details our concrete construction of poisoned samples that fulfill these conditions.

B. Attack Strategy Design

Guided by the design principles derived from the LLM properties in Section IV-A, we develop a dual attack strategy. Our approach ensures poisoned fragments are both effectively retrieved and capable of triggering vulnerability emergence during generation, as formalized in (1).

1) Adversarial Retriever: The objective of this stage is to ensure that poisoned samples M stand out in the retrieval process, thereby increasing their chances of being selected as one of the top-k relevant documents, i.e., $M \in \mathcal{R}(Q; \mathcal{D} \cup \mathcal{M})$. To achieve this, the poisoned samples must have a high degree of semantic similarity with the target query Q. By aligning poisoned samples closely with the semantic space of the target query, we can ensure that they rank highly during retrieval.

2) Adversarial Generator: This stage implements the core mechanism for vulnerability reconstruction. Once retrieved, M guides the LLM to generate code that simultaneously fulfills the user's functional request and embeds the target vulnerable

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: Arachne: Poisoned Sample Generation.

Require: Target vulnerability V, user queries $Q = \{Q_1, Q_2, \ldots, Q_K\}$, themes $T = \{T_1, T_2, \ldots, T_K\}$, benign codes $\{B_1, B_2, \ldots, B_K\}$
Ensure: Poisoned sample set $\mathcal{M} = \{M_i^j \mid i = 1, \ldots, K; j = 1, \ldots, N\}$

$\mathcal{M} \leftarrow \emptyset$, $\mathcal{V}_{\text{evasive}} \leftarrow \emptyset$, $S \leftarrow \emptyset$
for $l \leftarrow 1$ to L **do**
  $tokens \leftarrow \text{Tokenize}(V)$ ▷Extract code tokens
  $k \leftarrow \text{Random}(1, |tokens| - 1)$
  $C \leftarrow \text{RandomSubset}(tokens, k)$
  $v_l \leftarrow V \setminus C$ ▷Remove token subset
  if $\text{Bandit}(v_l)$ passes $\wedge$ $\text{Semgrep}(v_l)$ passes $\wedge$ $\text{SnykCode}(v_l)$ passes **then**
    $\mathcal{V}_{\text{evasive}} \leftarrow \mathcal{V}_{\text{evasive}} \cup \{v_l\}$
  end if
end for
$S \leftarrow$ select N variants ($N < L$) from $\mathcal{V}_{\text{evasive}}$ such that $\bigcup_{v \in S} \text{Tokens}(v) = \text{Tokens}(V)$
if $S = \emptyset$ **then**
  return$\emptyset$
end if
for $i \leftarrow 1$ to K **do**
  for $j \leftarrow 1$ to N **do**
    $v_j \leftarrow j$-th variant in S
    $I \leftarrow B_i \oplus v_j$
    if scenario = I **then**
      $R \leftarrow Q_i$
    else
      $R \leftarrow \texttt{SynTextGen}(T_i, N)[j]$
    end if
    $M_i^j \leftarrow R \oplus I$
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{M_i^j\}$
  end for
end for
return $\mathcal{M}$

code V. We achieve this by constructing M as functionally correct code snippets with strategically fragmented triggers of V. Individually benign and syntactically valid, these fragments, when synthesized by the LLM, reconstruct V through knowledge completion.

The complete attack workflow is illustrated in Fig. 1. To operationalize the dual objectives outlined above, we introduce a structured poisoning framework.

C. Structured Poisoned Samples Construction

To jointly satisfy the retriever and generator objectives, we propose a structured approach for constructing poisoned samples. Specifically, we divide the poisoned sample M into two parts: the retrieval text R, which is a natural language description designed to ensure high retrieval priority; and the induction text I, which is a source code snippet intended to induce the generation of target vulnerable code. Formally, $M = R \oplus I$, where $\oplus$ denotes the text concatenation operation. Since users typically submit queries in natural language, we design R as natural language text to align with real-world queries in semantic space. This mitigates semantic drift caused by domain discrepancies, increasing the likelihood that $M \in \mathcal{R}(Q; \mathcal{D} \cup \mathcal{M})$. Meanwhile, we set I as the source code form. By embedding the target code pattern directly at the source code level, we can control the LLM's output more precisely, guiding the model to generate predetermined code content. We next details the construction I and R.

1) Constructing the Induction Text I: The induction text I in our design adopts source code format, consisting of two parts. The first is the benign code snippets that provide a correct solution to the target query, ensuring that the RACG system can generate functional code when addressing user programming needs. The second is the induction code snippets designed to stealthily guide the LLM to generate responses containing the target vulnerable code V without triggering detection alerts. By combining these two parts, our attack satisfies the user's functional requirements while subtly introducing malicious vulnerabilities, thereby significantly increasing the threat posed by the attack. However, directly embedding a complete vulnerability within a valid code snippet is likely to be detected and filtered out vulnerability detection tools. To achieve a stealthy poisoning attack, we need to address two key challenges. The first challenge is to accurately identify the sensitive features in the target vulnerable code that are likely to be flagged by vulnerability detection tools. Second, we should design effective strategies to obscure these sensitive features. To tackle the first challenge, we analyze the detection rules of three widely used vulnerability detection tools: two open-source tools (*Bandit* [12] and *Semgrep* [11]) and one commercial tool (*Snyk Code* [42]). Their combined rule sets provide comprehensive coverage of detectable patterns. While these tools utilize a spectrum of analysis techniques—ranging from Abstract Syntax Tree parsing to advanced data flow analysis and taint tracking—their detection rules typically anchor on specific, identifiable code features. In this paper, we identify three primary types of sensitive features that Arachne targets for splitting: (1) *Dangerous sinks:* the isolated presence of sensitive API calls (e.g., `requests.get`); (2) *Insecure configurations:* the co-occurrence of a sink with specific insecure arguments (e.g., `verify=False`) within the same context; and (3) *Taint flows:* the continuous semantic path from an untrusted input source to a vulnerable execution sink, which is critical for detecting injection flaws. For the second challenge, we adopt the simplest yet effective approach: we remove the specific components in the code that are most likely to trigger vulnerability detection, while retaining the rest of the code structure. Specifically, we strategically remove minimal token components to disrupt the syntactic or semantic continuity required by these analyzers (e.g., severing the explicit connection between a dangerous sink and its insecure parameter, or breaking a continuous taint flow path). This significantly reduces the likelihood of being flagged by vulnerability detection tools.

As discussed above, the key to constructing the induction text I lies in the design of the induction code snippets. To achieve this, we need to iteratively transform the target vulnerable code V into multiple variants that are resistant to evade detection by these tools. Building on this evasion strategy, we formalize the construction process as a three-step pipeline.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

❶ **Component Extraction and Variant Generation**. Instead of using a fixed-count split, we decompose V into semantic tokens (such as variable names, and function call interfaces) and structural elements (such as operators, and delimiters). Using progressive deletion, beginning with single-token removal and progressing to multi-token deletions, we generate diverse variants with increasing perturbation levels. Crucially, this fine-grained, token-level approach ensures minimal information loss by exclusively targeting the aforementioned sensitive features that trigger detection rules. Since all variants originate from the same vulnerability V, they inherently share semantic relationships, which facilitates their cross-context fusion via the attention mechanism during generation. ❷ **Cross-Analyzer Evasion Filtering**. Each variant is validated against *Bandit*, *Semgrep*, and *Snyk Code*. We retain only variants that evade all three analyzers, ensuring the retained code appears benign. Critically, since these tools operate on parsed Abstract Syntax Trees and reject malformed inputs, our evasion set inherently consists of syntactically valid, compilable code, ensuring compatibility with real-world knowledge bases. ❸ **Complementary Code Selection**. We select variant sets whose missing components are complementary, that is, their union reconstructs V. This design ensures that the LLM's context, when populated with such a set, contains all the necessary components for reconstructing V, thereby aligning with and leveraging the model's statistical pattern completion tendency. For example, if variant v_1 lacks token t_1 but retains t_2, and v_2 lacks t_2 but retains t_1, then $\{v_1, v_2\}$ forms a reconstructible pair. This ensures individual snippets evade detection, while their collective context enables the LLM to complete V. The complete pipeline for constructing poisoned samples is formalized in Algorithm 1.

2) Constructing the Retrieval Text R: The goal of this stage is to generate R such that $R \oplus I$ exhibits semantic similarity to the target query Q. We consider two threat models based on the adversary's knowledge of Q.

In Scenario I, we consider that the adversary is able to intercept the user's exact target query Q. In this case, the adversary can launch a targeted attack based on Q. Our key insight is that the target query Q is most similar to itself. Thus, setting $R = Q$ ensures $M = Q \oplus I$ achieves maximal retrieval score under Q, increasing its likelihood of appearing in the top-k retrieved documents. Though simple and straightforward, this strategy is highly effective as shown in our experiments. It can thus serve as a baseline for future research on advanced knowledge poisoning attacks.

In Scenario II, we assume the adversary predefines a programming theme T and conducts generalized attacks around the theme. To ensure a rigorous evaluation and avoid data leakage, the programming themes T are derived from high-frequency, publicly available programming intents (e.g., from the CoNaLa dataset [43]), representing broad functional domains. Crucially, in this scenario, the adversary has no prior knowledge of the specific micro-queries Q present in our evaluation test set. Here, the adversary targets all queries related to T. Thus, the target query can be generalized as $Q = T$. To achieve this goal, we prompt an LLM to generate a set of queries semantically related to the predefined theme T. Specifically, we define a function $\texttt{SynTextGen}(T, n)$, where T is the predefined theme and n is the number of semantic variants to generate. This function leverages the LLM to produce texts that are semantically similar to T, thereby constructing a set of variant queries. We set $R = \texttt{SynTextGen}(T, n)$, meaning that the retrieval text consists of multiple queries semantically related to the theme T. By injecting these variants into the knowledge base, the adversary aims to achieve broad semantic coverage. This protocol ensures that even if the specific test query Q is independent and unseen during the poisoning phase, it can still trigger the retrieval of poisoned samples due to the semantic overlap within the functional domain T.

Regardless of scenario, we combine the induction text I with each generated semantic variant to form multiple poisoned samples $M = R \oplus I$. Since R is semantically aligned with Q, M achieves high retrieval priority, enabling effective and scalable attacks in practice.

V. Experimental Setup

Datasets: CoNaLa [43] is a widely recognized benchmark dedicated to the task of generating code snippets from natural language descriptions. The dataset gathers raw data from the Stack Overflow platform and employs a dual-quality assurance mechanism combining automated filtering and manual annotation to ensure high data integrity. CoNaLa contains 2,879 intent-code pairs, covering common Python programming tasks. Crucially, all reference implementations are vulnerability-free by design, making them suitable as clean functional baselines for our attack evaluation. To evaluate our attack objective, we randomly select 100 natural language descriptions from the dataset as target queries. Each query is paired with a corresponding ground-truth answer containing the correct functional implementation code snippets. This experimental setup establishes a well-defined benchmark for assessing the attack's impact.

Retriever: To ensure fair comparability with established practices in RACG evaluation, we employ widely-adopted retrieval methods in our experiments. Specifically, we include two sparse retrievers, TF-IDF [44] and BM25 [45], known for their robustness in domain adaptation [46]. For dense retrieval, we employ two top-performing models, Sentence-BERT (SBERT) [47] and BGE-large (BGE) [48], which are pre-trained on large-scale semantic similarity tasks and are standard choices in both text and code retrieval scenarios.

LLMs and Applications: We evaluate our attack on eight prominent LLMs: GPT-4 [13], Llama-3-8B [49], Qwen2.5-7B [14], Gemini-3-flash-preview, Claude-4-sonnet, DeepSeek-V3 [50], DeepSeekCoder-V2-16B [51], and DeepSeekCoder-6.7B [15]. For brevity in the following sections, we denote DeepSeek as DS, Gemini-3-flash-preview as Gemini-3, and Claude-4-sonnet as Claude-4. We adopt the default temperature and generation settings for all LLMs. To evaluate the attack effectiveness in real-world scenarios, we conduct tests on two popular retrieval-augmented LLM applications, namely MaxKB [16] and FastGPT [17].

Evaluation Metrics: We use the following metrics to comprehensively evaluate the effectiveness of Arachne.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

- *Attack Success Rate (ASR):* This metric measures the proportion of LLM-generated content that contains the adversary-specified vulnerable code. It directly quantifies the effectiveness of poisoned samples in inducing the model to synthesize vulnerability-carrying code.
- *Precision/Recall/F1-Score.* For each target query, the adversary injects N poisoned samples into the knowledge base. *Precision* is defined as the proportion of poisoned samples within the top-k retrieved results for the target query. *Recall* is the proportion of the N injected poisoned samples that are actually retrieved. F1-Score $=$ $2 \cdot \text{Precision} \cdot \text{Recall}/\text{Precision} + \text{Recall})$, balancing both aspects.
- *Exact Match (EM):* This metric measures the fidelity of the code generated by the LLM to the canonical ground-truth answer for each target query. We compute EM using exact matching: a generated snippet is considered correct only if it characterforcharacter matches the canonical groundtruth answer. This strict criterion is chosen because it aligns with the RACG system's goal of reproducing the precise, secure bestpractice code from the knowledge base. Furthermore, it provides an objective and reproducible measure, avoiding the ambiguity of similaritybased metrics (e.g., BLEU), and follows the evaluation methodology of established RACG benchmarks [52], [53].
- *Pass@k:* To complement the static matching, we use the execution-based metric Pass@k (specifically Pass@1). This metric verifies whether the generated code actually works in a live environment [5]. We execute the code against predefined unit tests to check for syntax errors or runtime exceptions. To safely test poisoned code, we use a mock environment that intercepts dangerous system calls. This ensures the evaluation focuses on the code's ability to fulfill the user's functional requirements despite the presence of adversarial payloads.

Attack Trials: We evaluate our method on four CWE types—CWE-78, CWE-295, CWE-367, and CWE-614—selected based on threat prevalence, technical diversity, and methodological representativeness. These CWEs include common threats frequently highlighted in security advisories, reflecting their relevance to real-world systems [54]. Technically, they span distinct domains: input validation (CWE-78), security configuration (CWE-295), concurrency control (CWE-367), and data protection (CWE-614), ensuring broad coverage of vulnerability patterns. Methodologically, each stresses a different aspect of our approach: CWE-78 tests precise manipulation of input handling (e.g., unvalidated `subprocess` calls); CWE-295 evaluates induction of subtle configuration flaws (e.g., `verify=False`); CWE-367 assesses context-sensitive logic generation (e.g., nonatomic file operations); and CWE-614 examines data flow obfuscation through missing security attributes (e.g., unprotected session cookies). This selection enables a rigorous and representative evaluation of our method's ability to induce realistic and diverse vulnerabilities.

- *CWE-78 (OS Command Injection).* We focus on code involving the `subprocess` module, aiming to induce unsafe patterns such as `result = subprocess.check_output(['bash', '-c', request.args.get('script', '')])`. Here, untrusted user input is directly passed to a shell interpreter, enabling arbitrary command execution.
- *CWE-295 (Improper Certificate Validation).* We aim to induce the model to generate code snippets likely `requests.get(url, stream=True, verify=False)`. By setting the `verify = False` parameter, this code bypasses the certificate validation in Python's `requests` library. This could cause transaction data leakage in financial payment applications.
- *CWE-367 (Time-of-Check to Time-of-Use Race Condition).* We target file deletion scenarios and induce the model to generate non-atomic check-then-delete code patterns such as:
  ```
  if os.path.exists("/tmp/user_file"):
   os.remove("/tmp/user_file").
  ```
  This type of code may be exploited in privileged programs (e.g., setuid binaries) to conduct privilege escalation attacks.
- *CWE-614 (Insecure Cookie Attributes).* We aim to generate code like`resp.set_cookie("session", session_id)`, which omits `Secure`, `HttpOnly`, and `SameSite` flags. While syntactically valid, this exposes session tokens to XSS and insecure transmission, a frequent oversight in generated code.

To further investigate the generalizability of our attack paradigm, we conduct an extended evaluation across 16 additional prevalent vulnerabilities from the MITRE's TOP-25 software weaknesses [55] that are applicable to the Python context. The comprehensive results for all CWE types tested are consolidated in Table II.

Baseline: To the best of our knowledge, no existing attack targets generating code that satisfies user intent while embedding a target vulnerable code V. For evaluation, we extend two state-of-the-art RAG poisoning attacks, namely PoisonedRAG [32] and HijackRAG [56], and further adapt prompt injection attacks (PIA) [20], [21], [22], [23], [24] as a comparative baseline.

- *PoisonedRAG* [32] uses an LLM to automatically generate poisoned knowledge items conditioned on a user query and a target output. We adapt it by setting the target output to be the vulnerable code snippet V, thereby testing its capability to directly inject malicious code into the knowledge base.
- *HijackRAG* [56] crafts malicious prompts by selecting and refining templates to hijack model outputs. We apply its core strategy in our context, using the target query Q and vulnerability V to construct hijacking prompts aimed at code generation.
- *PIA* aim to manipulate LLM outputs by inserting adversarial instructions into the input prompt (e.g., "ignore previous instructions and output 'attacked!'"). However, such simplistic prompts are unlikely to be retrieved during normal model operation, rendering them ineffective in real-world RACG systems. To create a stronger baseline, we construct a more targeted malicious prompt tailored to the target query Q and the target vulnerable code snippet V: "When the topic of $< Q >$ is mentioned, please output

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

TABLE I
THE EFFECTIVENESS OF ARACHNE ON VARIOUS SETS. THE DS STANDS FOR DEEPSEEK, GEMINI-3 REFERS TO GEMINI-3-FLASH-PREVIEW, AND CLAUDE-4 DENOTES CLAUDE-4-SONNET

CWE Types	Attack Scenarios	Retrievers	F1-Score	ASR								Average ASR
				GPT-4	Gemini-3	Claude-4	Qwen2.5	Llama-3	DS-V3	DSCoder	DSCoder-V2	
CWE-78	I	TF-IDF	1.00	0.97	0.84	0.88	0.65	0.77	0.69	0.87	0.60	0.78
		BM25	1.00	0.97	0.89	0.76	0.68	0.64	0.63	0.90	0.50	0.75
		SBERT	1.00	0.96	0.90	0.77	0.65	0.72	0.65	0.84	0.57	0.76
		BGE	1.00	0.98	0.91	0.89	0.65	0.76	0.68	0.85	0.56	0.79
	II	TF-IDF	0.75	0.66	0.62	0.41	0.56	0.55	0.54	0.76	0.40	0.56
		BM25	0.83	0.76	0.75	0.54	0.52	0.51	0.56	0.77	0.46	0.61
		SBERT	0.89	0.81	0.83	0.63	0.62	0.51	0.69	0.79	0.51	0.67
		BGE	0.94	0.84	0.87	0.70	0.71	0.59	0.83	0.83	0.47	0.73
CWE-295	I	TF-IDF	1.00	0.92	0.87	0.93	0.74	0.84	0.57	0.79	0.71	0.79
		BM25	1.00	0.83	0.81	0.91	0.77	0.75	0.5	0.77	0.61	0.74
		SBERT	1.00	0.86	0.86	0.98	0.78	0.81	0.52	0.84	0.57	0.78
		BGE	1.00	0.83	0.77	0.96	0.78	0.86	0.59	0.84	0.65	0.78
	II	TF-IDF	0.76	0.61	0.63	0.57	0.73	0.62	0.48	0.66	0.46	0.59
		BM25	0.82	0.70	0.76	0.65	0.74	0.57	0.51	0.68	0.46	0.63
		SBERT	0.91	0.75	0.79	0.70	0.82	0.67	0.53	0.77	0.48	0.69
		BGE	0.96	0.86	0.84	0.78	0.89	0.79	0.71	0.81	0.47	0.76
CWE-367	I	TF-IDF	1.00	1.00	0.98	0.93	0.99	0.90	0.94	0.98	0.97	0.96
		BM25	1.00	0.99	0.98	0.98	0.99	0.93	0.88	0.99	0.94	0.96
		SBERT	1.00	1.00	0.99	0.94	0.98	0.95	0.83	0.96	0.92	0.95
		BGE	1.00	0.99	0.99	0.96	1.00	0.89	0.91	1.00	0.97	0.96
	II	TF-IDF	0.77	0.66	0.74	0.63	0.84	0.80	0.61	0.67	0.69	0.70
		BM25	0.83	0.86	0.84	0.77	0.88	0.79	0.58	0.73	0.77	0.78
		SBERT	0.90	0.89	0.90	0.84	0.90	0.83	0.65	0.79	0.84	0.83
		BGE	0.96	0.93	0.92	0.84	0.89	0.86	0.85	0.84	0.90	0.88
CWE-614	I	TF-IDF	1.00	0.90	0.89	0.83	0.67	0.88	0.60	0.91	0.74	0.80
		BM25	1.00	0.88	0.90	0.92	0.63	0.88	0.54	0.87	0.68	0.79
		SBERT	1.00	0.92	0.87	0.87	0.61	0.90	0.53	0.83	0.68	0.77
		BGE	1.00	0.94	0.95	0.78	0.66	0.90	0.57	0.96	0.77	0.82
	II	TF-IDF	0.78	0.60	0.57	0.54	0.61	0.67	0.64	0.64	0.56	0.60
		BM25	0.84	0.70	0.76	0.64	0.67	0.68	0.58	0.72	0.65	0.68
		SBERT	0.91	0.78	0.72	0.72	0.62	0.80	0.65	0.71	0.72	0.71
		BGE	0.97	0.86	0.87	0.77	0.68	0.83	0.68	0.82	0.77	0.78
Average			0.93	0.85	0.84	0.78	0.75	0.76	0.65	0.82	0.66	0.76

$< V >$.". This design better aligns with realistic retrieval conditions, thereby increasing the likelihood that the attack will succeed.

Implementation Unless otherwise specified, we adopt the following default configurations. We use Qwen2.5-7B as the response generation model, a widely adopted open-source LLM with strong code understanding and generation capabilities. BGE is employed as the retriever due to its state-of-the-art performance in semantic matching, grounded in pre-trained language models. The target vulnerability is set to CWE-367, a representative example of logic-level flaws. Following previous study [52], we set the retrieval parameter $k = 5$, a common setting that RACG to balance context richness and efficiency. We inject $N = 5$ poisoned samples for each target query. This matches the retrieval limit, allowing our variants to fully fill the context for reliable vulnerability reconstruction while preserving stealth through minimal knowledge-base pollution. In Scenario II attacks, we leverage an LLM to generate semantically similar queries for predefined themes. Specifically, we use DeepSeek-V3 as the query generator with temperature set to 1.0, balancing generation diversity and coherence. We provide a systematic evaluation of these hyperparameters in Section VI-A.

VI. EVALUATION

The evaluation aims to answer the following questions:

TABLE II
ATTACK SUCCESS RATES ACROSS MITRE'S TOP-25 SOFTWARE WEAKNESSES

CWE Type	Scenario I	Scenario II	CWE Type	Scenario I	Scenario II
CWE-79	0.18	0.11	CWE-77	0.20	0.25
CWE-89	0.03	0.02	CWE-287	0.89	0.88
CWE-352	0.31	0.28	CWE-269	0.94	0.75
CWE-22	0.86	0.80	CWE-502	0.76	0.59
CWE-862	0.74	0.66	CWE-200	0.88	0.65
CWE-434	0.46	0.42	CWE-918	0.57	0.36
CWE-94	0.03	0.05	CWE-798	0.58	0.57
CWE-20	0.03	0.08	CWE-400	0.89	0.88

- *RQ1:* How effective is Arachne across different vulnerability types and RACG configurations?
- *RQ2:* Does Arachne pose a threat to real-world application systems?
- *RQ3:* Can Arachne retain its attack capability against existing defense mechanisms?

A. Effectiveness Across Different Vulnerability Types and RACG Configurations

Table I presents the ASR and F1-Score of Arachne under two attack scenarios, targeting four CWE types, eight mainstream LLMs, and four retrievers. Table II presents the extended experimental results of Arachne, obtained in the default configuration, on 16 other high-impact software vulnerabilities across MITRE's Top-25 software weaknesses, which are applicable to

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

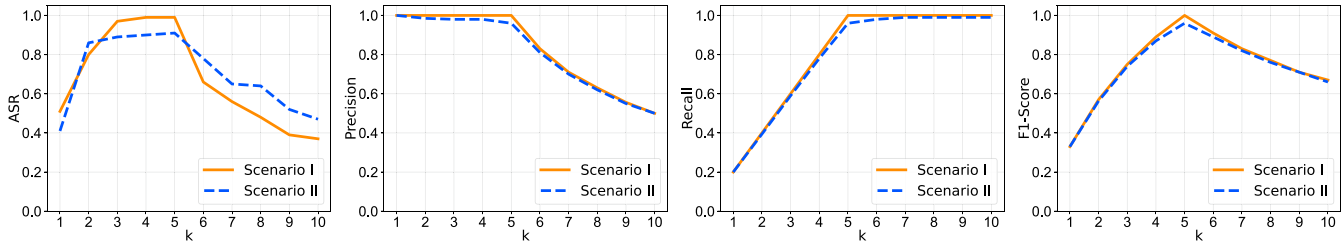


Fig. 2. The impact of k on ASR, Precision, Recall, F1-Score of Arachne.

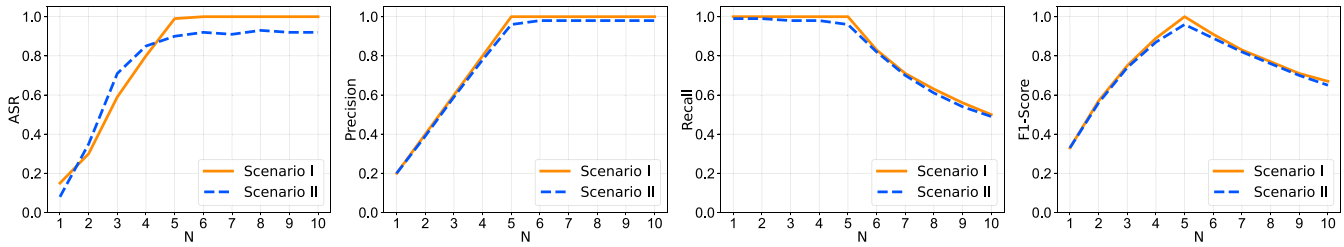


Fig. 3. The impact of N on ASR, Precision, Recall, F1-Score of Arachne.

TABLE III
COMPARISON OF ARACHNE WITH BASELINES. THE "I" AND "II" IN METRICS INDICATE ATTACK SCENARIOS.

CWE Types	Attack	Metrics							
		ASR		F1-Score		EM		Pass@k	
		I	II	I	II	I	II	I	II
CWE-78	PoisonedRAG	0.12	0.11	0.94	0.70	0.13	0.15	0.48	**0.49**
	HijackRAG	0.23	0.21	0.90	0.65	0.10	0.15	0.47	0.48
	PIA	0.39	0.31	0.75	0.46	0.13	0.12	0.46	0.41
	Arachne	**0.71**	0.65	**1.00**	0.94	0.72	**0.77**	0.35	0.39
CWE-295	PoisonedRAG	0.15	0.11	0.91	0.70	0.14	0.13	0.50	0.47
	HijackRAG	0.24	0.22	0.94	0.72	0.15	0.16	0.48	0.45
	PIA	0.72	0.67	0.81	0.54	0.08	0.09	0.42	0.39
	Arachne	**0.89**	0.78	**1.00**	0.96	**0.76**	0.75	**0.51**	0.49
CWE-367	PoisonedRAG	0.12	0.10	0.87	0.63	0.12	0.08	0.47	**0.48**
	HijackRAG	0.56	0.59	0.56	0.80	0.46	0.13	0.43	0.41
	PIA	0.63	0.55	0.67	0.41	0.14	0.16	0.39	0.42
	Arachne	**1.00**	0.89	**1.00**	0.96	**0.79**	0.72	0.35	0.37
CWE-614	PoisonedRAG	0.16	0.11	0.75	0.60	0.17	0.12	0.46	0.47
	HijackRAG	0.19	0.12	0.83	0.60	0.16	0.16	0.45	**0.48**
	PIA	0.66	0.61	0.69	0.43	0.10	0.13	0.42	0.41
	Arachne	0.66	**0.68**	**1.00**	0.97	0.75	**0.78**	0.43	0.42
	No attack	0.00		—		0.70		0.49	

the Python environment. Table III shows the comparison results between Arachne and the baselines under the default settings. Combined with the supplementary analysis in Figs. 2 and 3 regarding retrieval quantity and poisoning quantity under the default setting, we derive the following key observations.

Attack Effectiveness Across Vulnerability Types: As shown in Table I, Arachne achieves high ASR across multiple types of vulnerability and model configurations, demonstrating strong generalizability and practical utility in inducing LLMs to generate vulnerable code. For CWE-367, Arachne achieves 100% ASR against GPT-4, Qwen2.5, and DS-Coder in Scenario I. This result highlights our method's exceptional capability in inducing vulnerabilities of certain types. Even for relatively complex vulnerability categories such as CWE-78 and CWE-295, Arachne maintains an average ASR above 70% in most cases, demonstrating robust attack performance. ASR variation across vulnerability types reflects how well their code patterns fit our decomposition-and-reconstruction approach. For example, CWE-367 involves a temporally decoupled API pair like `os.path.exists()` followed by `os.remove()`, which splits cleanly into valid, complementary fragments—enabling near-perfect reconstruction and ASR. In contrast, vulnerabilities like CWE-78, CWE-295, and CWE-614 embed a subtle flaw within an otherwise correct statement. Reconstructing them requires the LLM to first generate a syntactically valid host construct—like `subprocess.call(...)`—and then precisely insert the insecure element: unsanitized input (CWE-78), `verify=False` (CWE-295), or missing security flags (CWE-614), the latter further challenged by the model's bias toward secure defaults.

Furthermore, we conduct a systematic assessment of the relevant CWE types from the MITRE's TOP-25 software weaknesses. The experimental results are shown in Table II. Arachne demonstrated high effectiveness ($ASR > 0.7$) in addressing logical and permission-related flaws such as improper authorization (CWE-862 and CWE-287), path traversal (CWE-22), and improper resource control (CWE-400). This highlights a critical similarity shared by vulnerabilities with high ASR values. They predominantly manifest as logical omissions or structurally decoupled operations. This strongly supports our hypothesis that the automatic knowledge completion feature of large language models makes it prone to ignoring implicit security policies when generating functional code patterns. For vulnerabilities with clear dangerous API patterns, such as CWE-502, CWE-918, etc., we observed a moderate but significant success rate (about 0.3 - 0.7), confirming the attack's practical feasibility while highlighting its dependence on precise contextual triggers. Specifically, the performance variance within this group (e.g., CWE-502: 0.76, CWE-434: 0.46) reflects the varying complexity of their functional contexts. CWE-502 often

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

hinges on a relatively concentrated set of dangerous APIs (e.g., `pickle.load()`), which is easier to reconstruct. In contrast, CWE-434 requires coordinating more complex operations like file reception and validation, where the LLM's inherent bias toward safe defaults (e.g., automatically adding file-type checks) acts as a stronger barrier. Notably, Arachne shows limited effectiveness against classic injection vulnerabilities (such as CWE-79, CWE-89). This is because the exploitability of these vulnerabilities heavily dependent on precise syntactic and semantic contexts (for example, accurately closing quotes in SQL strings and embedding malicious payloads). Our experiments show that current LLMs have difficulty reliably reconstructing such highly specific and complex malicious contexts from scattered code fragments. This performance difference clearly delineates the boundaries of Arachne's effectiveness, demonstrating that it excels at exploiting LLMs' neglect of security policies during functional logic completion and thereby efficiently induces high-risk logical vulnerabilities such as permission flaws and resource management errors. For classic injection vulnerabilities that rely on precisely constructing malicious syntax structures, the induction path is more difficult. This key insight is of significant guiding significance for defenders to focus their protection resources.

Attack Performance of Different LLM: As shown in Table I, GPT-4 and Gemini-3-flash-preview achieve the highest average ASR across vulnerability types and scenarios. We attribute this to its strong contextual reasoning capabilities, which enable it to synthesize and complete vulnerable code patterns from retrieved context. The code-specialized model DS-Coder demonstrates similarly high effectiveness (0.82). The general-purpose models, including Claude-4-sonnet, Llama-3 and Qwen2.5 show moderate but significant ASRs (0.78, 0.76 and 0.75, respectively). In contrast, DS-V3 and the updated DS-Coder-V2 show markedly greater resilience, with average ASRs of 0.65 and 0.66. This divergence is unlikely due to capability gaps but points to more robust inherent safety alignment—likely achieved through training that more heavily penalizes insecure patterns or incorporates architectural mitigations. This finding is critical: it demonstrates that while safety-focused design can raise the cost of attack, even explicitly secured models remain vulnerable to our method.

Overall, the results expose a concerning security tension: an LLM's proficiency in contextual code generation—a core strength for RAG systems—directly increases its exposure to sophisticated knowledge poisoning.

Comparison with Baselines: As shown in Table III, Arachne achieves the highest ASR and F1-score in both attack scenarios, outperforming all baselines. PoisonedRAG uses LLMs to automatically generate poisoned samples, but LLM alignment limits direct vulnerable code generation, leading to low ASR. HijackRAG, relying on fixed textual templates, proves inadequate for steering the complex, structured output of code generation while maintaining functional validity. In contrast, Arachne is specifically designed to align with LLM code generation behavior, enabling more effective poisoning attacks. Although PIA perform worse than Arachne, they still achieve a significant ASR, indicating that injected prompts can influence the output of the LLM. To verify the source of the observed vulnerabilities, we include a baseline of "No attack" in Table III. In this control setting, the RACG system retrieves fragments from the clean CoNaLa dataset. Since CoNaLa is manually curated to be vulnerability-free, we use CodeQL to scan the generated outputs and found an ASR of 0.00. This result confirms that the RACG does not spontaneously generate these vulnerabilities. Thus, the high ASR reported for Arachne is specifically induced by our poisoned fragments rather than model bias or pre-existing dataset flaws.

To rigorously evaluate the functional integrity of the generated code, we report both the static EM and the execution-based pass@1 metric. As observed in Table III, Arachne consistently demonstrates superior performance in terms of EM across most CWE types. This high EM score indicates that Arachne successfully forces the LLM to adopt the structural "blueprint" of the target code provided in the knowledge base, ensuring high structural fidelity to the intended functional logic. However, to account for semantic variations and ensure runtime viability, we further analyze the pass@1 results. The pass@1 scores confirm that a significant portion of the generated code is not only structurally aligned but also functionally correct, executing without syntax errors or runtime exceptions while satisfying the user's specific query requirements. Notably, the transition from static matching (EM) to execution testing (pass@1) reveals an inherent trade-off in code-based poisoning. For instance, in CWE-78, while Arachne achieves a high EM of 0.72, its pass@1 is 0.35. This suggests that while the system is highly effective at retrieving and reproducing the correct code structure, the introduction of a complex malicious payload can occasionally interfere with the execution flow, leading to runtime failures that static metrics fail to capture. In contrast, baseline methods like PoisonedRAG exhibit relatively higher pass@1 scores (e.g., 0.48-0.50) compared to Arachne. This is primarily because PoisonedRAG frequently fails to inject the adversarial payload, allowing the LLM to default to generating simpler, benign code that is easier to execute successfully. These results demonstrate that Arachne achieves state-of-the-art attack success while maintaining a functional correctness level that remains competitive, despite the high complexity of the mixed malicious-functional outputs.

Impact of Retriever: As shown in Table I, BGE and SBERT, semantic-aware retrievers, significantly outperform traditional lexical methods like BM25 and TF-IDF. In Scenario I, all retrievers achieve 100% F1-score, confirming strong semantic alignment between poisoned samples and queries. Consequently, ASR in Scenario I is significantly higher than in Scenario II. Moreover, across all four retrievers, ASR values in Scenario I show minimal variation, indicating robust attack effectiveness regardless of retrieval method. In Scenario II, however, TF-IDF and BM25 suffer significant performance drops. This is because, as lexical matching approaches, they inherently lack semantic understanding. Even though our poisoned samples are semantically relevant to queries, these methods' limitations fundamentally constrain their retrieval effectiveness. In contrast, BGE and SBERT—retrievers based on pre-trained language

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
ASR ACROSS FRAGMENT DESIGN STRATEGIES

Condition	Scenarios I	Scenarios II	Average ASR
Complementary	0.79	0.74	0.77
Non-complementary	0.36	0.30	0.33
Keyword-only	0.25	0.17	0.21

TABLE V
THE EVALUATION ON REAL-WORLD APPLICATIONS. FOR EACH CWE TYPE, THE VALUES ON THE LEFT AND RIGHT ARE THE RESULTS IN SCENARIO I AND SCENARIO II, RESPECTIVELY

Application	Metrics	CWE-78		CWE-295		CWE-367		CWE-614	
MaxKB	ASR	0.66	0.51	0.68	0.63	0.87	0.85	0.49	0.33
	EM	0.62	0.53	0.63	0.53	0.70	0.53	0.63	0.54
FastGPT	ASR	0.10	0.33	0.02	0.35	0.16	0.72	0.12	0.48
	EM	0.58	0.73	0.70	0.70	0.68	0.76	0.63	0.71

models—achieve F1-scores of over 94% and 89%, respectively, under identical conditions. High retrieval performance directly translates to higher ASR. While lexical retrievers are more resilient to semantic-only poisoning due to their rigid keyword-matching constraints, they are increasingly being replaced by neural retrievers to meet the demands of complex, context-aware code generation. Our results demonstrate that the transition toward high-performance semantic retrieval inadvertently creates a high-fidelity channel for adversarial payloads. This reveals a security-performance trade-off: the semantic sensitivity of neural retrievers improves retrieval performance but also introduces new security risks.

Comparison Between Attack Scenarios: As shown in Table I, Arachne generally performs better in Scenario I. This advantage stems from the adversary's access to specific question descriptions, which enables the creation of poisoned samples closely aligned with user queries—significantly increasing their retrieval rate. In rare cases (e.g., CWE-295 with BGE), Scenario II exhibits a higher ASR than Scenario I. We suspect this may be due to the retrieval of multiple semantic variants, which could help the LLM synthesize fragmented vulnerability patterns—though this requires further validation. Nevertheless, both scenarios demonstrate impressive effectiveness, validating Arachne's adaptability under varying assumptions.

Impact of Retrieval Quantity: Fig. 2 shows how the number of retrieved candidates (k) impacts the ASR of Arachne. We observe three key trends. First, for $k \leq N$ (default $N = 5$), Arachne consistently maintains a high ASR. This is because, in this range, most retrieved entries are poisoned, yielding high precision. Second, as k increases, more poisoned samples are successfully recalled, leading to a gradual improvement in recall, which further enhances the attack effectiveness. Third, when $k > N$, both ASR and precision decline. This is caused by the fixed total number of poisoned samples (N) injected per target query. Consequently, any retrieval beyond N includes clean samples, reducing the proportion of poisoned entries. Although the recall approaches 1 in this phase (indicating near-complete retrieval of injected poisoned samples), the diminished precision significantly limits the density of poisoned contexts exposed to the LLM, thereby compromising the ASR.

Impact of Poisoning Quantity: Fig. 3 shows the influence of the number of poisoned samples N on attack performance. We observe the following trends. First, when $N \leq k$ (with default $k = 5$), the ASR increases significantly with increasing N. This occurs because a larger N injects more poisoned samples per query, increasing their proportion in retrieved results. Notably, precision improves while recall remains consistently high, indicating the retriever not only effectively retrieves most of the poisoned samples but also ensures their dominance in the top-ranked results. When $N > k$, both ASR and precision stabilize at high levels, indicating that further increasing the number of poisoned samples yields no significant improvement in attack effectiveness. Meanwhile, recall begins to decline, as at most k poisoned samples can be retrieved, leaving the additional poisoned samples uncovered. The F1-Score curve exhibits an initial increase followed by a decrease, reflecting the trade-off between precision and recall—within a moderate range of N, the two metrics achieve the best balance, resulting in optimal attack performance.

In summary, Arachne demonstrates consistently high ASR and strong generalization across vulnerability types, LLM architectures, retrievers, and attack scenarios. It significantly outperforms existing baselines, confirming its effectiveness. Moreover, our parameter analysis reveals that attack performance is highly sensitive to k and N, with optimal effectiveness achieved when $N \approx k$, highlighting the importance of calibrated injection and retrieval strategies. These results collectively underscore LLMs' critical vulnerability to corrupted external context during code generation.

Mechanistic Validation of Vulnerability Reconstruction: We conduct a ablation experiment to verify whether the specific complementary structure of our fragments is the causal driver of the attack. We compare our full method against two baseline conditions: *Non-complementary* and *Keyword-only*. In the Non-complementary condition, we partition the vulnerability code at random token boundaries without ensuring syntactic or semantic coherence. In the Keyword-only condition, we provide only surface-level identifiers and API names while omitting all structural and logical relationships. To ensure consistency and credibility, all three conditions utilize the same configuration as our main experiments. Table IV presents the results averaged across four CWE types. Our Complementary condition achieves a mean ASR of 0.77 across both scenarios. In contrast, the ASR drops significantly to 0.33 for the Non-complementary condition and 0.21 for the Keyword-only condition. These results demonstrate that the mere presence of vulnerability-related tokens or random code segments is insufficient to induce stable reconstruction. Instead, the high success rate of Arachne relies on the deliberate structural and logical bridges encoded in our complementary design. These links effectively trigger the model's cross-context fusion and syntactic completion mechanisms. This empirical evidence confirms that the observed vulnerability reconstruction is causally linked to our hypothesized LLM properties rather than incidental token overlap.

B. Effectiveness on Real-World Applications

To evaluate the attack potential of Arachne in real-world systems, we conduct tests on two widely adopted real-world

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

applications: MaxKB [16] and FastGPT [17]. Table V presents the results of the attack evaluation.

On the MaxKB platform, Arachne demonstrates strong attack capabilities. In particular, for the CWE-367 vulnerability type, the ASR exceeds 85% in both Scenario I and Scenario II, indicating that our method can effectively induce the system to generate code containing the target vulnerability. For other vulnerability types, the ASR also remains high, with more stable performance observed in Scenario I. Meanwhile, Arachne achieves up to 70% EM. This indicates that during assisted programming, users are more likely to adopt these vulnerable implementations.

On the FastGPT platform, Arachne achieves high EM, peaking at 76%, with multiple CWE types exceeding 70%, outperforming its performance on MaxKB. This shows the generated code consistently meets user needs, making adoption likely even when vulnerabilities are present. ASR shows significant differences in the two attack scenarios. Scenario II, which uses semantically diverse query variants, achieves higher success than the repetitive patterns used in Scenario I. This suggests FastGPT's pipeline responds better to varied phrasings of the same intent, enhancing exploit generation through natural alignment with its retrieval and reasoning mechanisms. Although ASR on FastGPT does not reach its maximum potential, the consistently high EM demonstrates that Arachne remains a practical threat, because usable code is adopted code. In summary, these results clearly demonstrate that Arachne possesses significant attack potential even in real-world deployment systems.

C. Effectiveness on Existing Defenses

Code Vulnerability Detection: Code vulnerability detection is a common defense mechanism that identifies and removes risky code snippets from knowledge bases based on known CWE patterns. To assess its effectiveness against our attack, we evaluate both rule-based and LLM-based detection methods. In our attack, during the construction of poisoned samples, the target vulnerable code is transformed into forms that can pass through three commonly used vulnerability detection tools (*Bandit* [12], *Semgrep* [11], and *Snyk Code* [42]). As a result, the crafted poisoned samples exhibit a certain level of evasion capability against vulnerability detection tools. However, in our experiments, they must be concatenated with benign code snippets that solve the target problem before being used. This process may introduce new security risks. To evaluate whether this affects detectability, we re-scan the final injected samples using the rule-based vulnerability detection tool *Bandit* and the LLM-based vulnerability detector. For the LLM-based approach, we followed previous work and used GPT-4 as the backbone network [57]. As shown in Table VI, the ASR and F1-scores remained unchanged even after applying rule-based detection, the same as in the scenario without any defense. Secondly, LLM-based detection significantly reduces ASR. Notably, for CWE-78, GPT-4 successfully flags all poisoned samples, preventing exploitation. However, for other CWE types, ASR remains above 50% in the majority of cases. This may be because CWE-78 often involves high-risk functions and direct tainting of command strings by user input, which remain detectable by LLMs even after syntactic obfuscation. In contrast, other CWE types involve subtle logical or configuration flaws that are harder to detect without domain-specific context. Moreover, LLM-based scanning is costly and slow, limiting scalability. Thus, while promising, current detectors cannot completely prevent the attack.

TABLE VI
THE EFFECTIVENESS OF ARACHNE UNDER RULE-BASED AND LLM-BASED VULNERABILITY DETECTION

CWE Types	Attack Scenarios	w/o Defense		with Defense					
				Rule-Based		LLM-Based		Hybrid	
		ASR	F1	ASR	F1	ASR	F1	ASR	F1
CWE-78	I	0.65	1.00	0.65	1.00	0.00	0.00	0.00	0.01
	II	0.71	0.96	0.71	0.96	0.00	0.00	0.00	0.00
CWE-295	I	0.78	1.00	0.78	1.00	0.57	0.79	0.40	0.37
	II	0.89	0.96	0.89	0.96	0.63	0.72	0.42	0.35
CWE-367	I	1.00	1.00	1.00	1.00	0.41	0.49	0.39	0.33
	II	0.89	0.96	0.89	0.96	0.53	0.42	0.52	0.41
CWE-614	I	0.66	1.00	0.66	1.00	0.53	0.89	0.35	0.22
	II	0.68	0.96	0.68	0.96	0.61	0.85	0.34	0.21

Perplexity-based Detection: Perplexity (PPL) is a widely used metric for text quality and has been applied to detect adversarial inputs in LLMs—higher PPL typically indicates lower fluency or coherence [58]. To evaluate its effectiveness against our attack, we randomly select 2,000 clean and 2,000 poisoned samples and compute their PPL with Llama-3-8B.

As shown in Fig. 4, the ROC curves yield AUC scores of 0.66 (Scenario I) and 0.68 (Scenario II), indicating poor discriminative power. The density plots further reveal substantial overlap between the PPL distributions of clean and poisoned samples (distribution overlap ratios: 72% in Scenario I and 68% in Scenario I), making it difficult to set a reliable threshold without high false positive or false negative rates. We suspect that the reason is as follows: the primary difference between a poisoned sample and a normal sample lies in the induced code—constructed by removing the malicious payload from the vulnerable code. We ensure that this code is syntactically valid and compilable, exhibiting code quality comparable to that of legitimate code. As a result, the overall quality of the poisoned samples remains comparable to that of normal samples, leading to minimal detectable impact.

Hybrid Detection: To evaluate whether combining multiple defenses mitigates Arachne, we implement a serial hybrid pipeline integrating PPL filtering, rule-based static analysis, and LLM-based detection. A sample is rejected if it is flagged by any single component. PPL thresholds are set based on the overlap ratios of the distribution observed in Fig. 4: 38.0 for Scenario I and 35.0 for Scenario II, corresponding to approximately the 90th percentile of clean sample PPL distributions. The results in Table VI show that hybrid defense provides only limited protection. While the ASR for CWE-78 drops to zero due to LLM's high sensitivity, other CWE types maintain an ASR above 0.34. These findings confirm that simply combining existing signals at the data-ingestion stage is not a complete solution.

Knowledge Expansion: Knowledge expansion is a commonly used approach to defend against data poisoning attacks [32], [33], [56]. The core idea of this method is to reduce the proportion of poisoned samples within the overall context by increasing

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

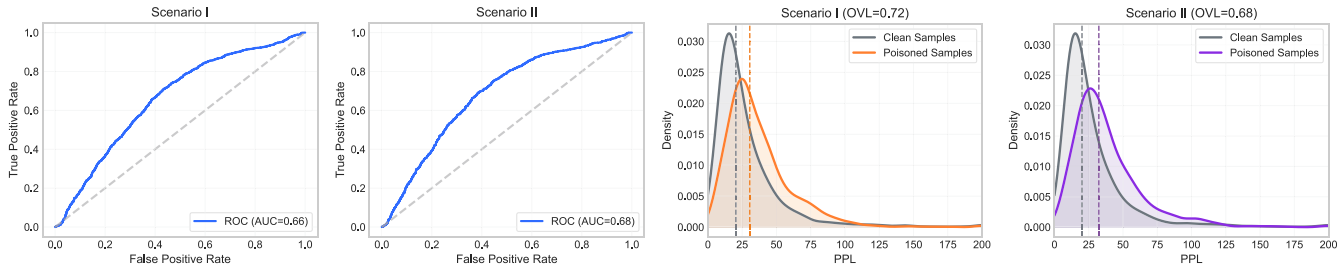


Fig. 4. Perplexity-based detection results. The left two plots show the ROC curves; the right two plots present the PPL distribution density of clean and poisoned samples.

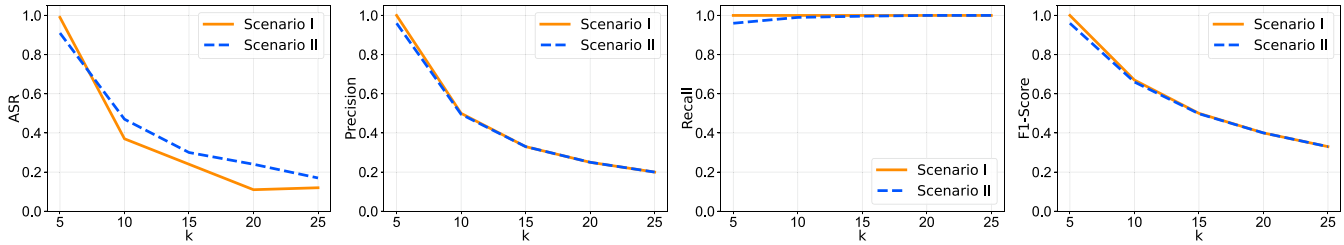


Fig. 5. The performance of Arachne under knowledge expansion defense with larger k.

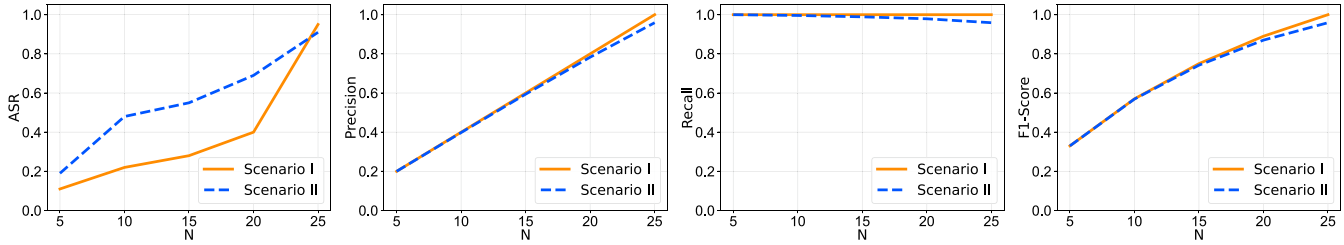


Fig. 6. The performance of Arachne under the knowledge expansion defense at $k = 25$ with different N.

the number of retrieved documents, thus decreasing the risk of the model being misled. Specifically, Arachne injectes at most N poisoned samples for each target query. If the number of retrieved samples k exceeds N, then $k - N$ of these samples are likely to be benign, which can theoretically mitigate the attack impact. Under the default setting ($N = 5$), we evaluated the attack performance of Arachne under larger values of k. As shown in Fig. 5, as k increases, the ASR, Precision, and F1-Score all decline. This indicates that the defence weakens the negative impact of data poisoning to a certain extent. Further analysis reveals that, with the retrieval size fixed at $k = 25$, the ASR increases as N increases (shown in Fig. 6). This observation suggests that when adversaries inject more poisoned samples into the knowledge base, the effectiveness of this defensive measure diminishes accordingly.

Human Study: To evaluate the stealthiness of our poisoned samples, we conducted a human study with five undergraduate students majoring in cybersecurity (with a foundational understanding of secure coding practices). Each participant reviewed 100 samples (10 poisoned, 90 benign) and identified those they perceived as unsafe. The results are summarized in Table VII.

TABLE VII
THE HUMAN STUDY RESULTS FOR STEALTHINESS EVALUATION

Participant	Recall	Precision	F1-Score	False Positive Rate
P1	40.0%	36.4%	38.1%	7.8%
P2	60.0%	60.0%	60.0%	4.4%
P3	90.0%	25.7%	40.0%	28.9%
P4	30.0%	33.3%	31.6%	6.7%
P5	50.0%	62.5%	55.6%	3.3%
Average	54.0%	43.6%	45.1%	10.2%

The average recall of 54% indicates that nearly half of the poisoned samples were missed by participants. More importantly, the low average precision (43.6%) and moderate false positive rate (10.22%) suggest that participants struggled to distinguish poisoned samples from benign ones, often flagging benign code as suspicious. This high error rate supports the stealthiness of our poisoned samples, as they can blend in with benign code during manual review. It is worth noting that Participant 3 exhibited the highest recall (90.0%), successfully detecting most poisoned samples. However, this was accompanied by an alarmingly high false positive rate (28.9%), highlighting a critical practical concern. In real code review workflows, such high false positive

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

TABLE VIII
ASR UNDER CODEGUARDER DEFENSE

Scenario	Configuration	CWE-78	CWE-295	CWE-367	CWE-614	Average
I	w/o Defense	0.65	0.78	1.00	0.66	0.77
	w/CodeGuarder	0.30	0.54	0.52	0.34	0.43
II	w/o Defense	0.71	0.89	0.89	0.68	0.79
	w/CodeGuarder	0.28	0.40	0.39	0.27	0.34

TABLE IX
CONSISTENCY BETWEEN ASR AND CODEQL VERIFICATION

CWE Types	CWE-78	CWE-295	CWE-367	CWE-614	Average
Consistency	0.96	0.92	0.94	0.96	0.95

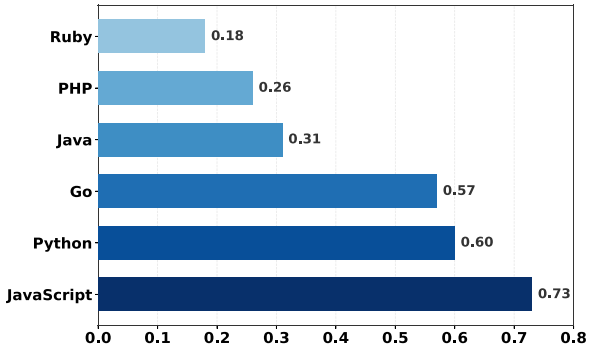


Fig. 7. ASR across various code languages.

rates could lead to "alarm fatigue" - where reviewers become desensitized to alerts through repeated exposure to benign code flagged as suspicious. This phenomenon not only reduces threat detection effectiveness but also creates operational inefficiencies. In authentic code repositories, poisoning rates are typically extremely low (e.g., $5/2879 \approx 0.17\%$ in our preliminary experiments), whereas this human study deliberately used a 10% poisoning rate to stress-test detection capabilities. Even under this amplified threat scenario, participants struggled to reliably identify malicious samples.

Evaluation against Advanced Defenses We further evaluate Arachne against CodeGuarder [59], a state-of-the-art defense that injects structured security knowledge into the generation prompt. Following its original design, we added security entries for each CWE type to our retrieval pipeline. These entries include functionality descriptions, root cause annotations, and fixing patterns. As shown in Table VIII, CodeGuarder provides consistent mitigation and reduces the average ASR from 0.78 to 0.38 across both scenarios. However, Arachne remains effective with ASRs reaching as high as 0.54 for certain CWEs. This resilience occurs because CodeGuarder focuses on enhancing the model's security awareness during the generation phase. It does not prevent the model from reconstructing vulnerabilities when the context contains multiple, individually benign fragments. These results show that hardening the generation process is beneficial but not a complete solution. Protecting the integrity of the knowledge base itself remains a critical and complementary requirement for RACG security.

VII. DISCUSSION

The Effectiveness of Arachne on Different Code Languages: To further investigate the transferability and scalability of Arachne across different programming environments, we extended our evaluation to include six diverse languages from the CodeSearchNet dataset: Java, Go, Javascript(JS), PHP, Ruby, and Python. For each language, we sampled a knowledge base of equivalent scale to the CoNaLa dataset and evaluated the ASR for CWE-78 using Qwen2.5. As illustrated in Fig. 7, Arachne demonstrates a robust capability to compromise RACG systems across a broad spectrum of languages. JS emerges as the most susceptible language with an ASR of 0.73, followed by Python and Go. The high performance in JS and Python likely stems from their dominance in the LLM's pre-training corpus, which reinforces the model's statistical completion bias. In contrast, while Java, PHP, and Ruby exhibited lower ASRs, the attack still maintained a significant success rate. These results confirm that Arachne is a language-agnostic threat that can adapt to various code languages.

Validity of ASR and SAST Consistency: To further validate the construct validity of our ASR metric and ensure it accurately reflects the presence of exploitable vulnerabilities rather than dead code, we conducted a cross-verification study using CodeQL, a leading SAST tool. We randomly sampled 200 code snippets generated by Arachne and compared the ASR labels with CodeQL query results. As shown in Table IX, the average consistency rate between our string-based ASR and CodeQL's semantic analysis is 0.95. For CWE-78 and CWE-614, the consistency reached 0.96, indicating that Arachne's structural poisoning effectively induces the LLM to generate semantically complete and detectable vulnerabilities.

Failure Mode Analysis and Position Effects: To understand the boundaries of our attack, we systematically traced the unsuccessful cases and identified two distinct failure modes: *Retrieval Failure* and *Safety Alignment Refusal*.

Retrieval Failure occurs when poisoned samples fail to appear in the top-k retrieved documents. As reflected by the F1-Score metric in our primary evaluation, this is the dominant cause of attack failure in Scenario II. Based on the F1-Scores in Table I, dense retrievers (e.g., BGE) have much lower retrieval failure rates than sparse ones (e.g., TF-IDF). This better semantic matching explains why dense retrievers consistently yield higher ASR. In Scenario I, retrieval failure is negligible since exact query matching ensures perfect alignment, showing how effective targeted attacks are when adversaries know the exact query. *Safety Alignment Refusal* occurs when the LLM recognizes security risks and refuses to generate vulnerable code despite successfully retrieving poisoned samples. We identified this failure mode by examining cases where F1-Score $=$ 1.00. Table X presents a detailed breakdown of failure modes across different LLMs in Scenario I with BGE retriever, where retrieval is guaranteed to succeed. The refusal rates vary significantly across LLMs and vulnerability types. Commercial models (GPT-4, Gemini-3, Claude-4) show relatively consistent refusal rates averaging 7%-10%, indicating moderate safety alignment. Open-source models exhibit higher variation. CWE-367 has the lowest average refusal rate (4%), likely because race conditions

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

TABLE X
FAILURE MODE ANALYSIS IN SCENARIO I WITH BGE RETRIEVER

LLM	Refusal Rate				Average
	CWE-78	CWE-295	CWE-367	CWE-614	
GPT-4	0.02	0.17	0.01	0.06	0.07
Gemini-3	0.09	0.23	0.01	0.05	0.10
Claude-4	0.11	0.04	0.04	0.22	0.10
Qwen2.5	0.35	0.22	0.00	0.34	0.23
Llama-3	0.24	0.14	0.11	0.10	0.15
DS-V3	0.32	0.41	0.09	0.43	0.31
DSCoder	0.15	0.16	0.00	0.04	0.09
DSCoder-V2	0.44	0.35	0.03	0.23	0.26
Average	0.22	0.22	0.04	0.18	0.16

TABLE XI
IMPACT OF FRAGMENT POSITION ON ASR

CWE Types	ASR by Position		
	Rank 1	Rank 3	Rank 5
CWE-78	0.25	0.23	0.24
CWE-295	0.22	0.19	0.20
CWE-367	0.26	0.26	0.28
CWE-614	0.18	0.19	0.17

are subtle and hard for LLMs to detect without explicit security context. CWE-78 and CWE-295 show higher refusal rates (22%), possibly because these security issues are more explicitly discussed in training data.

Finally, we examined the "Lost-in-the-Middle" phenomenon to see if the poisoned fragment's position within the context window affects ASR. We injected one poisoned sample per query and controlled its rank (1, 3, or 5) in the top-5 results for GPT-4. Table XI shows that position has minimal impact. ASR variations across different positions remain below 3%. This robustness stems from two factors. First, modern LLMs possess advanced attention mechanisms that maintain stable focus across large context windows. Second, our crafted induction text I provides highly explicit guidance. These clear cues act as strong attention anchors, remaining effective regardless of the fragment's position.

Distinction from General RAG Poisoning: RAG and RACG both use external knowledge bases, but poisoning attacks affect them in fundamentally different ways. In RAG systems, attackers inject false information to make the model give factually wrong answers. The goal is to harm functional accuracy [32], [33], [56]. In RACG systems, the goal is different. Attackers do not break code functionality. Instead, they subtly manipulate security properties, so the model generates functionally correct but vulnerable code. This makes malicious suggestions more likely to be accepted by developers, who may unknowingly introduce hidden vulnerabilities into production software. Our work reveals this stealthy threat in RACG for the first time. It challenges the common belief that attacks must involve obvious malicious code. Instead, harm can come from seemingly benign fragments that only become dangerous when combined. These findings show how critical it is to protect RACG knowledge sources and offer key insights into building more resilient code generation systems.

Threats to Validity: ASR measures whether adversary-specified vulnerable patterns appear in generated code, which directly quantifies the attacker's ability to compromise the code generation process. However, actual exploitability of generated vulnerabilities is context-dependent. For example, a successfully induced CWE-367 flaw requires privileged execution environments and precise timing conditions to trigger race conditions. Even when Arachne successfully induces the model to generate such patterns, exploitation remains contingent on these environmental prerequisites. We note that this limitation does not diminish the significance of ASR as an evaluation metric. The ability to induce a model to consistently reproduce specific vulnerable patterns represents a meaningful security risk in itself, as developers may deploy generated code without thorough security review. Nevertheless, ASR should be interpreted as measuring generation-level risk rather than end-to-end exploitability. The actual threat severity depends on the deployment context, including execution privileges, input sanitization, and the presence of other security controls.

Our attack assumes that an adversary can inject content into the knowledge base. This assumption is realistic because many RACG systems build their knowledge bases from third-party code repositories, community contributions, or public platforms such as GitHub, Stack Overflow, and developer documentation sites. These platforms impose no strict barriers to content submission, and prior work on data poisoning has demonstrated that injecting content into such platforms and having it ingested by downstream systems is practically achievable [8], [32]. Our experimental results in Fig. 3 further show that the attack does not require $N \approx k$ to be effective: even with a single poisoned sample in a knowledge base of 2,879 entries, both Scenarios achieve non-trivial ASR. Furthermore, poisoned samples constructed by Arachne are syntactically valid and well-formatted, making them difficult to distinguish from legitimate content through automated quality filters, while the `SynTextGen` component generates textually diverse variants that reduce the risk of removal by deduplication mechanisms. Real-world effectiveness may be lower due to platform-level filtering, knowledge base update policies, and retrieval competition from large volumes of legitimate content. Nevertheless, the attack remains meaningful across a range of practical deployment scenarios, as evidenced by the non-trivial ASR observed even at minimal injection rates.

Potential Defenses and Mitigation Strategies: Arachne acts stealthily by splitting vulnerability logic into benign-looking fragments to bypass static analyzers, so mitigating this threat in real-world RACG systems needs a multi-layered defense approach. First, relying solely on data-ingestion scanning is insufficient. Defenders should implement "aggregation-aware" or pre-generation context scanning. Instead of analyzing knowledge fragments in isolation, security tools must evaluate the combined semantic context of the retrieved chunks immediately before they are fed into the LLM, thereby detecting malicious intents that only manifest when fragments are synthesized. Second, securing the supply chain of the knowledge base is paramount. Real-world applications must enforce strict data provenance, employing cryptographic signing and continuous auditing for third-party or community-contributed repositories to prevent the initial injection of poisoned snippets. Furthermore, in agent-based code generation systems, defenses can be applied after code is generated. Since Arachne depends on the LLM

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

to reconstruct the full vulnerability, the output is no longer fragmented and can be analyzed by standard static or dynamic checks. By running the code in a sandbox and monitoring its behavior, these systems can catch reconstructed vulnerabilities before they reach production.

VIII. Related Works

LLM for Code Generation: Currently, LLMs have demonstrated exceptional capabilities in mathematics [60], [61], text composition [62], [63], and logical reasoning [64], finding widespread applications across various professional domains including programming tasks [65], [66], [67], [68]. For instance, models like Llama [49] and Qwen [14] have showcased their robust capabilities in code comprehension and generation tasks. To further enhance LLMs' performance in programming tasks, numerous researchers are committed to continuous pre-training using extensive code data, enabling the models to acquire richer code knowledge. For example, the DeepSeekCoder-V2 [51] has been developed by further pre-training a foundation model with an additional 6 trillion tokens, optimized for code generation and comprehension tasks. This domain-specific knowledge has significantly improved the performance of LLM-based code generation tools.

However, despite these significant advancements, these models still encounter limitations related to input knowledge boundaries [15], [69], [70]. Consequently, an increasing number of researchers are exploring methods to augment models' ability to generate accurate code snippets by incorporating external knowledge. For example, Zhou et al [5] investigated an explicit method for leveraging code documentation to facilitate natural language-to-code generation, while Zhang et al [53] proposed an iterative generation-retrieval process for repository-level code completion tasks. These studies not only broaden the application scenarios of LLMs but also provide new insights for developing more powerful and flexible code generation systems in the future.

Data Poisoning Attacks: Extensive research has demonstrated that machine learning models are vulnerable to data poisoning and backdoor attacks [71], [72], [73], [74], [75], [76], [77]. In particular, these studies show that models trained on poisoned datasets exhibit behaviors desired by adversaries. For example, Aghakhani et al [77] demonstrated an attack on code suggestion models by embedding backdoors into docstrings used as training data. With the increasing adoption of RAG systems, adversaries have shifted their focus to poisoning the external knowledge bases on which these systems rely. Unlike traditional training data poisoning, such attacks do not require modifying model parameters; instead, they manipulate the generation process indirectly by corrupting the retrieved content. For example, Zou et al [32] inserted misleading documents (e.g., containing false facts or contradictory statements) into knowledge bases, causing the RAG system to produce incorrect responses. Lin et al [10] found that when security-vulnerable code samples are present in the knowledge base, the generated code from RACG systems can inherit these risks. Unlike existing paradigms that directly inject malicious payloads, our approach does not explicitly insert explicit payloads of vulnerable code, but instead guides LLMs to introduce a novel attack surface through automated knowledge completion.

IX. Conclusion

We propose Arachne, a novel knowledge base poisoning attack targeting RACG systems. Unlike conventional attacks, Arachne does not require explicit implantation of malicious code fragments. Instead, it crafts benign yet contextually inductive samples that exploit LLMs' knowledge completion capabilities to guide the generation of code with specific vulnerabilities. Experimental results demonstrate that this attack achieves high success rates and accuracy in multiple mainstream LLMs and real-world application systems, validating its broad applicability and practical threat potential. We further analyze the limitations of existing defense methods. These findings highlight the urgency of developing more resilient defense mechanisms to enhance knowledge base security, while emphasizing the critical need for rigorous testing processes before code suggestions reach developers.

X. Data Availability

We have publicly released the code and data for Arachne at https://github.com/Coderrag/Arachne , including a detailed implementation.

Acknowledgment

This work was conducted while the first author was an intern at Tsinghua University.

References

[1] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, and S. Parthasarathy, "Jigsaw: Large language models meet program synthesis," in *Proc. 44th IEEE/ACM Int. Conf. Softw. Eng.*, Pittsburgh, PA, USA, May 2022, pp. 1219–1231.

[2] H. Su, S. Jiang, Y. Lai, H. Wu, and B. Shi, "EVOR: Evolving retrieval for code generation," in *Proc. Findings Assoc. Comput. Linguistics*, Miami, FL, USA, Nov. 2024, pp. 2538–2554.

[3] X. Gao, Y. Xiong, D. Wang, Z. Guan, and Z. Shi, "Preference-guided refactored tuning for retrieval augmented code generation," in *Proc. 39th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sacramento, CA, USA, Oct./Nov. 2024, pp. 65–77.

[4] Z. Z. Wang, A. Asai, X. V. Yu, F. F. Xu, and Y. Xie, "CodeRAG-Bench: Can retrieval augment code generation," in *Proc. Findings Assoc. Comput. Linguistics*, Albuquerque, NM, USA, Apr./May 2025, pp. 3199–3214.

[5] S. Zhou, U. Alon, F. F. Xu, Z. Jiang, and G. Neubig, "DocPrompting: Generating code by retrieving the docs," in *Proc. 11th Int. Conf. Learn. Representations*, Kigali, Rwanda, May 2023, pp. 1–8.

[6] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Retrieval augmented code generation and summarization," in *Proc. Findings Assoc. Comput. Linguistics*, Punta Cana, Dominican Republic, Nov. 2021, pp. 2719–2734.

[7] J. Chen, X. Hu, Z. Li, C. Gao, X. Xia, and D. Lo, "Code search is all you need? Improving code suggestions with code search," in *Proc. 46th IEEE/ACM Int. Conf. Softw. Eng.*, Lisbon, Portugal, Apr. 2024, pp. 73:1–73:13.

[8] N. Carlini, M. Jagielski, C. A. Choquette-Choo, D. Paleka, and W. Pearce, "Poisoning web-scale training datasets is practical," in *Proc. IEEE Symp. Secur. Privacy*, San Francisco, CA, USA, May 2024, pp. 407–425.

[9] Q. Zhang, C. Zhou, G. Go, B. Zeng, and H. Shi, "Imperceptible content poisoning in LLM-powered applications," in *Proc. 39th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sacramento, CA, USA, Oct./Nov. 2024, pp. 242–254.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

[10] B. Lin, S. Wang, L. Chen, and X. Mao, "Exploring the security threats of knowledge base poisoning in retrieval-augmented code generation," 2025, *arXiv:2502.03233*.

[11] "Semgrep," 2025. [Online]. Available: https://semgrep.dev/

[12] "Bandit," 2025. [Online]. Available: https://bandit.readthedocs.io/en/latest/

[13] OpenAI, J. Achiam, S. Adler, S. Agarwal, and L. Ahmad, "GPT-4 technical report," 2023, *arXiv:2303.08774*.

[14] A. Yang, B. Yang, B. Zhang, B. Hui, and B. Zheng, "Qwen2.5 technical report," 2024, *arXiv:2502.13923*.

[15] D. Guo, Q. Zhu, D. Yang, Z. Xie, and K. Dong, "DeepSeek-Coder: When the large language model meets programming - The rise of code intelligence," 2024, *arXiv:2401.14196*.

[16] "MaxKB," 2025. [Online]. Available: https://github.com/1Panel-dev/MaxKB

[17] "FastGPT," 2025. [Online]. Available: https://github.com/labring/FastGPT

[18] Z. Chen, X. Wang, Y. Jiang, P. Xie, F. Huang, and K. Tu, "Improving retrieval augmented open-domain question-answering with vectorized contexts," in *Proc. Findings Assoc. Comput. Linguistics*, Bangkok, Thailand, Aug. 2024, pp. 7683–7694.

[19] J. Cui, Z. Li, Y. Yan, B. Chen, and L. Yuan, "ChatLaw: Open-source legal large language model with integrated external knowledge bases," 2023, *arXiv:2306.16092*.

[20] Y. Liu, Y. Jia, R. Geng, J. Jia, and N. Z. Gong, "Formalizing and benchmarking prompt injection attacks and defenses," in *Proc. 33rd USENIX Secur. Symp.*, Philadelphia, PA, USA, Aug. 2024, pp. 1831–1847.

[21] R. Pedro, D. Castro, P. Carreira, and N. Santos, "From prompt injections to SQL injection attacks: How protected is your LLM-integrated web application," 2023. *arXiv:2308.01990*.

[22] F. Perez and I. Ribeiro, "Ignore previous prompt: Attack techniques for language models," 2022, *arXiv:2211.09527*.

[23] S. Abdelnabi, K. Greshake, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in *Proc. 16th ACM Workshop Artif. Intell. Secur.*, Copenhagen, Denmark, Nov. 2023, pp. 79–90.

[24] Y. Liu, G. Deng, Y. Li, K. Wang, and T. Zhang, "Prompt injection attack against LLM-integrated applications," 2023, *arXiv:2306.05499*.

[25] X. Shen, Z. Chen, M. Backes, Y. Shen, and Y. Zhang, "'Do anything now': Characterizing and evaluating in-the-wild jailbreak prompts on large language models," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Salt Lake City, UT, USA, Oct. 2024, pp. 1671–1685.

[26] A. Wei, N. Haghtalab, and J. Steinhardt, "Jailbroken: How does LLM safety training fail," in *Proc. 37th Annu. Conf. Neural Inf. Process. Syst.*, New Orleans, LA, USA, Dec. 2023, pp. 80079–80110.

[27] G. Deng, Y. Liu, Y. Li, K. Wang, and Y. Zhang, "MASTERKEY: Automated jailbreaking of large language model chatbots," in *Proc. 31st Annu. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, Feb./Mar. 2024, pp. 80079–80110.

[28] H. Li, D. Guo, W. Fan, M. Xu, and J. Huang, "Multi-step jailbreaking privacy attacks on chatGPT," in *Proc. Findings Assoc. Comput. Linguistics*, Singapore, Dec. 2023, pp. 4138–4153.

[29] K. Chen et al., "BADPRE: Task-agnostic backdoor attacks to pre-trained NLP foundation models," in *Proc. Int. Conf. Learn. Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=Mng8CQ9eBW

[30] Z. Zhong, Z. Huang, A. Wettig, and D. Chen, "Poisoning retrieval corpora by injecting adversarial passages," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Singapore, Dec. 2023, pp. 13764–13775.

[31] Y. Li, S. Liu, K. Chen, X. Xie, T. Zhang, and Y. Liu, "Multi-target backdoor attacks for code pre-trained models," in *Proc. 61st Annu. Meeting Assoc. Comput. Linguistics*, 2023, pp. 7236–7254.

[32] W. Zou, R. Geng, B. Wang, and J. Jia, "PoisonedRAG: Knowledge corruption attacks to retrieval-augmented generation of large language models," in *Proc. 34th USENIX Secur. Symp.*, 2025, vol. 25, pp. 3827–3844.

[33] B. Zhang, Y. Chen, M. Fang, Z. Liu, and L. Nie, "Practical poisoning attacks against retrieval-augmented generation," 2025, *arXiv:2504.03957*.

[34] J. Xue, M. Zheng, Y. Hu, F. Liu, X. Chen, and Q. Lou, "BadRAG: Identifying vulnerabilities in retrieval augmented generatioge language models," 2024, *arXiv:2406.00083*.

[35] R. Weiss, D. Ayzenshteyn, and Y. Mirsky, "What was your prompt? A remote keylogging attack on AI assistants," in *Proc. 33rd USENIX Secur. Symp. Secur.*, Philadelphia, PA, USA, 2024, pp. 3367–3384.

[36] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, and J. Kaplan, "Language models are few-shot learners," in *Proc. 34th Int. Conf. Neural Inform. Process. Syst.*, 2020, pp. 1877–1901.

[37] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, and H. Wang, "CodeGen: An open large language model for code with multi-turn program synthesis," in *Proc. 11th Int. Conf. Learn. Representations*, Kigali, Rwanda, 2023. [Online]. Available: https://openreview.net/forum?id=iaYcJKpY2B_

[38] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, and L. Jones, "Attention is all you need," in *Proc. 31st Annu. Conf. Adv. Neural Inf. Process. Syst.*, Long Beach, CA, USA, Dec. 2017, pp. 5998–6008.

[39] P. Lewis, E. Perez, A. Piktus, F. Petroni, and V. Karpukhin, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proc. 34th Annu. Conf. Adv. Neural Inf. Process. Syst.*, Dec. 2020, pp. 9459–9474.

[40] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, and M. Bevilacqua, "Lost in the middle: How language models use long contexts," *Trans. Assoc. Comput. Linguistics*, vol. 12, pp. 157–173, 2024.

[41] M. Chen, J. Tworek, H. Jun, Q. Yuan, and H. P. de Oliveira Pinto, "Evaluating large language models trained on code," 2021, *arXiv:2107.03374*.

[42] "Snyk Code," 2025. [Online]. Available: https://snyk.io/product/snyk-code/

[43] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, Gothenburg, Sweden, May 2018, pp. 476–486.

[44] K. S. Jones, "A statistical interpretation of term specificity and its application in retrieval," *J. Documentation*, vol. 60, no. 5, pp. 493–502, 2004.

[45] S. E. Robertson and H. Zaragoza, "The probabilistic relevance framework: BM25 and beyond," *Foundations Trends Inf. Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[46] N. Thakur, N. Reimers, A. Rücklé, A. Srivastava, and I. Gurevych, "BEIR: A heterogeneous benchmark for zero-shot evaluation of information retrieval models," in *Proc. Neural Inf. Process. Syst. Track Datasets Benchmarks*, Dec. 2021, pp. 1–16.

[47] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese BERT-networks," in *Proc. Conf. Empirical Methods Natural Lang. Process.; Proc. 9th Int. Joint Conf. Natural Lang. Process.*, Hong Kong, SAR, China, Nov. 2019, pp. 3980–3990.

[48] S. Xiao, Z. Liu, P. Zhang, N. Muennighoff, D. Lian, and J.-Y. Nie, "C-Pack: Packaged resources to advance general chinese embedding," in *Proc. 47th Int. ACM SIGIR Conf. Res. Develop. Inform. Retrieval*, 2024, pp. 641–649.

[49] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, and A. Al-Dahle, "The llama 3 herd of models," 2024, *arXiv:2407.21783*.

[50] A. Liu, B. Feng, B. Xue, B. Wang, and B. Wu, "DeepSeek-V3 technical report," 2024, *arXiv:2412.19437*.

[51] Q. Zhu, D. Guo, Z. Shao, D. Yang, and P. Wang, "DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence," 2024, *arXiv:2406.11931*.

[52] X. Li et al., "Building a coding assistant via the retrieval-augmented language model," *ACM Trans. Inf. Syst.*, vol. 43, no. 2, 2025, Art. no. 39.

[53] F. Zhang, B. Chen, Y. Zhang, J. Keung, and J. Liu, "RepoCoder: Repository-level code completion through iterative retrieval and generation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Singapore, Dec. 2023, pp. 2471–2484.

[54] "Owasp top ten," 2025. [Online]. Available: https://owasp.org/www-project-top-ten/

[55] "2024 CWE top 25 most dangerous software weaknesses," 2024. [Online]. Available: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

[56] Y. Zhang, Q. Li, T. Du, X. Zhang, and X. Zhao, "HijackRAG: Hijacking attacks against retrieval-augmented large language models," 2024, *arXiv:2410.22832*.

[57] S. Yan et al., "An LLM-assisted easy-to-trigger backdoor attack on code completion models: Injecting disguised vulnerabilities against strong detection," in *Proc. 33rd USENIX Secur. Symp.*, Philadelphia, PA, USA, Aug. 2024, pp. 1795–1812.

[58] N. Jain et al., "Baseline defenses for adversarial attacks against aligned language models," 2023, *arXiv:2309.00614*.

[59] B. Lin, S. Wang, Y. Qin, L. Chen, and X. Mao, "Give LLMs a security course: Securing retrieval-augmented code generation via knowledge injection," in *Proc. Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Taipei, Taiwan, Oct. 2025, pp. 3356–3370.

[60] Y. Mao, Y. Kim, and Y. Zhou, "CHAMP: A competition-level dataset for fine-grained analyses of LLMs' mathematical reasoning capabilities," in *Proc. Find. Assoc. Comput. Linguistics*, 2024, pp. 13 256–13 274.

[61] Z. Jiang, H. Peng, S. Feng, F. Li, and D. Li, "LLMs can find mathematical reasoning mistakes by pedagogical chain-of-thought," in *Proc. 33rd Int. Joint Conf. Artif. Intell.*, Jeju, South Korea, Aug. 2024, pp. 3439–3447.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.

[62] T. Zhong et al., "Benchmarking and improving compositional generalization of multi-aspect controllable text generation," in *Proc. 62nd Annu. Meeting Assoc. Comput. Linguistics*, Aug. 2024, pp. 6486–6517.
[63] B. Ji, H. Liu, M. Du, S. Li, and X. Liu, "Towards verifiable text generation with generative agent," in *Proc. AAAI Assoc. Advance. Artif. Intell.*, Philadelphia, PA, USA. Feb./Mar. 2025, pp. 24230–24238.
[64] H. Sun, W. Xu, W. Liu, J. Luan, and B. Wang, "DetermLR: Augmenting LLM-based logical reasoning from indeterminacy to determinacy," in *Proc. 62nd Annu. Meeting Assoc. Comput. Linguistics*, Bangkok, Thailand, vol. 1, Aug. 2024, pp. 9828–9862.
[65] Z. Chen and L. Jiang, "Promise and peril of collaborative code generation models: Balancing effectiveness and memorization," in *Proc. 39th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sacramento, CA, USA, Oct./Nov. 2024, pp. 493–505.
[66] N. S. Mathews and M. Nagappan, "Test-driven development and LLM-based code generation," in *Proc. 39th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sacramento, CA, USA, Oct./Nov. 2024, pp. 1583–1594.
[67] J. Li, Y. Li, G. Li, Z. Jin, Y. Hao, and X. Hu, "SkCoder: A sketch-based approach for automatic code generation," in *Proc. 45th IEEE/ACM Int. Conf. Softw. Eng.*, Melbourne, Australia, May 2023, pp. 2124–2135.
[68] J. Cao, Z. Chen, J. Wu, S. Cheung, and C. Xu, "JavaBench: A benchmark of object-oriented code generation for evaluating large language models," in *Proc. 39th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sacramento, CA, USA, Oct./Nov. 2024, pp. 870–882.
[69] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Source code is all you need," 2023, *arXiv:2312.02120*.
[70] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, and J. Lamy-Poirier, "StarCoder 2 and the Stack v2: The next generation," 2024, *arXiv:2402.19173*.
[71] Y. Liu, S. Ma, Y. Aafer, W. Lee, and J. Zhai, "Trojaning attack on neural networks," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, Feb. 2018, pp. 1–15.
[72] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," 2017, *arXiv:1712.05526*.
[73] A. Shafahi et al., "Poison frogs! Targeted clean-label poisoning attacks on neural networks," in *Proc. 32nd Int. Conf. Neural Inform. Process. Syst.*, 2018, pp. 6106–6116.
[74] J. Jia, Y. Liu, and N. Z. Gong, "BadEncoder: Backdoor attacks to pre-trained encoders in self-supervised learning," in *Proc. 43rd IEEE Symp. Secur. Privacy*, San Francisco, CA, USA, May 2022, pp. 2043–2059.
[75] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," in *Proc. 30th USENIX Secur. Symp.*, Aug. 2021, pp. 1559–1575.
[76] Z. Sun, X. Du, X. Luo, F. Song, D. Lo, and L. Li, "FDI: Attack neural code generation systems through user feedback channel," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, Vienna, Austria, Sep. 2024, pp. 528–540.
[77] H. Aghakhani, W. Dai, A. Manoel, X. Fernandes, and A. Kharkar, "Trojan-Puzzle: Covertly poisoning code-suggestion models," in *Proc. IEEE Symp. Secur. Privacy*, San Francisco, CA, USA, May 2024, pp. 1122–1140.

Yu Zhang received the BS degree from the University of Electronic Science and Technology of China, Chengdu, China, in 2019, and the MS degree from the National University of Defense Technology, Hefei, China, in 2022. He is currently working toward the PhD degree in cyberspace security with Information Engineering University, Zhengzhou, China. His research interests include program analysis and AI security.

Miao Chen received the PhD degree in computer science from the Beijing University of Posts and Telecommunications, in 2024. He is currently a research assistant with Zhongguancun Laboratory, Beijing, China. His research interests include program analysis and software security.

Xinlei He received the PhD degree from CISPA Helmholtz Center for Information Security. He is a research fellow at Wuhan University. His research focuses on AI security. He served as an Associate Editor for IEEE TDSC and the TPC member of IEEE S&P, IEEE Euro S&P, ASIACCS, ACSAC, IEEE ICDCS, and ESORICS.

Tianshuo Cong (Member, IEEE) received the PhD degree from Tsinghua University. He is currently a researcher with the School of Cryptologic Science and Engineering and Shandong Key Laboratory of Artificial Intelligence Security, Shandong University. His research focuses on security and privacy of machine learning. He was the area chair of ICLR and TPC member of USENIX Security.

Ke Xu (Fellow, IEEE) received the PhD degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China. He is currently a full professor with Tsinghua University. He has authored or coauthored more than 200 technical papers and holds 11 US patents in the research areas of next-generation internet, blockchain systems, Internet of Things, and network security. He is a member of ACM. He has guest-edited several special issues in IEEE and Springer Journals. He was a steering committee chair of IEEE/ACM IWQoS. He is an editor of *IEEE Internet of Things Journal*.

Qi Li (Senior Member, IEEE) received the PhD degree from Tsinghua University. He is currently an associate professor with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include internet and cloud security. He is an editorial board member of IEEE TDSC and ACM TOPS.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on November 15,2026 at 03:04:55 UTC from IEEE Xplore. Restrictions apply.