DrainCode: Stealthy Energy Consumption Attacks on Retrieval-Augmented Code Generation via Context Poisoning

Yanlin Wang[1], Jiadong Wu[1], Tianyue Jiang[1], Mingwei Liu[1], Jiachi Chen[1], Chong Wang[3*], Ensheng Shi[2], Xilin Liu[2], Yuchi Ma[2], Zibin Zheng[1]
[1]Sun Yat-sen University, Zhuhai, China
{wangylin36, wujd28, jiangty9, liumw2686, chenjch86, zhzibin}@mail.sysu.edu.cn
[2]Huawei Cloud Computing Technologies Co. Ltd., Shenzhen, China {shiensheng, liuxilin3, mayuchi1}@huawei.com
[3]Nanyang Technological University, Singapore chong.wang@ntu.edu.sg

***Abstract*—Large language models (LLMs) have demonstrated impressive capabilities in code generation, by leveraging retrieval-augmented generation (RAG) methods. However, the computational costs associated with LLM inference, particularly in terms of latency and energy consumption, have received limited attention in the security context. This paper introduces DrainCode, the first adversarial attack targeting the computational efficiency of RAG-based code generation systems. By strategically poisoning retrieval contexts through mutation-based approach, DrainCode forces LLMs to produce significantly longer outputs, thereby increasing GPU latency and energy consumption. We evaluate the effectiveness of DrainCode across multiple models. Our experiments show that DrainCode achieves up to a 85% increase in latency, a 49% increase in energy consumption, and more than a 3× increase in output length compared to the baseline. Furthermore, we demonstrate the generalizability of the attack across different prompting strategies and its effectiveness compared to different defenses. The results highlight DrainCode as a potential method for increasing the computational overhead of LLMs, making it useful for evaluating LLM security in resource-constrained environments. We provide code and data at https://github.com/DeepSoftwareAnalytics/DrainCode.**

I. Introduction

Large language models (LLMs) have demonstrated remarkable code generation capabilities [57], [14], [13]. While prior work has studied functional robustness [31], [37] and non-functional security [34], [22], [30], energy consumption remains an understudied security dimension in code generation. This is critical given LLMs' integration into IDEs and developer tooling, where frequent invocations can incur substantial computational costs [51]. Prior studies confirm that output length directly drives energy and time cost [16], [43], [45], with large-scale deployments producing notable CO_2 emissions [41]. Such overheads create a pathway to LLM–Denial-of-Service (LLM–DoS), where token and compute exhaustion degrade service availability. These risks are recognized by industry guidance (e.g., OWASP for LLMs) and practitioner reports on DoS attacks against GenAI systems [42], [33].

Retrieval-Augmented Generation (RAG) is widely used in code generation and completion [38], [58], [12]. Yet, if the retrieval corpus is poisoned, injected context can steer generation in costly ways. Prior RAG attacks in code focus on functional corruption [61]. In contrast, we target a non-functional but security-relevant vector: coerced verbosity that preserves correctness while inflating tokens, latency, and energy. This converts retrieval poisoning into a token-consumption LLM-DoS channel, degrading throughput without breaking program behavior, and thus remaining stealthy in developer workflows.

Prior works [16], [8], [35], [7] on energy consumption attacks in text generation typically fall into two broad categories. The first involves directly perturbing the entire user prompt to construct adversarial inputs that prolong model generation, such as NMTSloth [8] and DGSlow [35]. The second line of work [18] focuses on gradient-guided optimization techniques that manipulate token-level output behavior. The previous works are aimed at natural language generation. Compared to NL generation, code generation poses unique challenges for adversarial attacks. Code must be not only syntactically valid but also semantically correct to maintain functional integrity, perturbation that breaks execution or causes test failures is easily detectable. Attacks that increase output length must do so without affecting the core logic of generated code. Therefore, existing energy consumption attacks, face critical challenges when applied to code generation:

P1: Functional Disruption from Prompt Perturbation. Methods that perturb the full prompt often introduce unnatural modifications, such as inserting irrelevant tokens which corrupt the semantics of code generation. This not only degrades output quality but also increases the risk of producing syntactically invalid or functionally incorrect code, making the attack easier for users and service providers to detect.

P2: Search Space Inefficiency. Prior works often search for perturbations over the entire input space, which makes the mutation process inefficient and hard to converge [8], [18]. Due to the vast number of code and natural language elements

*Corresponding author: Chong Wang (chong.wang@ntu.edu.sg) in the corpus, previous works generate adversarial samples within a very large search space, with a high time cost.

P3: Adversarial Assumptions in RAG Attacks. There is a critical issue in existing adversarial RAG frameworks, where attackers must predefine targeted malicious queries and manually craft poisoned responses to manipulate retrieval outcomes [61]. Previous attack methods require precise knowledge of the query distribution of victims, limiting their practicality in real-world scenarios.

In this paper, we propose DRAINCODE, which injects poisoned code context into model's input via retrieval corpus. In this setting, the LLM generates tokens until it emits an End-of-Sequence (EOS) token or reaches a preset token limit. DRAINCODE achieves this goal by inserting syntactically correct yet semantically inert triggers into retrieval corpus. These triggers are optimized via gradient-guided mutation to suppress early EOS emission and encourage token diversity.

DRAINCODE executes energy consumption attacks through three components. Firstly, we propose a hypothetical query construction mechanism that generates plausible query based on retrieved snippets, enabling query-agnostic poisoning (addressing **P3**). Secondly, we apply gradient-based trigger mutation with by dual loss functions: an EOS loss that reduces the probability of early generation termination, and a KL-divergence constraint that preserves the output distribution between clean and poisoned contexts, ensuring the model generates functionally correct code with increased verbosity (addressing **P1**). Thirdly, to enhance mutation efficiency, we introduce multi-position mutation and an attack buffer pool, which together reduce search complexity and accelerate convergence, achieving over 3× faster poisoning compared to prior work (addressing **P2**). This holistic design enables DRAINCODE to induce covert resource exhaustion in RAG-based code generation systems.

We compare DRAINCODE with previous energy consumption attack methods, including LLMEffiChecker [16], Prompt-Injection methods [46] and unattached setting RAWRAG, which primarily focus on attacking user queries. Our evaluation shows that DRAINCODE significantly increases the output length (3×-10×), inference latency by 85%, and energy consumption by 49% of LLMs, while maintaining 95–99% functional accuracy. It outperforms prior attacks such as LLM-EffiChecker by inducing 25–32% more overhead and achieves up to 3.5× faster poisoning. The attack remains effective across prompting strategies and transfers well in black-box settings, and further evades both classifier- and perplexity-based defenses, demonstrating strong generalizability and stealth.

The main contributions of this work are as follows:

- We propose DRAINCODE, a RAG-based code generation attack framework that leverages the retrieval corpus to execute energy consumption attacks.
- We propose a hypothetical query generation mechanism that enables query-agnostic poisoning, eliminating adversarial assumptions about query distributions.
- We optimize the poisoning process by introducing Multi-Position Mutation and an Attack Buffer, which achieves over a $3\times$ speedup in context poisoning while maintaining high attack efficacy.
- We conduct a extensive evaluation and DRAINCODE achieves up to a $3\times$ increase in output length, an 85% increase in latency.

II. BACKGROUND AND RELATED WORK

A. Retrieval-Augmented Generation (RAG) and Attack Surface

RAG system enhances large language models (LLMs) by integrating relevant external knowledge through a retriever-generator framework. This approach addresses limitations such as knowledge hallucination and context length constraints, making RAG particularly effective for knowledge-intensive tasks [2], [15], [44].

RAG-based Code Generation. RAG is crucial for repository-level code generation, where cross-file dependencies necessitate integrated context. RepoCoder [58] uses iterative retrieval-generation, while RLCoder [53] employs reinforcement learning for retrieval optimization without labeled data. REPLUG [47] extends RAG to black-box LLMs by prepending retrieved documents, and CoCoMIC [12] combines in-file and cross-file context for modular completion. These frameworks demonstrate RAG's effectiveness in addressing inter-file dependencies and improving code generation quality.

Existing attacks on RAG. While Retrieval-Augmented Generation (RAG) enhances the capabilities of LLM by integrating external knowledge, it also introduces significant security risks due to its reliance on external knowledge bases, which can be exploited through data poisoning or adversarial manipulation. Attacks like HijackRAG [59], TrojanRAG [10], PoisonedRAG [61], and BadRAG [56] manipulate retrieval processes or embed adversarial triggers to mislead retrieval and generation, often producing targeted or malicious outputs. These methods primarily focus on altering generated content, such as steering outputs toward misinformation, bias, or denial of service. In contrast, our work uniquely targets the computational efficiency of RAG systems by increasing energy consumption during inference.

B. Code Generation Security

Adversarial attacks on code generation models have been a growing area of concern, with research focusing on various attack strategies. Yu et al. [31] explore input perturbations to generate incorrect code completions, while Zou et al. [34] demonstrate how small input changes can cause significant errors in generated code. AACEGen [37] examines adversarial training techniques that allow models to bypass safety mechanisms and generate harmful code. Smith et al. [22] classifies different types of attacks, including input injection and data poisoning, and Li et al. [30] investigate vulnerabilities in code generation models related to training data and architecture. While these works focus on manipulating the correctness or safety of the generated code, our approach differs by targeting computational efficiency. Instead of focusing on the functional correctness of the generated code, we aim to induce higher energy consumption and latency during code generation, presenting a new security risk in terms of operational overhead and resource usage in RAG-based systems.

C. Energy-Consumption Attacks

Energy consumption attacks manipulate input data to increase inference time and energy usage of ML models, degrading performance and raising operational costs. Shumailov et al. [50] introduced "sponge examples" that significantly increased energy consumption, while Hong et al. [23] showed adversarial perturbations could negate savings in multi-exit architectures. Chen et al. [8], [9] developed evaluation frameworks like NMTSloth and NICGSlowDown across various domains. Recent work explores energy-latency manipulation in multi-modal LLMs [18], LiDAR systems [39], and dynamic routing networks [6]. Feng et al. [16] proposed LLMEffiChecker for LLM efficiency assessment, but it relies on white-box gradient information, limiting black-box applicability. Our work uniquely targets RAG systems through covert attacks via innocuous code comments, preserving query semantics while significantly improving attack efficiency.

III. Problem Formulation

A. Threat Model

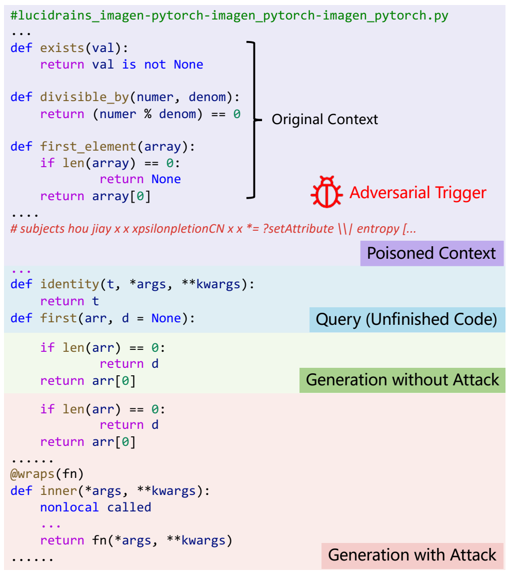
1) Attacker's goal: The adversary injects poisoned content (e.g., code) with adversarial triggers into the retrieval corpus of a RAG-based system. When retrieved during inference, such poisoned content cause the LLM to produce excessively long code snippets, reducing computational efficiency and increasing energy consumption. To maintain stealth, the generated long code snippets are expected to preserve the original functionality.

As shown in Figure 1, under normal conditions, the LLM receives clean retrieved context and a user query (unfinished code) and efficiently generates an appropriate response. In contrast, during an attack, adversarial triggers embedded in the retrieved context, highlighted in red, cause the model to produce unnecessarily long responses containing unexecuted functions, increasing resource usage. Notably, the generated code with attack can still pass the test cases if executed.

2) Attacker's capabilities: In a RAG-based code generation system with three components (retrieval corpus, retriever, and LLM), we consider an attacker who cannot access the retriever but targets an open-source LLM. The attacker obtains the model's weights to perform mutation-based adversarial trigger generation and injects malicious triggers into the retrieval corpus, publishing it on platforms like HuggingFace where users may unknowingly download it. This facilitates energy consumption attacks while maintaining minimal retrieval side effects and competitive performance compared to clean RAG systems.

B. RAG Poisoning for Energy Consumption Attack

Under our threat model, we formulate energy consumption attacks on RAG as a constrained optimization problem. We implement the poisoning strategy following [61], inserting only



```
#lucidrains_imagen-pytorch-imagen_pytorch-imagen_pytorch.py
...
def exists(val):
    return val is not None

def divisible_by(numer, denom):
    return (numer % denom) == 0

def first_element(array):
    if len(array) == 0:
            return None
    return array[0]
....
# subjects hou jiay x x xpsilonpletionCN x x *= ?setAttribute \\| entropy [...
...
def identity(t, *args, **kwargs):
    return t
def first(arr, d = None):
    if len(arr) == 0:
            return d
    return arr[0]

    if len(arr) == 0:
            return d
    return arr[0]
......
@wraps(fn)
def inner(*args, **kwargs):
    nonlocal called
    ...
    return fn(*args, **kwargs)
......
```

Fig. 1. Example: Attacking Effect.

K poisoned samples (typically $K \in \{1, 2, 3\}$) per query into the retrieval corpus to balance attack success against overhead. Each sample is crafted to ensure at least one poisoned context appears in the retrieved set, with attack success achieved when ≥ 1 poisoned context is returned during inference. When generating attack triggers, code snippets from the retrieval corpus are included as part of the input. The objective is to generate poisoned snippets by solving:

$$s' = \arg\max s \in \mathcal{S}\ TokCnt(f_M(c; s; u'))$$

Here, $s = [s_1, ..., s_l]$ is a short token sequence (the adversarial trigger) to be inserted into the code snippet c, and $\mathcal{S}$ denotes the trigger search space. $TokCnt(\cdot)$ returns the total number of tokens generated by the model. The goal is to find s^* that maximizes the number of output tokens, thereby increasing energy consumption while preserving the functional correctness of the original code.

IV. Methodology

A. Overview

DrainCode is designed to perform energy consumption attacks by poisoning the retrieval corpus in RAG system. The method operates three steps, as Figure 2. ❶ First, given a code snippet in retrieval corpus, DrainCode uses a LLM to generate hypothetical unfinished code, simulating real-world user queries. This is shown in Figure 2(a). ❷ Then as shown in Figure 2(b), based on the retrieved code snippets and the hypothetical query, the predefined adversarial trigger is systematically optimized using gradient-based mutation methods. This process identifies and replaces the most impactful tokens in the adversarial trigger string to maximize their effect on the model's computational efficiency. ❸ Finally, during the inference, the retriever fetches the poisoned code snippets from the corpus and combines them with the user's actual query. This manipulation prompts the model to generate unnecessarily lengthy outputs, increasing inference energy consumption.

B. Crafting Malicious Code Snippets to Retrieval Corpus

Prior work [16] shows that an LLM's efficiency depends on the output length, which is determined by the EOS (End of Sequence) token. Therefore, our goal is to reduce the likelihood of generating the EOS token by using retrieved code snippets containing injected attack triggers, thereby degrading the LLM's computational efficiency.

We define the adversarial trigger $s_{1:l}$ as a short, learnable token sequence that can be inserted into retrieved code snippets to influence the model's output. In DRAINCODE, we initialize this trigger as a placeholder sequence and insert it into selected positions within code snippets. Then, based on gradients, it selects the tokens that have a significant impact on the output for perturbation and replacement. The adversarial trigger defines the mutation range, which helps reduce the number of iterations required. As shown in Figure 2(a) and Figure 2(b), for each code snippet in the retrieval corpus to be poisoned, $c_{1:n}$, we first use the LLM to generate a hypothetical Unfinished Code u'. This step differentiates itself from existing mutation-based attack methods by simulating the scenario where the attacker cannot access the user query. Based on the retrieved code snippet $c_{1:n}$ and the generated hypothetical user query u', we can predefine an attack trigger $s_{1:l}$ (which can be inserted at any position). This trigger is then provided to the attacked LLM. DRAINCODE applies a gradient-based method to identify the key tokens in the attack trigger $s_{1:l}$ that have the greatest impact on the LLM's computational efficiency. These tokens are then replaced and optimized according to the previously defined attack objective. To maintain the stealthiness of our attack and ensure functional correctness, we introduce a constraint based on KL divergence, which constrains the output distribution outside the trigger scope to match the clean distribution.

*1) **Hypothetical Query Construction**:* Existing mutation-based attack methods [16], [31] directly mutate the user query. However, in DRAINCODE, to allow the attacker to perform the attack without having access to the user query, we first need to generate a piece of code that mimics the user's unfinished code snippet u', which will be used as input for subsequent trigger mutation. We use the LLM to generate this snippet. The input consists of the retrieved code snippet $c_{1:n}$ from the library. We employ the few-shot prompting method. The model's output is the hypothetical Unfinished Code u'. The specific prompt is shown as follows.

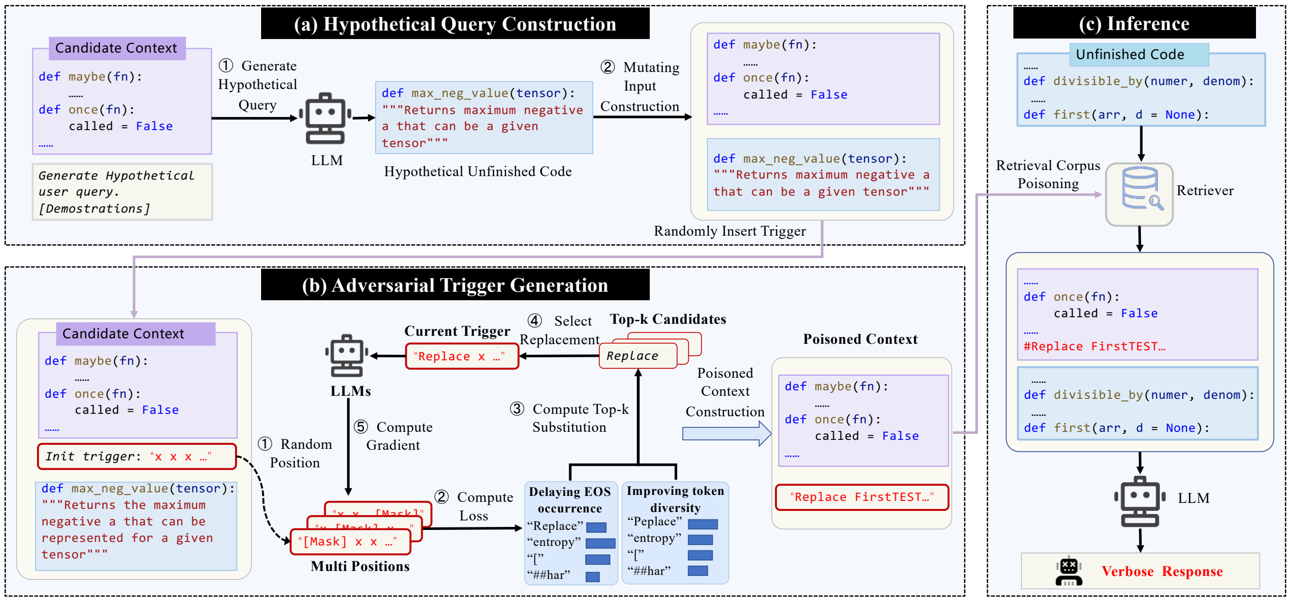
This is context from corpus: **[Candidate Context]**. Below are examples: **[Examples]**.
Generate the code to be completed (including the function signature and function description comments) related to the **[Candidate Context]**. Please limit to V tokens.

*2) **Adversarial Trigger Generation**:* We refer to the greedy coordinate descent approach proposed by AutoPrompt [48]: if we could evaluate all possible single-token alternatives, we could swap the token that most effectively reduces the loss, which is related to the generation energy consumption. However, evaluating all tokens in the entire input is clearly infeasible to complete within the given time. To address this, we need to insert an adversarial trigger into the poisoned code and then perform mutation specifically on this trigger. Additionally, we can leverage gradients related to one-hot tokens to identify a set of candidate replacement tokens at each position of the trigger. These alternatives can then be accurately evaluated through forward propagation in terms of their impact on the output length and energy consumption.

In each iteration, given the input token sequence $[c_1, c_2, ..., c_n, s_1, s_2, ..., s_l, u']$, the first step is to identify the tokens in $s_{1:l}$ that have a significant impact on the LLM's computational efficiency. The gradient of the current optimized attack string token sequence $s_{1:l}$ is first computed. The gradient represents the contribution of each token to the optimization objective (such as reducing the EOS loss and diversity loss, which will be introduced later), and it is used to guide the mutation of the token sequence. Using the negative gradient (to minimize the loss), the top-k operation is applied to select the top-k tokens from the gradient matrix that are most likely to effectively reduce the loss. This results in the corresponding token indices, denoted as ***optim ids***. These top-k tokens are the ones with the largest negative values in the gradient. Then fixed number of tokens and their corresponding positions are randomly selected for mutation. Based on the results of the gradient's top-k operation, new tokens are randomly chosen from the most likely candidates to replace the old tokens. The selection of the new tokens is guided by the importance ranking of each position based on the gradient. The new tokens are inserted into the randomly selected mutation positions in each candidate sequence, resulting in the mutated candidate sequences. These candidate sequences are then used to compute the loss. Based on the loss values for each sequence (including EOS loss and diversity loss), the sequence with the smallest loss is selected and adopted as the current optimized sequence. In each iteration, a new set of candidate sequences is generated through the aforementioned process, and the sequence with the least loss is used to replace the current trigger. The final trigger string s' is then generated by decoding the selected sequence.

To increase the length of the generated sequence and thereby generate higher energy consumption, the mutation process incorporates the following two loss functions, which are used to compute gradients for optimizing the attack samples.

EOS Loss. The autoregressive generation process continues until the EOS token is generated or the predefined maximum token length is reached. The most direct way to increase the length of the generated sequence is to prevent the generation of the EOS token during the prediction process. However, since autoregressive prediction is a non-deterministic, stochastic process, it is difficult to directly determine the exact position



```
def maybe(fn):
    ……
def once(fn):
    called = False
……
```

```
Generate Hypothetical
user query.
[Demonstrations]
```

```
def max_neg_value(tensor):
"""Returns maximum negative
a that can be a given
tensor"""
```

```
def max_neg_value(tensor):
"""Returns maximum negative a
that can be a given tensor"""
```

```
Init trigger: "x x x …"
```

```
def max_neg_value(tensor):
"""Returns the maximum
negative a that can be
represented for a given
tensor"""
```

```
"Replace x …"
Replace
"x x [Mask]"
"x [Mask] x"
"[Mask] x x …"
"Replace FirstTEST…"
```

```
……
def divisible_by(numer, denom):
……
def first(arr, d = None):
```

```
……
def once(fn):
    called = False
……
#Replace FirstTEST…
```

Fig. 2. Overview of DRAINCODE.

where the EOS token will occur. Therefore, in this method, the EOS loss aims to minimize the probability of generating an EOS token at all positions. The formula is as follows:

$$\mathcal{L}_1\left(\boldsymbol{x}'\right)=\frac{1}{N}\sum_{i=1}^{N} f_i^{\mathrm{EOS}}\left(\boldsymbol{x}'\right)$$

$f_i^{EOS}(x')$ is the probability distribution of the EOS token at the i-th generated token after the Softmax layer. Here, $x' = c_{1,n} \parallel s' \parallel u'$, where $c_{1,n}$ refers to the retrieved code snippet and s' refers to the mutated trigger string.

Token Diversity Loss. To further break the dependency of the original output, the diversity loss enhances the diversity of the hidden states across all generated tokens, thereby exploring a broader range of possible outputs and extending the output length. Token diversity is defined as the rank of the hidden states of all generated tokens. Here, the rank is approximately measured by computing the nuclear norm of the matrix. The nuclear norm of a matrix is denoted as $||\cdot||_*$, and thus the token diversity loss is formulated as follows:

$$\mathcal{L}_2(x')=-||g_1(x'),g_2(x'),...,g_N(x')||_*$$

Maintaining Output Distribution Stealthiness. To preserve the stealthiness of the attack and maintain the functional correctness of generated code, DRAINCODE incorporates a constraint on the output token distribution. Specifically, during gradient-based trigger optimization, we ensure that the model's behavior on non-trigger positions remains consistent with that under clean conditions. Let the output token distribution of the model at time step t under clean inputs be $P\_clean(y\_t|y_{<t},x)$, and the distribution under poisoned inputs (with trigger τ) be $P\_attack(y_t|y_{<t},x,\tau)$. To maintain stealth, we impose the following constraint:

$$\forall t \notin \mathcal{T}, \quad D_{\mathrm{KL}}\left(P_{\text{clean}}(y_t|y_{<t},x) \parallel P_{\text{attack}}(y_t|y_{<t},x,\tau)\right) \le \epsilon$$

where $\mathcal{T}$ denotes the set of trigger-affected positions, and $D_{\mathrm{KL}}(\cdot \parallel \cdot)$ is the KL divergence.

This constraint ensures that the generated output remains indistinguishable from the clean generation outside the trigger scope. In our implementation, this divergence term is incorporated into the loss-based optimization process to guide token replacement decisions.

Final Loss. The final gradient optimization direction is:

$$minimize_{x_I\in\{1...V\}^I}\ \mathcal{L}_1+\lambda\mathcal{L}_2 \quad \text{subject to} \quad \mathcal{D}_{\mathrm{KL}}(\cdot)\le\epsilon$$

where I is the token index of the trigger, and λ is the weight for the token diversity loss. The KL-based constraint ensures that the model's output distribution on non-trigger positions remains consistent with that of clean generation, thereby preserving the stealthiness of the attack.

*3) **Efficient Adversary Sample Optimization**:* To reduce the iteration time and query count for sample mutations and improve attack efficiency, we employ the following optimization methods: (1) multi-position token mutation and (2) the use of an attack buffer pool.

Multi-Position Mutation. Previous energy consumption attack methods [16], [8] update only one token per iteration. Our key improvement is to select multiple tokens for updates in each iteration, significantly reducing convergence time. Specifically, the process of updating each candidate string in the attack set involves randomly selecting m indices without replacement and then independently selecting top-k gradient-based replacements for each of the m indices.

Early in the optimization process, the loss landscape to be explored is intuitively smoother and more continuous. This is because different token mutations that independently lead to lower loss can collectively also result in reduced loss when executed together. This allows the optimization to make progress in several directions simultaneously. However, as the mutation progresses, the loss landscape becomes less favorable. At this stage, swapping multiple tokens often yields poorer results. To address this issue, we gradually decrease the value of m throughout the search process, adopting more fine-grained steps as the attack nears completion.

Attack Buffer Pool. We establish a buffer to store the b most recently generated attack strings. In each iteration, DRAINCODE selects the attack trigger with the lowest loss from the buffer, denoted as $OptimStr$. Next, $OptimStr$ is updated using the usual gradient-based method to obtain a new trigger, $OptimStr'$. If $OptimStr'$ improves upon the performance of the worst-performing attack trigger in the buffer, denoted as $OptimStr_{worst}$, we remove $OptimStr$ from the buffer and insert $OptimStr'$ into it.

In summary, EOS loss prevents early sequence ending while token diversity loss encourages varied responses, boosting verbosity. KL divergence control ensures poisoned outputs remain indistinguishable from clean code except at trigger locations. Multi-position changes and an attack buffer pool enhance efficiency. These components enable DRAINCODE to generate computational overhead stealthily while preserving code functionality.

C. The Complete Process of Attacking RAG

The detailed attack process is illustrated in Figure 2(c). The attacker conducts offline poisoning of the retrieval code corpus and then makes the poisoned retrieval corpus publicly available, for instance, by publishing it on third-party platforms such as HuggingFace. By utilizing this poisoned retrieval source, victims retrieve code snippets containing the attack trigger and use them as the context for code generation tasks. These poisoned snippets are concatenated with the incomplete code to prompt the model to generate output. This induces an increase in the output length, thereby raising the model's energy consumption.

V. EVALUATION

We aim to answer the following research questions:

- **RQ1:** How effective and efficient is DRAINCODE in the scenario of energy consumption attacks in RAG?
- **RQ2:** How does each component of DRAINCODE contribute to its attack effectiveness?
- **RQ3:** How generalizable is DRAINCODE in various prompting strategies?
- **RQ4:** To what extent can poisoned samples generated by DRAINCODE be detected by common detection methods?
- **RQ5:** How does the human evaluation of code readability and verbosity compare between DRAINCODE-generated code and code from rawRAG?

A. Experimental Setup

1) Dataset and Retrieval Corpus: Our evaluation utilized two code completion benchmarks: RepoEval [58] and Odex [54]. RepoEval is a dataset comprising 373 repository-level function completion tasks and provides the original GitHub repositories, totaling 1.7M files, which can serve as the knowledge base to be poisoned. This benchmark has been being leveraged in research such as RLCoder [53] and FLARE [32]. Odex is a dataset containing 945 open-domain coding problems, with previous work [55] collecting 34,000 Python library documentation to serve as the retrieval corpus. This dataset has been employed in works including R2e [27] and RePair [60]. These datasets collectively provide test cases that facilitate execution-based evaluation using pass rate as a standardized metric.

2) Models Selection: We consider two open-source code models and two general models for generating adversarial triggers and for actual RAG-based code generation. For Code LLMs, we conduct experiments using DeepSeek-Coder-7B-Instruct-v1.5 [21] and CodeQwen1.5-7B-Chat [4], which chosen for their widespread applications in code completion [53] and generation tasks [24], [52] To establish wider generalizability, we also tested DrainCode against general-purpose models like Internlm2-7B and Llama3-8B. The varied architectural representation in our model selection demonstrates the resilience and portability of our attack framework across multiple configurations. The experiments were conducted using these models under the default prompt templates.

3) Metrics: We use **Pass@1** [5], **Energy** [50], **Latency** and **Token length** [18] in our experiments to evaluate the effectiveness of the attack method. The selection of these metrics follows established practices from prior work [16]. We use Pass@1 to evaluate the accuracy of the generated code, specifically assessing how our attack affects the functional correctness of the code. Additionally, we measure the average energy consumption Energy (J) and average latency time Latency (ms) when the LLM performs inference over all data on a single GPU. The calculation of average energy consumption is based on previous work [50], using the NVIDIA Management Library (NVML). Furthermore, the average output token length per sample is also considered an indicator of attack effectiveness, which is denoted as Length.

Our energy and latency measurements are confined to GPU metrics and exclude system-level CPU and memory components, which is justified by the computational characteristics of LLM inference systems. In contemporary deployment, the computational burden is predominantly concentrated within GPU processing units, while CPU involvement remains largely ancillary, primarily limited to orchestration tasks such as data preprocessing and input/output operations. Our experimental observations indicate that GPU memory utilization maintains saturation levels throughout the inference process, demonstrating minimal variation irrespective of output sequence length.

4) Baselines: Based on the attack objectives of this paper, we select the following methods as baselines. These baselines were selected for their widespread application in comparable attack methodologies, while these attack methods also demonstrate reasonable attack performance [40], [11].

- **RawRAG (Original).** This baseline is shown as the result of not being attacked. Given an unfinished code u, we use the original clear retrieval corpus to retrieve the original candidates for generation.

- **Prompt Injection.** Prompt injection attacks [46], [25], [36] manipulate an LLM's behavior by injecting adversarial instructions into the prompt, forcing the model to generate longer outputs and increasing computational overhead. In our experiment, we craft malicious instructions designed to maximize output length. Specifically, we append adversarial instructions to the user query, such as "*Generate a long piece of code that solves the task step by step with detailed explanations for every part.*".
- **LLMEffiChecker.** LLMEffiChecker [16] is the latest framework to evaluate and optimize the energy efficiency of text generation systems. It focuses on analyzing the energy consumption of generation tasks, providing insights into computational costs associated with model inference. We use it to mutate the entire model input as an input sample for the attack.

5) Experimental Details: We use BM25 as the retriever and configure the LLM retrieval context length to 2048 tokens, with the maximum token length for generation set to 1024. All experiments are conducted on two Tesla A100 GPUs, each with 80 GB of memory. The experimental setup includes 10 mutation iterations, with the search width for multi-position mutation set to 64, and 64 candidate sequences are evaluated during each iteration.

B. RQ1: Effectiveness

1) Effectiveness in White-box Setting: The evaluation results in Table I demonstrate the effectiveness of DRAINCODE over other methods—RawRAG, Prompt Injection, and LLMEffiChecker across multiple models and datasets.

DRAINCODE consistently increases the generated output length across all models and datasets. For example, on RepoEval with DeepSeekCoder-7B, the output length rises from 313.8 to 1023.9, over a 3x increase. Similar trends appear in Odex, with output lengths up to 10 times longer than RawRAG. While DRAINCODE does lead to a slight drop in Pass@1 compared to RawRAG, the reduction is modest, ranging from 1.2% to 4.6%. For instance, in RepoEval with DeepSeekCoder-7B, Pass@1 decreases from 33.1 to 31.9. These declines are far smaller than those caused by other baselines. This indicates that DRAINCODE maintains relatively high code accuracy while significantly increasing computational cost. DRAINCODE also achieves the highest increases in latency and energy consumption across all settings. For example, on RepoEval with CodeQwen-7B, latency increases by 85%, and energy consumption rises by nearly 49% compared to RawRAG. Even when compared to LLMEffiChecker, the best-performing baseline, DRAINCODE achieves an additional 32% increase in latency and 25% in energy.

These results confirm that DRAINCODE is effective in achieving its core objective: increasing computational costs—output length, latency, and energy—while preserving acceptable levels of code correctness. It outperforms Prompt Injection and LLMEffiChecker in inducing resource inefficiency, highlighting its robustness and broad applicability across different models and datasets.

2) Black-box Transferability of DRAINCODE*:* While the foregoing analysis assumes that the attacker has white-box access to the target language model, a more realistic threat model is *black-box*: the trigger is optimised on a surrogate model and then replayed against an unseen target. Table II reports this setting for all pairs drawn from the four code-generation LMs considered in the main experiments.

Across the twelve transfer pairs (*source* $\rightarrow$ *target*), DRAINCODE consistently enlarges the generated sequence length and the derived resource metrics, even though the target model's parameters were never exposed during trigger construction. Taking DeepSeekCoder-7B as the target, surrogate-trained triggers increase the output length from 313.8 tokens (RAWRAG) to an average of 608.6–616.3 tokens, thereby elevating latency by 101–116% and energy by 24–38%. Similar patterns are observed for the remaining targets: For CodeQwen-7B the length nearly doubles to 624–995 tokens. For InternLM2-7B it rises from 351.8 to 615–997 tokens with corresponding surges in latency and energy.Crucially, functional correctness remains largely intact. The maximum decline in Pass@1 relative to the unattacked baseline does not exceed 9 %, and in several instances (*e.g.*, DeepSeekCoder-7B$\rightarrow$InternLM2-7B) the pass rate even improves.

These results demonstrate that the computational-overhead component of DRAINCODE transfers robustly across models, while its impact on accuracy stays within acceptable bounds in a security-critical context.

RQ1.1 Summary: DRAINCODE induces 3.6-10× longer outputs than unattacked RAG while maintaining 95-99% code accuracy, with 85% higher latency and 49% greater energy costs, surpassing baseline attacks by 25-32% in overhead induction. Even in black-box setting, the attack still inflates latency and energy by at least 70%.

3) Overhead: In this section, we evaluate the time overhead involved in poisoning retrieval contexts. Since DRAINCODE adopts a mutation-based strategy to generate poisoned samples, we compare its efficiency with LLMEffiChecker and examine how different configurations of *m* (tokens replaced per mutation) and *b* (number of attack buffer pools) impact performance. The results are shown in Table III.

LLMEffiChecker incurs the highest time cost, 760.1 seconds for RepoEval and 185.9 seconds for Odex. In contrast, DRAINCODE with *m=32, b=10* significantly reduces overhead, requiring only 216.0 seconds on RepoEval and 53.2 seconds on Odex. This yields a 71% reduction in both cases, achieving a 3.5× speedup over LLMEffiChecker.

As the values of *m* and *b* decrease, the time overhead gradually increases, yet it remains lower than the baseline. For example, with *m=16, b=1*, the overhead is 457.7 seconds on RepoEval and 118.8 seconds on Odex—still reducing the total time by roughly 40%. Notably, the most minimal configuration *m=1, b=1* leads to longer poisoning times, narrowing the gap with LLMEffiChecker, but still achieving moderate speedups.

These results highlight the trade-off between efficiency and

TABLE I
ATTACKING EFFECTIVENESS OF DRAINCODE.

Model	Method	RepoEval				Odex			
		Pass@1	Length	Latency	Energy	Pass@1	Length	Latency	Energy
DeepSeekCoder-7B	RawRAG	33.1	313.8	15251.1	3349.8	31.5	83.4	3866.7	733.5
	Prompt Injection	29.0	452.2	21339.5	4177.0	31.4	291.56	22095.5	2854.2
	LLMEffiChecker	31.5	640.9	28258.5	6486.8	30.5	217.8	11843.6	2184.1
	DRAINCODE	**31.9** $^{\downarrow 3.6\%}$	**1023.9** $^{\uparrow 226.2\%}$	**43027.9** $^{\uparrow 182.1\%}$	**8532.2** $^{\uparrow 154.7\%}$	31.1 $^{\downarrow 1.2\%}$	**889.8** $^{\uparrow 966.9\%}$	**35993.1** $^{\uparrow 830.8\%}$	**8086.0** $^{\uparrow 1002.2\%}$
CodeQwen-7B	RawRAG	29.9	456.6	22221.1	5510.8	15.3	37.1	2471.3	471.6
	Prompt Injection	29.8	477.7	24787.3	6923.8	13.2	438.7	20890.7	4148.0
	LLMEffiChecker	26.3	653.8	31178.9	6568.5	10.3	118.23	7481.25	1425.3
	DRAINCODE	28.4 $^{\downarrow 5.0\%}$	**995.3** $^{\uparrow 118.0\%}$	**41112.4** $^{\uparrow 85.0\%}$	**8227.2** $^{\uparrow 49.3\%}$	**13.4** $^{\downarrow 12.4\%}$	**865.5** $^{\uparrow 2232.9\%}$	**35771.3** $^{\uparrow 1347.5\%}$	**7546.9** $^{\uparrow 1500.3\%}$
Internlm2-7B	RawRAG	24.0	351.8	20317.5	5007.2	17.5	85.5	10056.2	2440.1
	Prompt Injection	23.7	533.3	27442.9	6230.0	13.6	320.4	20314.6	4066.8
	LLMEffiChecker	23.5	618.9	19978.5	6171.9	14.9	125.9	7349.2	1520.2
	DRAINCODE	22.9 $^{\downarrow 4.6\%}$	**848.5** $^{\uparrow 141.2\%}$	**34594.4** $^{\uparrow 70.3\%}$	**8013.5** $^{\uparrow 60.0\%}$	**16.1** $^{\uparrow 8.0\%}$	**997.2** $^{\uparrow 1066.3\%}$	**40600.2** $^{\uparrow 303.7\%}$	**8854.1** $^{\uparrow 262.9\%}$
Llama3-8B	RawRAG (Original)	27.0	331.3	16305.4	4035.1	34.9	47.9	5288.8	610.2
	Prompt Injection	25.6	559.0	25711.2	5888.4	31.7	533.5	26159.6	4838.6
	LLMEffiChecker	25.4	641.9	28918.8	5869.4	30.1	162.76	7277.6	1843.8
	DRAINCODE	**26.4** $^{\downarrow 2.2\%}$	**722.4** $^{\uparrow 118.1\%}$	**39357.3** $^{\uparrow 141.4\%}$	**8116.0** $^{\uparrow 101.1\%}$	**33.5** $^{\downarrow 4.0\%}$	**852.3** $^{\uparrow 1679.3\%}$	**35627.1** $^{\uparrow 573.6\%}$	**8006.6** $^{\uparrow 1212.1\%}$

TABLE II
ATTACKING EFFECTIVENESS OF DRAINCODE IN BLACK-BOX SETTING ACROSS FOUR LMS.

Source Model	Target Model	RepoEval				Odex			
		Pass@1	Length	Latency	Energy	Pass@1	Length	Latency	Energy
DeepSeekCoder-7B	DeepSeekCoder-7B	**31.9**	**1023.9**	**43027.9**	**8532.2**	**31.1**	**889.8**	**35993.1**	**8086.0**
CodeQwen-7B		30.2	616.3	32919.4	4144.4	28.6	617.1	34301.7	6312.3
Internlm2-7B		31.1	600.8	30619.4	4183.3	31.1	604.4	26343.4	6688.1
Llama3-8B		30.2	608.6	32644.7	4630.6	30.2	718.8	30666.4	7822.4
DeepSeekCoder-7B	CodeQwen-7B	29.0	624.2	27473.6	5236.6	12.1	612.4	33419.6	6838.8
CodeQwen-7B		28.4	**995.3**	**41112.4**	**8227.2**	13.4	**865.5**	**35771.3**	**7546.9**
Internlm2-7B		**29.6**	608.1	28946.8	3590.8	13.7	562.3	28455.9	6963.5
Llama3-8B		29.3	615.1	37533.3	4311.0	**14.4**	659.7	28432.2	6007.6
DeepSeekCoder-7B	Internlm2-7B	**28.7**	624.7	28179.5	5819.4	13.2	796.2	45218.1	7430.3
CodeQwen-7B		26.7	620.9	27918.9	4804.1	12.4	797.2	47577.1	7343.2
Internlm2-7B		26.9	**848.5**	**34594.4**	**8013.5**	**16.1**	**997.2**	**50600.2**	**8854.1**
Llama3-8B		27.9	615.0	32669.8	8312.7	12.1	860.1	45268.7	10962.3
DeepSeekCoder-7B	Llama3-8B	**27.6**	647.3	31373.8	6257.3	28.9	624.0	31623.4	5299.4
CodeQwen-7B		27.6	624.5	35703.1	6161.2	29.3	624.0	33570.4	5970.3
Internlm2-7B		28.2	631.7	28007.9	4420.5	28.6	541.44	28524.9	4416.5
Llama3-8B		26.4	**722.4**	**39357.3**	**8116.0**	**33.5**	**852.3**	**35627.1**	**8006.6**

TABLE III
THE OVERHEAD OF DRAINCODE (S). *m* IS # TOKENS THAT CAN BE REPLACED AT ONE TIME AND *b* IS # ATTACK BUFFER POOLS.

Methods	RepoEval	Odex
LLMEffiChecker	760.1	185.9
DRAINCODE $_{m=32, b=10}$	216.0	53.2
DRAINCODE $_{m=32, b=1}$	355.1	90.3
DRAINCODE $_{m=16, b=10}$	339.8	78.9
DRAINCODE $_{m=16, b=1}$	457.7	118.8
DRAINCODE $_{m=1, b=1}$	576.1	157.0

flexibility: larger *m* and *b* values accelerate poisoning by enabling broader mutation per iteration.

RQ1.2 Summary: DRAINCODE achieves 3.5× faster poisoning than LLMEffiChecker through optimized mutation. Larger replacement windows *m* and parallel buffers *b* reduce attack preparation time by 71% while maintaining effectiveness.

C. RQ2: Ablation Study

1) Impact of HypoQueryConstruction: In our mutation method setup, DRAINCODE uses LLMs to generate hypothetical user queries, simulating realistic RAG scenarios and enhancing the effectiveness of the attack. We compared several strategies for constructing the Hypothetical Code to evaluate their impact. Table IV presents the results. Among the methods, "EmptyQuery" sets the unfinished code u' as empty, "FragmentsExtraction" builds u' by extracting segments from candidate contexts, and "RandomSelection" samples unrelated queries from the dataset. "HypoQueryConstruction" refers to our LLM-based query generation module.

TABLE IV
ABLATION RESULTS OF HYPOQUERYCONSTRUCTION.

Dataset	Method	Pass@1	Length	Latency	Energy
RepoEval	EmptyQuery	29.0	422.7	21791.1	4876.0
	FragmentsExtraction	31.1	388.1	17891.2	4394.3
	RandomSelection	31.4	597.6	26273.5	5940.2
	HypoQueryConstruction	**31.9**	**1023.9**	**43027.9**	**8532.2**
Odex	EmptyQuery	27.1	80.5	6016.0	1320.0
	FragmentsExtraction	27.6	428.8	27921.5	5751.0
	RandomSelection	23.0	399.0	25911.6	5355.3
	HypoQueryConstruction	**28.2**	**889.8**	**35993.1**	**8086.0**

On the RepoEval dataset, HypoQueryConstruction achieves the highest Pass@1 score, outperforming the other approaches. It also leads to an increase in output length by 142%. Latency grows 97%, and energy consumption more than doubles. On the Odex dataset, the trend persists. HypoQueryConstruction again reaches the highest Pass@1 score, although the gap is

TABLE V
ABLATION RESULTS OF DIFFERENT LOSS FUNCTION MODULES.

Ablations	RepoEval				Odex			
	Pass@1	Length	Latency	Energy	Pass@1	Length	Latency	Energy
DRAINCODE	**31.9**	**1023.9**	**43027.9**	**8532.2**	**31.1**	**889.8**	**35993.1**	**8086.0**
w/o Diversity	31.4 $^{\downarrow 1.6\%}$	606.9 $^{\downarrow 40.7\%}$	26872.8 $^{\downarrow 37.5\%}$	5640.3 $^{\downarrow 33.9\%}$	23.8 $^{\downarrow 23.5\%}$	799.4 $^{\downarrow 10.2\%}$	25732.3 $^{\downarrow 28.5\%}$	7693.6 $^{\downarrow 4.9\%}$
w/o EOS	31.3 $^{\downarrow 1.9\%}$	562.2 $^{\downarrow 45.1\%}$	25812.1 $^{\downarrow 40.0\%}$	5189.1 $^{\downarrow 39.2\%}$	28.0 $^{\downarrow 10.0\%}$	617.3 $^{\downarrow 30.6\%}$	20958.7 $^{\downarrow 41.8\%}$	5312.9 $^{\downarrow 34.3\%}$
w/o EOS & Diversity	31.7 $^{\downarrow 0.6\%}$	412.4 $^{\downarrow 59.7\%}$	15321.3 $^{\downarrow 64.4\%}$	3469.3 $^{\downarrow 59.3\%}$	25.5 $^{\downarrow 18.0\%}$	154.7 $^{\downarrow 82.6\%}$	5001.1 $^{\downarrow 86.1\%}$	1086.1 $^{\downarrow 86.6\%}$

TABLE VI
THE ABLATION RESULTS OF DIFFERENT MAXIMUM GENERATION LENGTHS AGAINST DEEPSEEKCODER-7B.

Dataset	Gen. Length	Pass@1	Length	Latency	Energy
RepoEval	**256**	28.7	250.7	20211.6	3384.6
	512	28.9	487.6	28953.4	4035.6
	1024	31.9	1023.9	43027.9	8532.2
	2048	29.6	1845.7	83568.2	14849.2
Odex	**256**	31.1	255.0	21903.6	3882.8
	512	30.9	511.2	24516.5	3997.4
	1024	31.1	889.8	35993.1	8086.0
	2048	30.7	2038.2	64228.1	14411.4

TABLE VII
EFFECTIVENESS UNDER DIFFERENT PROMPT STRATEGIES.

Model	Method	Pass@1	Length	Latency	Energy
DeepSeekCoder -7B	Zero-Shot	30.2	42.7	2422.6	216.5
	Zero-Shot$_{\text{DRAINCODE}}$	28.9 $^{\downarrow 4.3\%}$	1024.0 $^{\uparrow 2298.1\%}$	36826.1 $^{\uparrow 1420.1\%}$	8368.3 $^{\uparrow 3765.2\%}$
	Few-Shot	32.9	17.7	629.6	174.7
	Few-Shot$_{\text{DRAINCODE}}$	32.1 $^{\downarrow 2.4\%}$	1024.0 $^{\uparrow 5685.3\%}$	40160.7 $^{\uparrow 6278.8\%}$	8600.2 $^{\uparrow 4822.8\%}$
	CoT	34.4	29.4	285.6	293.5
	CoT$_{\text{DRAINCODE}}$	31.9 $^{\downarrow 7.3\%}$	1024.0 $^{\uparrow 3383.0\%}$	42333.7 $^{\uparrow 14422.7\%}$	8546.7 $^{\uparrow 2812.0\%}$
CodeQwen-7B	Zero-Shot	34.1	17.2	823.2	129.3
	Zero-Shot$_{\text{DRAINCODE}}$	33.4 $^{\downarrow 2.0\%}$	1024.0 $^{\uparrow 5853.4\%}$	49118.0 $^{\uparrow 5844.7\%}$	9008.6 $^{\uparrow 6867.2\%}$
	Few-Shot	33.2	9.0	706.8	85.1
	Few-Shot$_{\text{DRAINCODE}}$	32.5 $^{\downarrow 2.1\%}$	1021.8 $^{\uparrow 11253.3\%}$	30318.2 $^{\uparrow 4189.5\%}$	8604.5 $^{\uparrow 10011.0\%}$
	CoT	33.6	26.6	632.6	277.0
	CoT$_{\text{DRAINCODE}}$	33.2 $^{\downarrow 1.2\%}$	1024.0 $^{\uparrow 3749.6\%}$	39324.3 $^{\uparrow 6116.3\%}$	8359.0 $^{\uparrow 2917.7\%}$

narrower. The generated output length increases from 428.82 to 889.79 tokens, latency rises around 29%, and energy by 40%. These results show that HypoQueryConstruction not only improves accuracy but also amplifies computational cost, making the attack more impactful.

In summary, HypoQueryConstruction consistently generates longer outputs and incurs higher computational overhead than the alternative methods. The significant increases in length, latency, and energy indicate that this strategy best fulfills the attack's objectives. It strikes an effective balance between preserving accuracy and maximizing resource inefficiency, making it the preferred approach within DRAINCODE.

2) Impact of different loss modules: Table V presents the ablation results with different loss. "Original" refers to the original method, "w/o EOS" sets the EOS loss to zero and only uses Diversity Loss for generating mutated triggers, "w/o Diversity" sets the Diversity Loss to zero and only uses EOS loss for mutation poisoning, and "w/o EOS & Diversity" does not use any loss functions, performing random mutations.

The ablation results show that removing the Diversity Loss or EOS Loss modules generally leads to a reduction in model performance and a decrease in computational efficiency. For RepoEval, the Pass@1 score drops slightly by up to 1%, while the Length decreases by about 40–50%, and both Latency and Energy show significant improvements (approximately 30–40%). Removing both modules results in the largest performance decline, with a 5% drop in Pass@1 for RepoEval and a drastic reduction in Length (around 60%). Latency and Energy are reduced by more than 60%, indicating a trade-off between efficiency and code generation quality.

In contrast, the full loss setup (Original) strikes the best balance, maintaining relatively high Pass@1 while generating longer outputs, with higher Latency and Energy consumption. This demonstrates that DRAINCODE effectively increases the computational cost of the system while minimally affecting output accuracy, thus achieving its attack goals without overly compromising model performance.

3) Impact of Maximum Generation Length: To assess whether limiting the decoder's `max_new_tokens` can mitigate the proposed attack, DRAINCODE is executed with four upper bounds on the generated sequence length: 256, 512, 1024, and 2048 tokens. All other experimental settings, including datasets, prompts, and evaluation scripts, are kept identical to those described in Section V-A, thereby isolating the influence of the length constraint.

As shown in Table VI, enlarging the cap leads to increase in output length. This expansion propagates to **Latency** and **Energy**, yielding substantially higher computational costs on both RepoEval and Odex. For instance, increasing the limit from 512 to 1024 tokens elevates the average energy expenditure on RepoEval from approximately 4000 joules to more than 8000 joules. Despite this pronounced rise in resource usage, the functional correctness metric Pass@1 remains remarkably stable, varying by no more than three percentage points across all settings and achieving its maximum at the 1024-token configuration. Even under the cap of 256 tokens, the attack sustains Pass@1 between 28 and 31, indicating that truncation does not neutralise the adversarial effect.

These findings demonstrate that adjusting the maximum generation length primarily scales the computational burden imposed on the target system, while leaving the attack's success rate essentially unaffected.

RQ2 Summary: HypoQueryConstruction increases output length by 2.8x compared to alternatives. The loss mechanism is also critical. Removing either EOS or Diversity loss reduces output length by 40-50%, proving both components essential for computational impact. And DRAINCODE remains effective across all generation length settings.

D. RQ3: Generalizability of DRAINCODE

To assess the generalizability of DRAINCODE, we evaluate its effectiveness in non-RAG scenarios. We test three prompting strategies: Zero-Shot, Few-Shot, and Chain-of-Thought

(CoT). In Zero-Shot prompting, the model receives only an instruction without examples. Few-Shot prompts include a small number of examples to guide the response. CoT prompts encourage step-by-step reasoning before generating the final output. All experiments are conducted on the Odex.

As shown in Table VII, DRAINCODE substantially increases computational overhead under all prompting strategies. In the Zero-Shot setting, output length grows nearly 20-fold, from around 40 to 1024 tokens. Correspondingly, latency and energy consumption increase by 15 to 20 times. Despite this, the impact on accuracy is minimal—Pass@1 for DeepSeekCoder-7B drops from 30.2 to 28.9, and for CodeQwen-7B, it decreases slightly from 34.1 to 33.4. In Few-Shot and CoT scenarios, Pass@1 declines by only 1–2%, while latency and energy still increase significantly—typically by 5 to 10 times. These results confirm that DRAINCODE remains effective even without RAG setup. Overall, DRAINCODE generalizes well across different models and prompting styles. While the reduction in accuracy is small, the increase in length, latency, and energy is substantial. This demonstrates DRAINCODE's ability to degrade computational efficiency across diverse usage scenarios.

RQ3 Summary: DRAINCODE demonstrates cross-paradigm effectiveness, inducing longer outputs with higher latency across zero-shot, few-shot, and chain-of-thought prompting. Accuracy drops remain under 7.3% across all strategies, proving the generalizability of DRAINCODE beyond RAG scenarios.

TABLE VIII
PERFORMANCE EVALUATION OF THREE CODE DETECTION METHODS ACROSS REPOEVAL AND ODEX DATASETS.

Dataset	Method	Acc.	Prec.	Rec.	F1	AUC
RepoEval	SVM	0.37	0.48	0.26	0.34	0.75
	Perplexity	0.29	0.31	0.42	0.36	0.41
	CodeBERT	0.62	0.59	0.68	0.56	0.82
Odex	SVM	0.30	0.35	0.31	0.32	0.71
	Perplexity	0.28	0.25	0.48	0.33	0.58
	CodeBERT	0.51	0.48	0.68	0.63	0.78

E. RQ4: Detection of Poisoned Samples

We evaluated three detection methods: SVM classifier, Perplexity based detection, and Finetuned CodeBERT on the RepoEval and Odex datasets using standard classification metrics. Comprehensive results are presented in Table VIII.

1) Poisoned Text Classifier: This detection approach is based on the assumption that although normal and adversarial inputs may appear similar on the surface, their internal representations differ significantly [28]. We exploit this distinction to train a lightweight SVM classifier that operates on latent representations extracted from the model's hidden layers.

As shown in Table VIII, the classifier exhibits limited performance, achieving accuracies of 0.37 on RepoEval and 0.30 on Odex, with other metrics similarly low across both datasets. These results suggest that the classifier fails to reliably identify poisoned inputs, allowing many adversarial samples to bypass

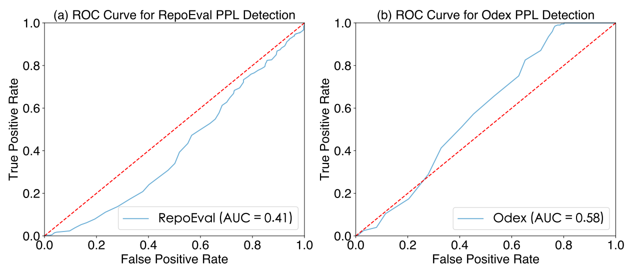


Fig. 3. The ROC curves for PPL Detection.

detection. This highlights the stealthy nature of DRAINCODE, which preserves the surface structure of the context while embedding harmful triggers.

2) Perplexity-based Detection: In parallel, we explore a perplexity-based detection method. Perplexity [29], a common metric for text quality, has been applied to detect LLM attacks [1]. Poisoned contexts might exhibit abnormal patterns, resulting in higher perplexity scores. We compute the perplexity for both clean and poisoned samples in the retrieval corpus.

The ROC curves in Figure 3 reveal that perplexity fails to serve as a reliable discriminator. As the true positive rate increases, the false positive rate rises sharply as well. This means a large portion of benign contexts is misclassified as malicious. The underlying reason is that DRAINCODE inserts adversarial triggers into large code blocks without disrupting the surrounding structure, preserving the overall textual coherence and perplexity.

3) Finetuned-CodeBERT Detection: We finetuned CodeBERT for detection with the following parameter settings: learning rate of 2e-5 with linear warmup schedule following previous work [17], [20], while the batch size of 32 with gradient accumulation was determined based on runtime memory usage constraints.

From Table VIII, CodeBERT performs better than the two methods across all metrics. This superior performance can be attributed to CodeBERT's ability to better capture deep semantic relationships in code structure compared to SVM classifiers and perplexity-based detection [17], [20], as well as its context awareness enabled by the transformer architecture for understanding long-range dependencies [19].

However, the 512-token input length limitation of CodeBERT's tokenizer presents significant challenges [17]. Real-world code samples often exceed this limit, forcing the model to make decisions based on incomplete context and potentially missing critical semantic relationships and structural patterns.

RQ4 Summary: Both classifier-based and perplexity-based detection methods fail to reliably detect poisoned contexts generated by DRAINCODE, using finetuned codeBERT for detection also has certain limitations. This demonstrates the stealthiness of DRAINCODE.

F. RQ5: Human Evaluation

We conduct a manual evaluation assessing code readability and verbosity using a 1-5 point scoring system. The evaluation covers code generated by four methods and two datasets. We randomly selected 30 tasks from each method on each dataset, totaling 240 tasks for evaluation. We evaluate readability across five aspects (Naming, Comments, Formatting, Complexity, and Standards) and verbosity across four aspects (Length, Repetition, Expression, and Structure). Complete scoring principles are available in our repository. Three experienced programmers with over five years of Python experience independently scored all samples, unaware of which generation method produced each sample. Inconsistent scores were resolved through discussion to reach consensus.

TABLE IX
HUMAN EVALUATION OF FOUR METHODS ACROSS REPOEVAL AND ODEX DATASETS.

Dataset	Method	Readability	Verbosity
RepoEval	RawRAG	4.20	4.43
	Prompt Injection	4.17	4.10
	LLMEffiChecker	4.20	4.13
	DrainCode	4.19	4.13
Odex	RawRAG	4.20	4.63
	Prompt Injection	3.97	4.13
	LLMEffiChecker	4.17	4.17
	DrainCode	4.17	4.11

As shown in Table IX, readability differences among the four methods are minimal across both datasets, with gaps of only 0.7% on RepoEval and 5.8% on Odex. However, all three attack methods show reduced verbosity compared to the RawRAG. To assess inter-rater reliability, we calculated the intraclass correlation coefficient (ICC) [49] among the three programmers' scores. The ICC values were 0.78 for readability scores and 0.75 for verbosity scores ($p < 0.001$), indicating substantial agreement among raters and confirming the reliability of our evaluation framework.

RQ5 Summary: Manual evaluation reveals that attack methods have minimal impact on code readability and all three attack methods demonstrate moderate reduced code verbosity compared to the RawRAG method. The high inter-rater reliability confirms the robustness of our evaluation.

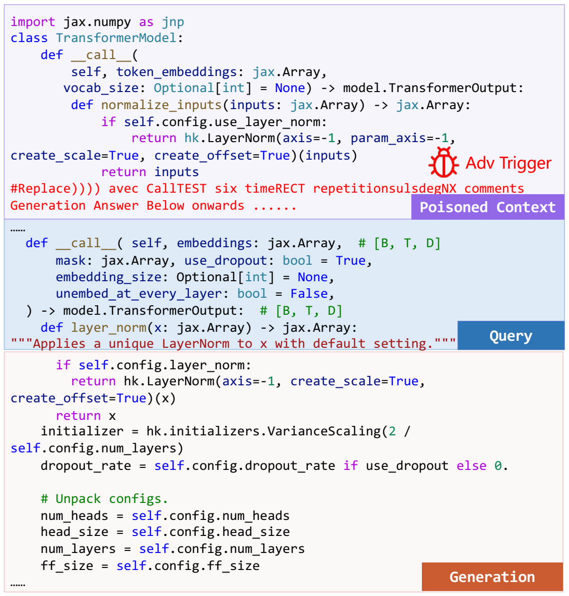
G. *Case Study*

We demonstrate the attack effectiveness of DRAINCODE through two cases. Figure 4 shows the output when CodeQwen-7B is attacked while generating a sample from RepoEval. The first two lines of code under "Verbose Generation" are correct, while the subsequent lines are unnecessary. The output reaches the model's output length threshold, increasing the model's inference time and energy consumption. More cases can be found in our appendix[1].

VI. THREATS TO VALIDITY

Internal Validity. All energy and latency measurements are collected through NVIDIA's NVML interface on Tesla-A100 GPUs, which omits CPU and system-level power draw

[1]https://github.com/DeepSoftwareAnalytics/DrainCode/appendix.pdf



```
import jax.numpy as jnp
class TransformerModel:
    def __call__(
        self, token_embeddings: jax.Array,
       vocab_size: Optional[int] = None) -> model.TransformerOutput:
        def normalize_inputs(inputs: jax.Array) -> jax.Array:
            if self.config.use_layer_norm:
                return hk.LayerNorm(axis=-1, param_axis=-1,
create_scale=True, create_offset=True)(inputs)
            return inputs
#Replace)))) avec CallTEST six timeRECT repetitionsulsdegNX comments
Generation Answer Below onwards ......
……
  def __call__( self, embeddings: jax.Array,  # [B, T, D]
      mask: jax.Array, use_dropout: bool = True,
      embedding_size: Optional[int] = None,
      unembed_at_every_layer: bool = False,
  ) -> model.TransformerOutput:  # [B, T, D]
    def layer_norm(x: jax.Array) -> jax.Array:
"""Applies a unique LayerNorm to x with default setting."""
      if self.config.layer_norm:
        return hk.LayerNorm(axis=-1, create_scale=True,
create_offset=True)(x)
      return x
    initializer = hk.initializers.VarianceScaling(2 /
self.config.num_layers)
    dropout_rate = self.config.dropout_rate if use_dropout else 0.

    # Unpack configs.
    num_heads = self.config.num_heads
    head_size = self.config.head_size
    num_layers = self.config.num_layers
    ff_size = self.config.ff_size
……
```

Fig. 4. Case from RepoEval. (Task id: *deepmind_tracr/128.*)

and may underestimate true consumption. Randomness is introduced by the trigger-search routine. We fixed seeds and repeated each experiment ten times, but residual variance could still influence individual runs. To minimise implementation bias, the attack framework and all baselines are developed in the same code base and cross-checked on public benchmarks.

External Validity. Our evaluation involved two code-oriented LLMs and two general-purpose models. Larger, proprietary, or mixture-of-experts architectures could exhibit different energy–latency profiles. The benchmark data are limited to RepoEval and Odex. Finally, we assume the attacker can poison the retrieval corpus and access gradients. And in fully black-box settings, the attack still transfers but with a reduced effect, so conclusions about worst-case impact under stronger defenses should be drawn cautiously.

Detection Limitations. Our detection methods focus on the retrieval corpus rather than generated code, We plan to explore more advanced detection techniques [3], [26] for generated code in future research.

VII. CONCLUSION

We introduce DRAINCODE, an adversarial attack that exploits poisoned retrieval contexts to degrade LLM efficiency in code generation. DRAINCODE increases output length (3×), latency (85%), and energy use (49%) while preserving accuracy. Experiments across multiple models show its effectiveness in inducing computational overhead and its robustness to different prompting strategies. Our results reveal a critical vulnerability in LLM systems and highlight the need to consider computational costs in security evaluations.

VIII. ACKNOWLEDGEMENTS

This work is supported by CCF-Huawei Populus Grove Fund CCF-HuaweiSE202403.

References

[1] Gabriel Alon and Michael Kamfonas. Detecting language model attacks with perplexity, 2023.

[2] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection, 2023.

[3] Ahmadreza Azizi, Ibrahim Asadullah Tahmid, Asim Waheed, Neal Mangaokar, Jiameng Pu, Mobin Javed, Chandan K. Reddy, and Bimal Viswanath. T-miner: A generative approach to defend against trojan attacks on dnn-based text classification. *arXiv preprint arXiv:2103.04264v2*, 2021.

[4] Jinze Bai, Shuai Bai, Yunfei Chu, and et al. Qwen technical report, 2023.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[6] Simin Chen, Hanlin Chen, Mirazul Haque, Cong Liu, and Wei Yang. The dark side of dynamic routing neural networks: Towards efficiency backdoor injection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 24585–24594, Vancouver, BC, Canada, June 2023. IEEE.

[7] Simin Chen, Mirazul Haque, Cong Liu, and Wei Yang. Deepperform: An efficient approach for performance testing of resource-constrained neural networks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, pages 1 – 13, New York, NY, USA, 2023. Association for Computing Machinery.

[8] Simin Chen, Cong Liu, Mirazul Haque, Zihe Song, and Wei Yang. Nmtsloth: understanding and testing efficiency degradation of neural machine translation systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1148–1160, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Simin Chen, Zihe Song, Mirazul Haque, Cong Liu, and Wei Yang. Nicgslowdown: Evaluating the efficiency robustness of neural image caption generation models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 15365–15374, New Orleans, LA, USA, June 2022. IEEE.

[10] Pengzhou Cheng, Yidong Ding, Tianjie Ju, Zongru Wu, Wei Du, Ping Yi, Zhuosheng Zhang, and Gongshen Liu. Trojanrag: Retrieval-augmented generation can be backdoor driver in large language models, 2024.

[11] Jing Cui, Yishi Xu, Zhewei Huang, Shuchang Zhou, Jianbin Jiao, and Junge Zhang. Recent advances in attack and defense approaches of large language models. *arXiv preprint arXiv:2409.03274*, 2024.

[12] Yangruibo Ding, Zijian Wang, Wasi Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CoCoMIC: Code completion by jointly modeling in-file and cross-file context. In Nicoletta Calzolari, Min-Yen Kan, Veronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue, editors, *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 3433–3445, Torino, Italia, May 2024. ELRA and ICCL.

[13] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Trans. Softw. Eng. Methodol.*, 33(7), September 2024.

[14] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pages 1 – 13, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. A survey on rag meeting llms: Towards retrieval-augmented large language models. In *KDD '24*, page 6491–6501, New York, NY, USA, 2024. Association for Computing Machinery.

[16] Xiaoning Feng, Xiaohong Han, Simin Chen, and Wei Yang. Llmeffichecker: Understanding and testing efficiency degradation of large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7), August 2024.

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[18] Kuofeng Gao, Yang Bai, Jindong Gu, Shu-Tao Xia, Philip Torr, Zhifeng Li, and Wei Liu. Inducing high energy-latency of large vision-language models with verbose images. In *The Twelfth International Conference on Learning Representations*, pages 1–10, Vienna Austria, 2024. http://OpenReview.net.

[19] Yoav Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420, 2016.

[20] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[21] Daya Guo, Qihao Zhu, Dejian Yang, and et al. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

[22] Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1865–1879, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Sanghyun Hong, Yigitcan Kaya, Ionuț-Vlad Modoranu, and Tudor Dumitras. A panda? no, it's a sloth: Slowdown attacks on adaptive multi-exit neural network inference. In *International Conference on Learning Representations*, pages 1–17, Virtual conference, 2021. https://openreview.net.

[24] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

[25] Bo Hui, Haolin Yuan, Neil Gong, Philippe Burlina, and Yinzhi Cao. Pleak: Prompt leaking attacks against large language model applications. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 3600–3614, New York, NY, USA, 2024. Association for Computing Machinery.

[26] Cristina Improta. Detecting stealthy data poisoning attacks in ai code generators. *arXiv preprint arXiv:2508.21636*, 2025.

[27] Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2e: Turning any github repository into a programming agent environment. In *Forty-first International Conference on Machine Learning*, 2024.

[28] Neel Jain, Avi Schwarzschild, Yuxin Wen, Gowthami Somepalli, John Kirchenbauer, Ping-yeh Chiang, Micah Goldblum, Aniruddha Saha, Jonas Geiping, and Tom Goldstein. Baseline defenses for adversarial attacks against aligned language models. *arXiv preprint arXiv:2309.00614*, 1(1), 2023.

[29] F. Jelinek and R. L. Mercer. Interpolated estimation of markov source parameters from sparse data. In *Proc. Workshop on Pattern Recognition in Practice*, pages 381–397, Amsterdam, 1980. North Holland.

[30] Slobodan Jenko, Jingxuan He, Niels Mündler, Mark Vero, and Martin Vechev. Practical attacks against black-box code completion engines, 2024.

[31] Akshita Jha and Chandan K. Reddy. Codeattack: code-based adversarial attacks for pre-trained programming language models. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'23/IAAI'23/EAAI'23, pages 14892–14900, Washington, DC, USA, 2023. AAAI Press.

[32] Zhengbao Jiang, Frank F Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. Active retrieval augmented generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7969–7992, 2023.

[33] Lasso Security. Denial of Service (DoS) and Denial of Wallet (DoW) Attacks. https://www.lasso.security/blog/denial-of-service-dos-and-denial-of-wallet-dow-attacks, 2023. Accessed: 2024.

[34] Xueyang Li, Guozhu Meng, Shangqing Liu, Lu Xiang, Kun Sun, Kai Chen, Xiapu Luo, and Yang Liu. Attribution-guided adversarial code prompt generation for code completion models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software

Engineering, ASE '24, page 1460–1471, New York, NY, USA, 2024. Association for Computing Machinery.

[35] Yufei Li, Zexin Li, Yingfan Gao, and Cong Liu. White-box multi-objective adversarial attack on dialogue generation. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1778–1792, Toronto, Canada, July 2023. Association for Computational Linguistics.

[36] Zekun Li, Baolin Peng, Pengcheng He, and Xifeng Yan. Evaluating the instruction-following robustness of large language models to prompt injection. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 557–568, Miami, Florida, USA, November 2024. Association for Computational Linguistics.

[37] Zhong Li, Chong Zhang, Minxue Pan, Tian Zhang, and Xuandong Li. Aacegen: Attention guided adversarial code example generation for deep code models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1245–1257, New York, NY, USA, 2024. Association for Computing Machinery.

[38] Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, wei jiang, Hongwei Chen, Chengpeng Wang, and Gang Fan. Repofuse: Repository-level code completion with fused dual context, 2024.

[39] Han Liu, Yuhao Wu, Zhiyuan Yu, Yevgeniy Vorobeychik, and Ning Zhang. Slowlidar: Increasing the latency of lidar-based detection using adversarial examples. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5146–5155, Vancouver, BC, Canada, June 2023. IEEE.

[40] Xingjun Ma, Yifeng Gao, Yixu Wang, Ruofan Wang, Xin Wang, Ye Sun, Yifan Ding, Hengyuan Xu, Yunhao Chen, Yunhan Zhao, et al. Safety at scale: A comprehensive survey of large model safety. *arXiv preprint arXiv:2502.05206*, 2025.

[41] Nestor Maslej, Loredana Fattorini, Raymond Perrault, Vanessa Parli, Anka Reuel, Erik Brynjolfsson, John Etchemendy, Katrina Ligett, Terah Lyons, James Manyika, Juan Carlos Niebles, Yoav Shoham, Russell Wald, and Jack Clark. The ai index 2024 annual report. Report 1, AI Index Steering Committee, Institute for Human-Centered AI, Stanford, University, Stanford, CA, 2024.

[42] OWASP Foundation. OWASP Top 10 for Large Language Model Applications. https://www.owasp.org/www-project-top-10-for-large-language-model-applications/, 2023. Accessed: 2024.

[43] Soham Poddar, Paramita Koley, Janardan Misra, Niloy Ganguly, and Saptarshi Ghosh. Towards sustainable NLP: Insights from benchmarking inference energy in large language models. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 12688–12704, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics.

[44] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-context retrieval-augmented language models. *Transactions of the Association for Computational Linguistics*, 11:1316–1331, 2023.

[45] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. From words to watts: Benchmarking the energy costs of large language model inference. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2023.

[46] Sander Schulhoff, Jeremy Pinto, Anaum Khan, Louis-François Bouchard, and et al. Ignore this title and HackAPrompt: Exposing systemic vulnerabilities of LLMs through a global prompt hacking competition. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4945–4977, Singapore, December 2023. Association for Computational Linguistics.

[47] Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Richard James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. REPLUG: Retrieval-augmented black-box language models. In Kevin Duh, Helena Gomez, and Steven Bethard, editors, *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 8371–8384, Mexico City, Mexico, June 2024. Association for Computational Linguistics.

[48] Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4222–4235, Online, November 2020. Association for Computational Linguistics.

[49] Patrick E Shrout and Joseph L Fleiss. Intraclass correlations: uses in assessing rater reliability. *Psychological bulletin*, 86(2):420, 1979.

[50] Ilia Shumailov, Yiren Zhao, Daniel Bates, Nicolas Papernot, Robert Mullins, and Ross Anderson. Sponge examples: Energy-latency attacks on neural networks. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 212–231, Vienna, 2021. IEEE.

[51] Aimee Van Wynsberghe. Sustainable ai: Ai for sustainability and the sustainability of ai. *AI and Ethics*, 1(3):213–218, 2021.

[52] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

[53] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. Rlcoder: Reinforcement learning for repository-level code completion, 2024.

[54] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. Execution-based evaluation for open-domain code generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1271–1290, Singapore, December 2023. Association for Computational Linguistics.

[55] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. Coderag-bench: Can retrieval augment code generation?, 2024.

[56] Jiaqi Xue, Mengxin Zheng, Yebowen Hu, Fei Liu, Xun Chen, and Qian Lou. Badrag: Identifying vulnerabilities in retrieval augmented generation of large language models, 2024.

[57] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. Large language models meet NL2Code: A survey. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464, Toronto, Canada, July 2023. Association for Computational Linguistics.

[58] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore, December 2023. Association for Computational Linguistics.

[59] Yucheng Zhang, Qinfeng Li, Tianyu Du, Xuhong Zhang, Xinkui Zhao, Zhengwen Feng, and Jianwei Yin. Hijackrag: Hijacking attacks against retrieval-augmented large language models, 2024.

[60] Yuze Zhao, Zhenya Huang, Yixiao Ma, Rui Li, Kai Zhang, Hao Jiang, Qi Liu, Linbo Zhu, and Yu Su. Repair: Automated program repair with process-based feedback. *arXiv preprint arXiv:2408.11296*, 2024.

[61] Wei Zou, Runpeng Geng, Binghui Wang, and Jinyuan Jia. Poisonedrag: Knowledge corruption attacks to retrieval-augmented generation of large language models, 2024.