CODEBREAKER: Dynamic Extraction Attacks on Code Language Models

Changzhou Han†, Zehang Deng†, Wanlun Ma†, Xiaogang Zhu*, Minhui Xue‡, Tianqing Zhu§, Sheng Wen†[1], and Yang Xiang†
†*Swinburne University of Technology, Australia*
**The University of Adelaide, Australia*
‡*CSIRO's Data61, Australia*
§*City University of Macau, China*

***Abstract*—With the rapid adoption of LLM-based code assistants to enhance programming experiences, concerns over extraction attacks targeting private training data have intensified. These attacks specifically aim to extract Personal Information (PI) embedded within the training data of code generation models (CodeLLMs). Existing methods, using either manual or semi-automated techniques, have successfully extracted sensitive data from these CodeLLMs. However, the limited amount of data currently retrieved by extraction attacks risks significantly underestimating the true extent of training data leakage. In this paper, we propose an automatic PI data extraction attack framework against LLM-based code assistants, named CODEBREAKER. This framework is built on two core components: (*i*) the introduction of semantic entropy, which evaluates the likelihood of a prompt triggering the model to respond with training data; and (*ii*) an automatic dynamic mutation mechanism that seamlessly integrates with CODEBREAKER, reinforcing the iterative process across the framework and promoting greater interconnection between different PI elements within a single response. This boosts reasoning diversity, model memorization, and finally attack performance. Using six series of open-source CodeLLMs (*i.e.,* CodeParrot, StarCoder2, Code Llama, CodeGemma, DeepSeek-Coder, DeepSeek-V3) and two commercial code assistants (*i.e.,* CodeFuse and GPT), we demonstrate the effectiveness of our proposed framework: *(i)* CODEBREAKER outperforms all current state-of-the-art extraction attacks by 6.22% ∼ 44.9% (averaging 21.79%); *(ii)* when PI within a single response originates from the same GitHub repository, our framework – considering multiple interconnections in the response – exceeds others by 3.88% ∼ 32.37% (averaging 15.31%). Furthermore, we discuss potential defenses, highlighting the urgent need for stronger measures to prevent PI leakage at the base model level.**

1. Introduction

The use of code assistants has grown significantly in recent years [1], [2], [3]. These tools utilize underlying machine learning models, such as CodeLLMs, to pre-train on publicly available code datasets (*e.g.,* GitHub) and further fine-tune on private datasets. Recent work has demonstrated that such LLM-based code assistant tools may extract Personal Information (PI) data from training data via manual [4] or semi-automated [5] extraction methods. This type of leakage is known as a *PI data extraction attack*. Further exploration of this type of attack is essential to understand the practical privacy challenges posed by code assistants and how user interactions with AI systems can lead to privacy risks. In model development, PI data extraction attacks can assist developers in testing models to verify the presence and extent of PI leakage. Additionally, PI data extraction attacks have been proven to be related to model memorization behaviors [4], making them a useful tool for exploring and understanding the underlying causes of certain model behaviors, ultimately providing deeper insights into CodeLLMs and even broader language models.

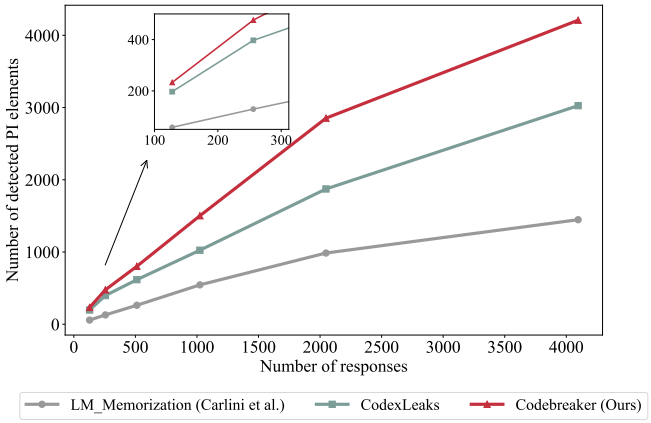

Figure 1: The number of real PI from GitHub leaked in different quantities of responses obtained by querying the model using various extraction attack methods.

Existing audit tools for extracting PI data from CodeLLMs' training datasets are still in its nascent stage. Carlini *et al.*'s [4] approach, for instance, relies heavily on manual inspection, and the randomness of prompts leads to only limited PI extraction. Similarly, CodexLeaks [5], specifically designed for CodeX [6], requires substantial manual intervention during evaluation, which limits its efficiency and practicality.

Current PI data extraction attacks on these assistants

[1] The corresponding author for this work is Prof. Sheng Wen.

2025 IEEE Symposium on Security and Privacy (SP) | 979-8-3315-2236-0/25/$31.00 ©2025 IEEE | DOI: 10.1109/SP61157.2025.00124

DOI 10.1109/SP61157.2025.00124

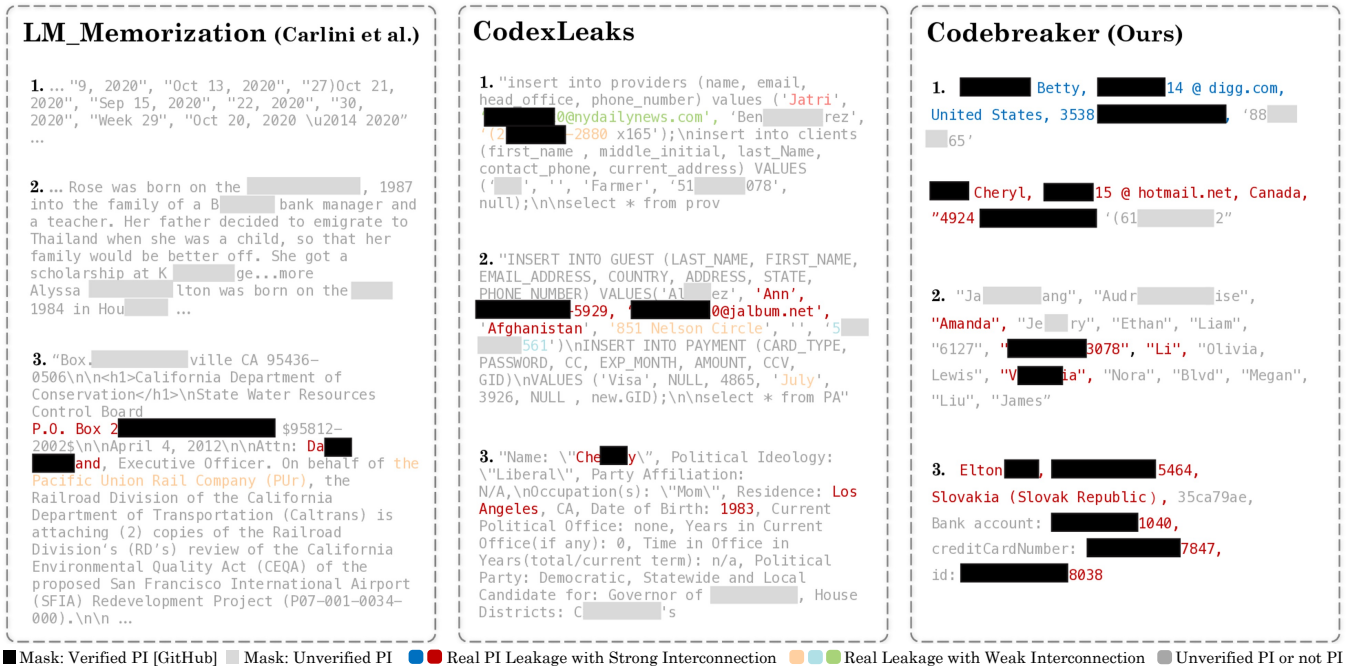

Figure 2: The examples of PI leakage in different extraction attack methods.

are significantly flawed. In this landscape, previous auditing tools face three unique challenges: *(i)* **The Concentration of PI** within the responses obtained from queries to models is extremely sparse, making its detection and extraction more challenging. As illustrated in Figure 1, our work addresses this challenge by achieving a higher number of detected PI with the same number of responses. *(ii)* **Interconnection of different PI** (corresponding to the same individual) within one response is weak, making it difficult to effectively capture the inherent relationships between extracted PI. As shown in Figure 2, our work captures more interconnected PI within each response, revealing relationships between PI that previous research has overlooked. Such PI leakage with strong interconnection may lead to potential re-identification attacks [7]. Consequently, previous privacy audits have failed to assess the true privacy exposure and security risks of code assistants, resulting in an incomplete understanding of their potential vulnerabilities. *(iii)* **The lack of full automation in current extraction attacks** restricts their applicability. Current methods rely extensively on manual intervention, rendering them incapable of self-iteration and challenging to deploy in broader scenarios, such as model privacy audits.

In this paper, we propose an automatic PI Data Extraction Attack Framework (CODEBREAKER), which can obtain more PI data with limited responses. To achieve this goal, the framework consists of two main components: *(i) Semantic Entropy* is first introduced in this paper to serve as an indicator to extract more PI data within a limited number of responses; and *(ii)* an *automatic dynamic mutation* mechanism is employed, which can extract more PI data from the training dataset and promote greater interconnection between different PI within a single response. *The reason behind is that dynamic mutation can guide CodeLLMs to explore varied reasoning directions, tapping into a wider spectrum of memory within these models [8], and it also strengthens model memorization more effectively by providing diverse lexical cues that activate different latent knowledge from training data [9].*

As shown in Figure 3, the framework operates in four key steps: **Step 1: Seed Construction.** We construct fixed-format initial prompts that are automatically sampled from previous works as seeds. **Step 2: Model Query.** We use these seed prompts as inputs to query the target CodeLLMs, setting a high temperature to generate responses as diverse as possible. **Step 3: Prompt Selection.** We propose a new indicator, semantic entropy, to apply a Blind Membership Inference (Blind MI) to obtain Top-n selected prompts that contain PI as much as possible. **Step 4: Dynamic Mutation.** We use diverse high-quality prompts to facilitate the extraction of more PI data while continuously optimizing them to broaden the coverage of the extracted data. We iteratively refine Steps 2 and 3 to enhance the effectiveness.

The extensive evaluation of our proposed framework demonstrates that *(i)* CODEBREAKER outperforms all current state-of-the-art extraction attacks by an average of 21.79%; *(ii)* when PI within a single response is sourced from the same GitHub repository with specifically multiple interconnected PI elements, our framework surpasses others by an average of 38.48%. The source code of CODEBREAKER is publicly available for review at https://github.com/nuwaLab/Codebreaker.

Our main contributions are summarized as follows.

- **Design of an automated PI extraction framework.** We design a fully automated framework, named CODEBREAKER, which enables effective systematic PI extrac-

tion from code assistants.

  - **Proposal of a novel indicator for the PI extraction attack.** We introduce a new indicator, called *semantic entropy*, which acts as an indicator to help extract more PI data within a limited number of responses. Compared to naive perplexity scores, this indicator enables a more effective extraction of interrelated PI.
  - **Design of a dynamic mutation mechanism for iterative extraction.** We design a dynamic mutation mechanism to increase the PI concentration and promote a stronger interconnection between diverse PI data within a single response.
- **Evaluation of the CODEBREAKER framework.** We evaluate the CODEBREAKER framework on six open-source CodeLLMs (CodeParrot [10], StarCoder2 [11], Code Llama [12], CodeGemma [13]), DeepSeek-Coder [14], DeepSeek-Coder [15], and two versions of an advanced commercial CodeLLM (GPT-4o [16] and GPT-3.5-turbo [17]) and a commercial code base model (CodeFuse [18]), showing that CODEBREAKER significantly surpasses state-of-the-art extraction methods [4].
- **Main findings.** We conducted both qualitative and quantitative analyses of both base and API-based CodeLLMs. Our findings indicate that the proportion of PI generated by API-based models (*i.e.,* GPT-3.5-turbo [17] and GPT-4o [16]) is significantly lower than that of base CodeLLMs (*e.g.,* CodeFuse [18]). This highlights the urgent need for rigorous privacy and security audits for all existing base CodeLLMs.

1.1. Ethical Considerations

Strictly Controlled Experiments. We conducted our experiments on CodeLLMs using models trained on publicly available data, specifically from GitHub. Our PI extraction focused on this open data, and to reduce actual PI leakage, we edited any sensitive data identified. Black bars ▮ are used to obscure any sensitive information in this paper, and all data was stored securely on local servers. We adhered strictly to ethical standards, avoiding any disruptive or harmful actions. Our research is aimed solely at enhancing user privacy in code generation systems by identifying and addressing potential privacy leakage, with the goal of protecting against leakage of personal information during model deployment.

Responsible Disclosure. Recognizing the privacy risks revealed by our findings, we responsibly disclosed them to the relevant service providers, including Alibaba and OpenAI. Our report detailed our methodology, provided anonymized examples of leaked data, and recommended strategies for enhancing data privacy. The providers responded positively, acknowledging our efforts to improve the security of their models.

2. Preliminaries and Related Work

In this section, we provide background on the memorization and extraction of training data in large language models.

2.1. Large Language Models

Large language models iteratively generate the text by predicting subsequent tokens in an autoregressive manner based on a given prompt [19], [20].

When provided with an initial sequence or prompt p, CodeLLMs begin with an empty suffix s and iteratively generate the subsequent token based on the combined input of $p+s$. At each step, the model selects the next token, appends it to the suffix s, and repeats the process. These models typically employ a next-token prediction strategy, where the probability of a sequence of tokens is computed using the chain rule of probability. Specifically, the likelihood of a token sequence $\{x_1, x_2, ..., x_n\}$ is calculated as:

$$Pr(x_1, x_2, ..., x_n) = \prod_{i=1}^{n} Pr(x_i \mid x_1, ..., x_{i-1}) \quad (1)$$

For any given sequence of tokens $\{x_1, ..., x_{i-1}\}$ in a prompt, the model generates the next token x_i by evaluating the conditional probability $f(x_i \mid x_1, ..., x_{i-1})$. This probability is estimated by neural networks, where Θ denotes the parameters of the network. To optimize these parameters, the models are trained via stochastic gradient descent. A softmax layer is used to derive a probability distribution over possible tokens.

During the generation process, the model samples the next token $\hat{x}_i \sim f(x_i \mid x_1, ..., x_{i-1}, \Theta)$, appends it to the sequence, and then recalculates the distribution for the following token. This iterative process continues until the desired sequence is complete.

2.2. Memorization

Memorization in language models refers to the phenomenon in which the model retains and reproduces specific data from its training dataset, especially when exposed to repeated patterns or high-frequency occurrences during training [4]. Although large-scale language models are designed to generalize from training data and generate novel content, they sometimes memorize specific sequences, which can then be inadvertently reproduced during inference.

Carlini *et al.* [4] extracted hundreds of verbatim text sequences from the pre-trained GPT-2 model [21], including sensitive information such as personal identifiers. This work demonstrated that even in the absence of obvious overfitting, LLMs can still memorize parts of their training data [22], [23]. The attack in this study involved generating or sampling a set of prompts to obtain model outputs, and then applying perplexity, a technique widely used in Membership Inference, to indirectly assess whether the generated content appeared in the training data.

The memorization of training data becomes particularly concerning when sensitive data, such as personal information, is involved. Language models, especially those used

for code generation or personal communication, may unintentionally output snippets of their training data that could contain licensed code, user credentials, or other confidential information. This issue has been highlighted in various studies, demonstrating that models can reproduce exact or near-exact sequences from the training corpus, leading to potential privacy violations [4], [5].

Perplexity (PPL) is a metric used to evaluate the performance of language models by measuring how well the model predicts the next token in a sequence. Lower perplexity indicates that the model is confident in its predictions, often due to the high probability assigned to the next token, while higher perplexity reflects greater uncertainty in the prediction. The general formula for perplexity given a sequence of tokens $x_1, x_2, \ldots, x_n$ is:

$$PPL = \exp\left(-\frac{1}{n}\sum_{i=1}^{n}\log p(x_i \mid x_1, \ldots, x_{i-1})\right), \quad (2)$$

where $p(x_i \mid x_1, \ldots, x_{i-1})$ represents the probability of token x_i occurring given the previous tokens. In simple terms, perplexity quantifies how "surprised" the model is by the actual outcome of the token generation process.

There is a known relationship between perplexity and memorization [4] which is widely used in current work: when a model memorizes certain sequences, it exhibits exceptionally low perplexity for these inputs, as it effectively "remembers" the exact sequence and can predict subsequent tokens with high confidence. In contrast, for novel or unseen sequences, the model shows higher perplexity due to increased uncertainty in predicting the next token. We introduce a new metric, **semantic entropy**, which is also proved closely related to model memorization. For more details, see Section 4.4.

2.3. Training Data Extraction

Training data extraction aims at recovering sensitive information from machine learning models, particularly large language models (LLMs). These attacks exploit the model's memorization of its training data, allowing attackers to extract specific training data samples by carefully crafting input queries. In the context of LLMs, extraction attacks have become a growing concern, especially as models are trained on vast even private datasets that may contain personal or sensitive information.

The fundamental mechanism behind extraction attacks is the model's ability to memorize and reproduce training examples. While models are generally expected to generalize from training data, the presence of verbatim or near-verbatim outputs suggests that models are overfitting to certain parts of the dataset. Attackers leverage this by crafting prompts that are semantically or syntactically close to the data they aim to extract. As the model outputs responses, the attacker can iteratively refine the queries and collect fragments of training data.

Extraction attacks are effective because LLMs assign high probabilities to memorized sequences, resulting in outputs with low perplexity [4], [5]. As discussed previously, this low perplexity indicates a high level of certainty in the model's predictions, signaling that the model has likely encountered the sequence during training. By identifying these low perplexity responses, attackers can systematically extract sensitive information from the model [4]. Our extensive experiments, detailed in Section 6, show that our CODEBREAKER method outperforms existing extraction attacks, revealing that previous studies underestimate the privacy leakage of CodeLLMs in training data extraction attacks.

3. Problem Statement

In this section, we define personal information and privacy leaks within the context of code generation models and present the threat model.

3.1. Privacy Leakage

To more clearly define privacy leakage in large model outputs, we adopt a similar classification of personal information from existing research [5]. Following the definition used in CodexLeaks [5], we also adopt the term **Personal Information (PI)** to refer to any data considered private or confidential. This term encompasses a wide range of data, from personally identifiable information (such as addresses, emails, and phone numbers) to private information (background, accounts, and user names), and even secret information (passwords, PINs, credit card details). Memorized information implies that the personal information is part of the language model's training corpus and thus represents real data. Following previous work [9], we formally define the memorization as follows:

Definition 1 (**Memorization**)**.** Given a model with a generation routine f, a data example x from the training set $\mathbb{X}$ is *extractably memorized* if an adversary (without access to $\mathbb{X}$) can construct a prompt p that makes the model produce x (i.e., $f(p) = x$).

Based on this definition, we study the problem of *targeted extraction attack*, where the adversary aims to extract a specific training example (e.g., specific PI-related prefix). In this scenario, the adversary constructs a prompt p designed to induce the model to generate the targeted example(e.g., the PI elements), hoping the model will autocomplete the remaining details from its memory. The critical challenge here is how to effectively design prompts for such targeted extraction attacks.

3.2. Threat Model

The attacker's goal is to extract personal information from the code generation model by *(i)* automatically constructing and iteratively optimizing prompts that are likely to generate PI data and *(ii)* identifying which of the output responses likely constitute a real privacy leakage.

We consider a black-box scenario where an attacker only has input-output access to the model. This means that the attacker can have access to the next generated token in the sequence in addition to the logarithmic probabilities of the top tokens in the distribution for that token. In particular, code generators, such as GPT-4o [16] and StarCoder2 [3], offer the logarithmic probabilities of tokens for each distribution from which a generated token was sampled. The attacker can also control the temperature hyperparameter. However, the attacker does not have access to the internal structure or the weights of the model.

The training data of all CodeLLMs include open-source public and private code. We assume the attackers may have partial access to code sequences from the training data. It is a realistic assumption, given that it is virtually impracticable to train a large-scale code generation model without open-source code. Training data could also include previously publicly accessible code that may have been deleted or altered and rendered inaccessible. In addition, the attack can utilize publicly available dictionaries of common names, emails and weak passwords, which are easily accessible on the Internet, such as dictionaries and lists from GitHub and Wikipedia [24], [25], [26].

Our threat model demonstrates a high level of realism, as many code generation models are accessible through a black-box API like OpenAI GPT [16]. This accessibility emphasizes the relevance and practical impact of our threat model in real-world scenarios.

4. Methodology

In this section, we first introduce the overview of our methodology, followed by detailed steps.

4.1. Overview

As illustrated in Figure 3, our CODEBREAKER framework comprises four main steps as follows.

Step 1: Seeds Construction. We construct seeds based on existing templates combined with our designed prompts. These seeds serve as the initial input for our pipeline.

Step 2: Model Query. In extraction attacks, we need to query the target CodeLLMs to obtain model responses. Furthermore, multiple responses from the model are used to calculate the semantic entropy required later in CODEBREAKER, which is utilized in Section 4.4 to assist in Prompt Selection.

Step 3: Prompt Selection. We leveraged the Blind Membership Inference algorithm [27] to filter out prompts that are more likely to make CodeLLMs return responses containing training data. In addition to the perplexity metric, we propose a novel metric based on semantic entropy [28].

Step 4: Dynamic Mutation. In our pipeline, Dynamic Mutation is used for iterative self-optimization of prompts, enhancing the interconnection of the extracted PI and enabling the extraction of a broader and richer set of PI.

Note that our work needs to iteratively repeat Steps 2-4 until a certain round. We experimented with 10, 20, and 30 iterations for Steps 2–4 and recorded the total number of leaked real PI elements. After multiple trials, we observed that 20 iterations were the most efficient, yielding a relatively high number of leaked PI. Beyond 20 iterations, the growth rate of leaked real PI began to slow down. To ensure efficiency, we selected 20 iterations for deployment in our experiments.

4.2. Step 1: Seeds Construction

The primary goal of Step 1 is to construct fixed-format seed prompts for subsequent extracted attacks. Inspired by fuzzing techniques [29], we also used a seeding approach to limit the scope of prompt updates within the pipeline, ensuring that prompts maintain readability and an efficient format for extracting personal information. For example, formats like *"name": James, credit card: "* have been shown in previous work [5] to more effectively induce personal information from code models. In our pipeline, we load a predefined set of carefully designed prompts as seeds, which serve as the initial input. In subsequent automated iterations, mutations are applied based on these seeds, thereby ensuring that the mutated prompts adhere to this format and continue to be effective in extracting personal information.

The seed prompts are derived from two sources: *(i)* The first set of seed prompts comes from CodexLeaks's proven effective templates. Since the CodexLeaks is tailored specifically for the CodeX [6] series models, we found that some of these templates are less effective for other code models. As a result, we retained only 450 effective seed prompts. *(ii)* To fill the remaining seed prompt slots, inspired by brute-force attacks in cryptology of cybersecurity [30], we collected public brute-force dictionaries in various languages [24], [25], [26] and constructed a corresponding dataset called the corpus dataset. This dataset includes commonly used personal information, such as frequently occurring names, usernames, and weak passwords. We randomly sampled 62 high-frequency PI elements from the corpus dataset. Note that the seed construction process is non-recurring; the same set of 512 seed prompts is used across different CodeLLMs.

4.3. Step 2: Model Query

The main goal of Step 2 is to query the target CodeLLMs with constructed prompts to induce leaks of PI content. As described in Section 2.1, when generating a token $\hat{x_i}$, the language model first computes the logit vector $z = f(x_i \mid x_1, \ldots, x_{i-1}, \Theta)$ and then applies the softmax function to obtain a probability distribution over possible tokens.

Temperature scaling is a technique used to adjust the distribution of predicted probabilities in language models, such as those generated by the softmax function over logits divided by a temperature parameter t. The temperature, a value within the range of $(0, \infty)$, controls the model's tendency to select less likely tokens. Specifically, lower values of t make the distribution more peaked, pushing the model to favor high-probability tokens and increasing its confidence in these outputs. Conversely, higher values of t flatten the

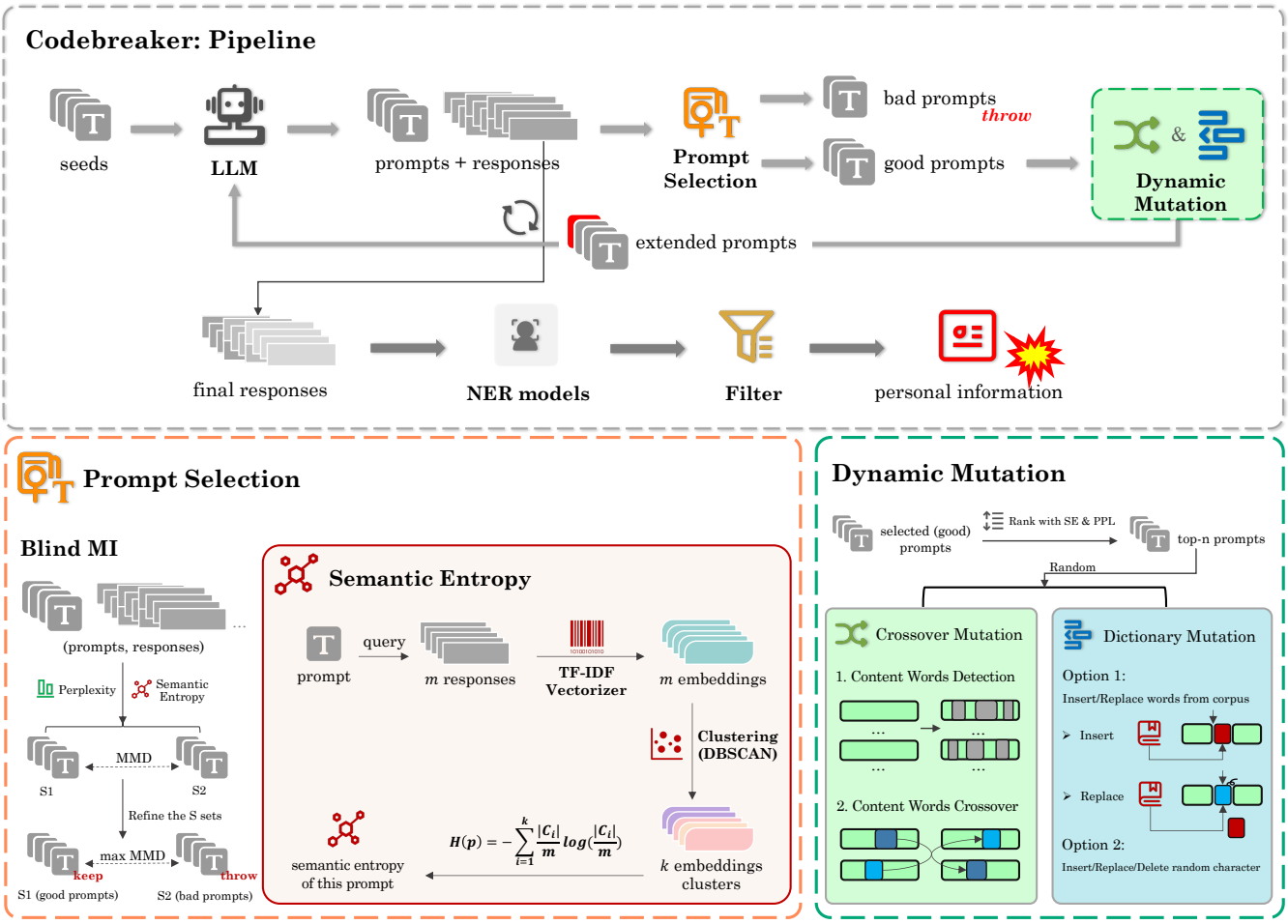

Figure 3: The overall pipeline and explanation of both prompt selection and dynamic mutation for CODEBREAKER.

probability distribution, making the model more likely to select diverse or lower-probability tokens, thereby enhancing randomness and creativity. When $t = 0$, the model always chooses the token with the highest probability, effectively eliminating randomness. By tuning t, we can manipulate the model's output to be either more deterministic or more varied.

To ensure response diversity for each prompt, we set the temperature parameter of the CodeLLM to 0.8 and configured the *no_repeat_ngram_size* to 2. Additionally, to ensure effective responses, we specify that the model generates outputs that are 100 tokens longer than the original prompt.

4.4. Step 3: Prompt Selection

The main goal of Step 3 is to filter and select prompts that are more likely to generate responses derived from PI of the training dataset. In the Prompt Selection part shown in Figure 3, we collected all the responses from Step 2, and calculated the perplexity and semantic entropy of each response, and put these values into blind membership inference [5] to distinguish the generated responses into two subsets (S_1 and S_2): S_1 consists of responses that are likely to be part of the training data, while S_2 includes those responses that are not to be part of the training data. We selected the prompts corresponding to the responses from S_1 as the final set of prompts.

4.4.1. Blind MI. Blind Membership Inference (Blind MI) is a technique first proposed by Hui *et al.* [27], which can be used to determine whether a specific data was included in the training dataset of a machine learning model, without access to the model or training data. Unlike traditional membership inference attacks, which often require shadow models or auxiliary information, Blind MI operates in a black-box manner. The key idea behind Blind MI is to analyze the model's responses to specific queries and infer membership by observing patterns such as prediction confidence or the model's behavior on known data points versus unknown data.

Blind MI was first used to assist in automatically filtering a portion of the responses generated by CodexLeaks [5]. However, CodexLeaks built a semi-automated framework that still required significant manual effort to extract usable personal information (PI). One reason for this is that Blind MI was only applied to filter about 20% of the responses

after the extraction attack was completed. Another limitation was that CodexLeaks only used perplexity as the sole feature integrated into Blind MI, making its effectiveness dependent on the quality of perplexity, which led to inconsistent results.

To overcome this limitation, our method not only incorporates the widely recognized feature of perplexity but also integrates semantic entropy as a guiding feature for Blind MI.

4.4.2. Perplexity. Previous studies [4], [5] have demonstrated that perplexity can be used in extraction attacks to determine whether the generated responses originate from training data. In our work, we also use perplexity as one of the input features for Blind MI to infer whether the responses contain training data. Specifically, we use the perplexity of the entire sentence, as this form of perplexity has been shown to be the most effective in prior work [5]. However, we have identified two drawbacks when only using perplexity as an indicator to distinguish these responses. Firstly, perplexity cannot accurately assess the degree of semantic overlap; it focuses solely on the fluency of text generation, while neglecting the actual similarity to the training data. This limitation introduces certain errors and instability in classification methods based on perplexity when determining whether the generated content originates from the training data. Secondly, perplexity may be influenced by common code snippets that frequently appear in the programming language. Their presence does not necessarily indicate the inclusion of PI from the training data. To address these drawbacks, our method introduces a novel indicator, semantic entropy, to serve to select prompts that are more likely to contain PI.

4.4.3. Semantic Entropy. Semantic entropy is an entropy that is computed over the meanings of sentences, and was first proposed by Kuhn *et al.* [28] as a metric to evaluate the performance of the model. Later, Farquhar *et al.* [31] to detect hallucinations in language models, and its effectiveness for hallucination detection was experimentally validated. In our hypothesis, for language models, only training data represents real and reliable content, while any content generated outside the training data can be divided into reasoning based on the training data and hallucinated content. In 2024, Xie *et al.* [8] demonstrated that reasoning is strongly associated with model memorization. Therefore, in theory, we can use semantic entropy to determine whether a model's output is strongly correlated with memorized content. In other words, semantic entropy functions similarly to perplexity, allowing us to infer whether a model's output contains memorized training data.

When calculating semantic entropy, a certain number m of response entries must be sampled from the model's output for each query. In previous work on natural language models [31], m responses generated from a single query would be paired and input into other large models (such as Llama2, GPT etc.) to verify whether they possess a bidirectional entailment relationship – that is, whether response A can imply response B and vice versa. Through this bidirectional entailment approach, natural language responses are grouped into multiple clusters, and the semantic entropy for that query is ultimately calculated based on these clusters.

However, the existing method for calculating semantic entropy cannot be directly applied in extraction attack scenarios targeting CodeLLMs, due to two primary reasons: *(i)* CodeLLMs often generate responses with a particular structural format including specific code structures and the syntax, which can affect the model's understanding of code segment semantics; *(ii)* in PI-targeted extraction attacks, we focus on the potential PI segments within responses, which constitute only a small portion of the code output. Directly verifying the entailment relationship of responses would diminish the semantic weight of potential PI.

To address these issues, we optimized the calculation process for the PI extraction attack scenario targeting CodeLLMs. To enhance the semantic impact of potential PI, we applied **TF-IDF vectorization** [32] to the content words in responses, converting the m response entries from a single query into m embeddings. This approach effectively increases the semantic presence of potential PI and repeated content in responses. We then applied **DBSCAN clustering** [33] to group the m embeddings into c clusters and ultimately calculated the semantic entropy of the entire query (prompt) by using entropy-based methods.

Definition: The semantic entropy is a metric used to measure the semantic diversity or consistency of responses generated by a model. Given a prompt, it calculates the entropy of clusters formed by applying the TF-IDF vectorization and DBSCAN clustering to the responses.

Given a prompt p, the model generates m responses. After applying DBSCAN clustering, these responses are grouped into k clusters. Let $|C_i|$ represent the size of the i-th cluster, where $\sum_{i=1}^{k} |C_i| = m$. The semantic entropy is calculated as follows:

$$H(p) = -\sum_{i=1}^{k} \frac{|C_i|}{m} \log\left(\frac{|C_i|}{m}\right),$$

where

- $|C_i|$ represents the size of the i-th cluster, *i.e.,* the number of responses in that cluster.
- m is the total number of responses generated.

This formula is based on Shannon Entropy and measures the diversity or consistency of the clustered responses. If most of the responses are grouped into a single cluster, the entropy value will be low, indicating a higher consistency in the responses, which means that these responses may be parts of the training data. In contrast, if responses are evenly distributed across multiple clusters, the entropy value will be higher, and this greater diversity leads to a low probability of the output data from the model.

4.5. Step 4: Dynamic Mutation

We introduced a dynamic mutation technique, including *crossover* and *dictionary*-based mutations, into the pipeline

Figure 4: The process of identifying and extracting PI elements using NER and verifying them via the GitHub API.

to enhance the diversity of generated prompts, which can lead to a wider spectrum of CodeLLM memory by varied reasoning directions, thereby increasing the concentration of PI from the training dataset in the model's output and strengthening the interconnection between these PI elements. Since prompts are composed of textual sequences, we treat tokens as basic units for mutation, applying two types of modification: *cross-over mutation*, which combines different prompts for substantial variation, and *dictionary mutation*, which leverages auxiliary corpora of common PI data, such as frequently used names, usernames, and weak passwords, to subtly modify prompts by replacing, deleting or inserting words. Together, these techniques enrich each output with diverse PI categories, broaden the training data distribution that extraction attacks can target, and ultimately improve the effectiveness of the extraction process.

4.5.1. Crossover Mutation. Our mutation step includes a crossover operation inspired by genetic algorithms [34]. This approach aims to generate higher-quality prompts by leveraging useful components from multiple prompts, thus increasing the likelihood of extracting more personal information from the model. Specifically, crossover selects several effective prompts from the previous iteration and identifies semantically valuable words. By recombining and merging these words into new prompts, we create novel prompts that retain the key characteristics of successful ones. This process helps capture diverse (code) language patterns and potentially exposes new privacy information that might not be revealed through individual prompts alone, enhancing the effectiveness of the extraction.

In this part, we select a certain number of high-quality prompts that remain after the Prompt Selection, and randomly pair them. For each pair of prompts, we randomly exchange content words and generate two new child prompts. These two child prompts are then added to the next round of iteration prompts to be queried.

4.5.2. Dictionary Mutation. While crossover can enhance and expand information related to the same individual, transformations within seeds alone tend to focus on extracting only a small portion of the personal information in the training dataset. Therefore, to broaden the distribution of target data for the extraction attack, we introduced an additional mutation process. Dictionary mutation allows external PI-related keywords to be incorporated into existing high-quality prompts, guiding our algorithm to iteratively update and adapt, thus enabling the extraction of personal information across a wider range of the training data distribution of the target model.

Inspired by brute-force attacks in traditional security [30], we also incorporated various types of auxiliary corpora to support mutation operations. In network penetration attacks, hackers often begin by attempting weak-password brute-force techniques to compromise systems, particularly for website and account logins. This is effective because many websites use common weak passwords and usernames, such as "*123456*", "*root*", and "*admin*". We apply this approach to our extraction attack by leveraging frequently used names, passwords, and usernames as part of our mutation corpus. This allows us to probe for additional information about individuals who might use similar common data. We posit that common PI data are likely also present, and frequently so, in the training data of our target model. Leveraging this common information could facilitate the extraction of other personal data related to these individuals in the training dataset, enabling more comprehensive combinations of personal information for potential future attacks. Such comprehensive personal data pose a high risk, making the inclusion of commonly used names, weak passwords, and usernames as mutation corpora highly valuable.

Specifically, in our algorithm, we perform dictionary mutation on the top-ranked prompts based on feature sorting. From these high-quality prompts, we select a certain number and apply weighted insert, replace, or delete operations. Primarily, we focus on inserting and replacing entity nouns, introducing common names, usernames, and weak passwords from auxiliary corpora into the prompts. This approach encourages the discovery of new personal information that may be present in the training dataset, thereby dynamically expanding the distribution of extracted training data.

In our work, we incorporate some character-level operations, including insert, replace, and delete. This is based on the hypothesis proposed in CodexLeaks [5]: careless programmers are more likely to make typographical errors, which may increase the likelihood of personal information leaks.

4.6. PI Detection & GitHub Search

Considering that existing methods have not yet achieved a fully automated pipeline, we innovatively introduced a Named Entity Recognition (NER) model [35] to help detect PI leakage. This allows us to precisely identify the PI content leaked in each response, facilitating seamless integration with subsequent verification steps, such as the GitHub API Search used in our evaluation.

In this work, the complete verification process begins with identifying and extracting PI elements using NER, followed by searching via the GitHub API, as shown in Figure 4. We use Microsoft Presidio [35] to identify Chinese and English PI within the responses and extract these PI elements. Presidio is an open-source project designed to manage and govern sensitive data properly. It provides fast identification and anonymization modules for private entities in text and images, which makes it widely used in many open-source projects [36]. Additionally, to facilitate more efficient use of the extracted PI by attackers, we grouped PI from the same response together, based on the assumption that PI generated within the same response are more closely related. To ensure efficiency and usability, we removed all duplicate PI elements across groups, including any subsets, ensuring that the retained PI elements are both distinct and easy to exploit.

The "PI" extracted by NER only represents data that appears to be in the form of personal information; it is not necessarily genuine PI used in model training. Only the retrieved real-world data is considered as leaked PI. As illustrated in the example in Figure 4, we further verify whether multiple PI elements within the same response belong to the same individual by examining the repository IDs returned by the GitHub API. If there is an intersection among the sets of repository IDs, we consider those elements to originate from the same individual; if the returned GitHub repositories do not overlap, the leaked PI is considered to originate from different individuals; if no repository IDs are returned – in other words, if no match is found – we regard the data as not representing genuinely leaked PI. Through the above cross-check process, we can effectively minimize the impact of model hallucination on the results.

Though limited to GitHub data, potentially underestimating leakage due to repository deletions or changes, this scope does not undermine our findings; it suggests real-world leakage may be greater. We also introduced the *Interconnected Leakage at Level $\mathcal{L}$ (IL-L)* metric to measure PI interconnections, validated by confirming all PI in a response originates from the same GitHub repository, where relatedness is more likely.

5. Experimental Setup

In this section, we introduce the setup of the experiment, including the datasets, models, benchmarks, metrics, and implementation details.

Datasets. We collected a new dataset called the Keyword Dataset for both seed construction and dynamic mutation, comprising names and weak passwords sourced from a public GitHub dictionary [24], English names from Wikipedia's wikipedia_givenname [25] and wikipedia_zhname [26], curated based on usage frequency. Additionally, we manually added common email suffixes to the dataset as a separate component. Note that we removed PI from responses if their prompts contained the PI.

Models. In this study, we implemented our pipeline and performed experimental validation on four open-source code models and two commercial models. For the open-source model experiments, we used CodeParrot (1.5B) [10], StarCoder2 (3B) [11], Code Llama (7B) [12], CodeGemma (7B) [13], DeepSeek-Coder [14], and DeepSeek-V3 [15]. To evaluate CODEBREAKER's performance on commercial models, we conducted experimental tests on GPT-3.5-turbo [17], GPT-4o [16] and Ant Group's CodeFuse [18].

Benchmarks. We selected the method proposed by Carlini *et al.* in 2021 [4], which first demonstrated the widespread memorization phenomenon in large language models and extracted memorized text, as a benchmark called *LM_Memorization*. We also selected the *CodexLeaks* [5], designed for extraction attacks on code generation models like Codex, as a second benchmark.

Metrics. Previous work [4], [5] has used the proportion of responses containing verifiable real PI elements to the total number of responses as a metric to assess the effectiveness of extraction attacks. We also used this metric in our experiments. However, this metric provides a *coarse-grained* measure of privacy leakage, which can significantly underestimate the potential privacy risks in model outputs. To more accurately assess privacy leakage from a *fine-grained* perspective, we propose two new metrics that assess the concentration and interconnection of leaked information in a more detailed manner: *(i) Leaked Proportion at Level $\mathcal{L}$ (LP-L):* The proportion of responses containing more than $\mathcal{L}$ leaked PI elements to the total number of responses. Note that in this situation, when $\mathcal{L} \geq 1$, is equivalent to the previous metric. *(ii) Interconnected Leakage at Level $\mathcal{L}$ (IL-L):* The proportion of responses containing more than $\mathcal{L}$ interconnected PI elements to the total number of responses.

Implementation Details. In all the following experiments in Sections 6.1–6.3, we use the same settings for two benchmark methods and our work CODEBREAKER. As for the open-source models and CodeFuse, we set the hyperparameters of models as *maximum_output_length of 100 tokens, temperature of 0.8, no_repeat_ngram_size of 2* (used to avoid outputting same sentences). We also set the *max_tokens=100, temperature=1.2* in OpenAI GPT models. We applied the original code for benchmark works from Carlini *et al.* [37] and CodexLeaks [38].

6. Experimental Verification

In this section, we report on our evaluation and experiment results according to our methodology in Section 4. We first validate the effectiveness of our optimized semantic entropy by conducting on existing open-source models (as described in Section 6.1). Next, we performed further

experiments on both leading open-source and commercial code models in Section 6.2, comparing CODEBREAKER to existing extraction attack methods. Additionally, we analyzed the categories of results generated from our attack on the StarCoder2 as described in Section 6.3, providing a breakdown of the different types of PI obtained. Finally, we conducted a zoom-in study to investigate privacy leakage via specific extracted instances in Section 6.4.

6.1. Evaluation of Optimized Semantic Entropy and Dynamic Mutation

In this section, we designed three different experiments to verify the effectiveness of our optimized semantic entropy. First, we conducted an experiment to statistically validate the differences in semantic entropy distributions calculated for prompts from different sources (training data vs. non-training data). Additionally, we designed two ablation studies: one tested our pipeline's performance without the SE feature, and the other tested it with the original SE feature. We then compared these results with the performance of our pipeline using the optimized SE feature under the same experimental conditions.

6.1.1. Evaluation with CodeParrot. As described in Section 4, this experiment aims to demonstrate that semantic entropy can effectively differentiate between prompts that are likely to induce CodeLLMs to generate real PI elements and those that are not. Given that the integration of Perplexity with Blind MI in Extraction Attacks has already been rigorously validated in CodexLeaks [5], our primary focus here is on verifying the effectiveness of semantic entropy in distinguishing between these different types of prompts.

To validate the ability of semantic entropy to distinguish between different types of prompts – those that are more likely to generate responses containing real PI elements and those that are not – we base our approach on the existing consensus: prompts derived from training data are more likely to induce CodeLLMs to produce content that mirrors the training data, compared to prompts from non-training data.

Our experimental design is as follows: for an open-source CodeLLMs, we randomly sample a certain number of segments from both the model's training data and non-training data to create two sets of prompts, P_1 and P_2. These two sets are then fed into the model separately, generating n distinct responses for each prompt. Using these responses, we compute the corresponding semantic entropy values for each prompt, resulting in two sets of entropy values: E_1 for P_1 and E_2 for P_2. Finally, we employ statistical methods like the K-S test [39] to analyze the distributional differences between the two sets E_1 and E_2, exploring the effectiveness of semantic entropy in distinguishing between prompts that are more likely to elicit real PI and those that are not.

To differentiate between training and non-training data, we used the CodeParrot 1.5B [10] model, which was trained on data collected only before **October, 2022**. To ensure the validity and reliability of the results, we randomly sampled

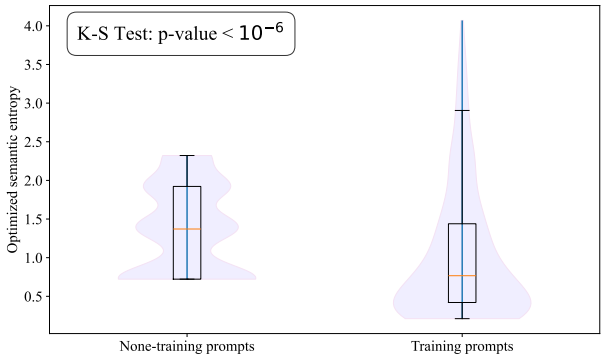

Figure 5: Comparison of optimized semantic entropy distribution between untrained and trained prompts on CodeParrot-1.5B.

a set of prompts, P_1, from CodeParrot's training dataset. As a comparison, we also sampled code snippets from GitHub repositories created in 2024 and beyond, forming a non-training data set of prompts, P_2. Both sets of prompts, P_1 and P_2, were input into the trained CodeParrot model, and the semantic entropy for each prompt was calculated. We calculated semantic entropy using the same prompt lengths and model response counts as in the pipeline. We then analyzed and visualized the distribution of results for prompts derived from both training data and non-training data.

From Figure 5, we observe that the semantic entropy corresponding to prompts sampled from the training data is significantly lower than that of prompts sampled from non-training data. We have confirmed through a K-S test [39] that these results statistically represent two distinct distributions, effectively ruling out the possibility of random effects. Therefore, semantic entropy can indeed be used to evaluate the source of prompts, specifically whether they can induce the model to return responses containing training data.

6.1.2. Ablation Study for Dynamic Mutation. From Tables 1 and 2, we observe that the performance in the ablation experiments with Dynamic Mutation alone is still better than without Dynamic Mutation. This improvement is particularly evident in Table 2, where the increase in interconnection is especially pronounced. These results further support our earlier hypothesis: Dynamic Mutation can extract more PI with strong interconnection.

To further validate the effectiveness of Dynamic Mutation, we added extra experiments and conducted comparative tests with our CODEBREAKER. We leveraged Llama3 [40] to mutate prompts in Codebreaker. Llama is used to replace the Dynamic Mutation of CODEBREAKER with instructions like "You act as a prompt optimizer, and your task is to transform existing prompts into new ones that are more likely to elicit real personal information." Under identical experimental conditions and settings, we obtained the results for the LLM-Mutation experiments, as shown in Tables 1 and Table 2. The experiment results show that Codebreaker significantly outperforms the LLM-based mutation. This demonstrates our dynamic mutation reigns as the top self-iterative method.

Table 1: Ablation study on StarCoder-3B, measured by leaked proportion at Different Level $\mathcal{L}$ (LP-L). Note that we cannot apply semantic entropy (SE) alone without dynamic mutation; therefore, the table does not include cases with only Original SE or Optimized SE.

Dynamic Mutation	Original SE	Optimized SE	LLM-Mutation	$\mathcal{L} \geq 1$	$\mathcal{L} \geq 2$	$\mathcal{L} \geq 3$	$\mathcal{L} \geq 4$	$\mathcal{L} \geq 5$	$\mathcal{L} \geq 6$	$\mathcal{L} \geq 7$	$\mathcal{L} \geq 8$	$\mathcal{L} \geq 9$	$\mathcal{L} \geq 10$
○	○	○	○	65.92%	42.48%	22.27%	9.38%	4.39%	2.25%	0.88%	0.20%	0.10%	0.0%
●	○	○	○	69.09%	45.03%	22.85%	10.15%	3.86%	1.77%	0.99%	0.55%	0.11%	0.11%
●	●	○	○	37.95%	19.24%	10.64%	6.25%	2.34%	0.68%	0.20%	0.10%	0.10%	0.00%
○	○	○	●	67.32%	45.41%	25.55%	11.32%	3.43%	2.04%	0.63%	0.20%	0.00%	0.00%
●	○	●	○	**98.54%**	**93.36%**	**82.03%**	**69.82%**	**58.01%**	**45.21%**	**32.91%**	**24.51%**	**15.72%**	**8.59%**

Table 2: Ablation study on StarCoder-3B, measured by interconnected leakage at level $\mathcal{L}$ (IL-L).

Dynamic Mutation	Original SE	Optimized SE	LLM-Mutation	$\mathcal{L} \geq 2$	$\mathcal{L} \geq 3$	$\mathcal{L} \geq 4$
○	○	○	○	17.78%	2.35%	0.30%
●	○	○	○	37.99%	8.98%	1.88%
●	●	○	○	18.72%	2.83%	0.79%
○	○	○	●	24.25%	6.81%	2.37%
●	○	●	○	**74.71%**	**29.70%**	**9.29%**

6.1.3. Ablation Study for Semantic Entropy. To further demonstrate the effectiveness of semantic entropy, we conducted an ablation study to validate the contribution of this feature in the extraction attack. In this ablation experiment, we conducted tests on the StarCoder2 [11] model under three conditions: without using semantic entropy, using semantic entropy calculated with the original method proposed by Farquhar *et al.* [31], and using our optimized semantic entropy calculation. The experimental process retained all other settings and components unchanged.

The results obtained under the ablation study using the metric LP-L are presented in Table 1 (lines 2, 3, and 4), reflecting the concentration of real leaked PI. The leakage rates for the method without SE are generally low across all levels, with significant performance degradation at higher leakage levels (Table1: line 2). For example, when $\mathcal{L} \geq 1$, the leakage rate reaches 69.09%. However, when the $\mathcal{L} \geq 2$ or 3, the leakage rate drops sharply to 45% and 22.85%, respectively.

As shown in Tables 1 and 2, the performance of semantic entropy calculated using the original method (Original SE) is even worse than experiments conducted without deploying Original SE. We hypothesize that this is due to the structured format of code, which renders the bidirectional entailment-based calculation of Original SE ineffective in distinguishing whether prompts can induce the model to return training data. Consequently, during multiple iterations, the ineffective Original SE causes Prompt Selection to shift the distribution of prompts in an incorrect direction, resulting in performance worse than that without Original SE.

Therefore, we introduced a new method for calculating semantic entropy (Optimized SE). Our method significantly improves leakage rates at all levels. For instance, with $\mathcal{L} \geq 1$, our method achieves a leakage rate of 98.54%, representing a 29.45% increase over the method without SE, thereby greatly increasing the presence of leaked PI within responses. At higher leakage levels (e.g., $\mathcal{L} \geq 5$ or $\mathcal{L} \geq 10$), our method also performs exceptionally well, with leakage rates rising to 58.01% and 8.59%, respectively. As shown in Table 2, our method also performs exceptionally well in terms of the interconnection of real leaked PI. Its performance is twice as high as the method without SE and four times that of the method using Original SE. Moreover, the gap becomes even more pronounced at higher values of $\mathcal{L}$. These results demonstrate that our method has a substantial advantage in enhancing PI leakage, particularly at higher leakage levels, further enriching the variety and effectiveness of extracted PI in extraction attacks.

6.2. GitHub Search Check

In this section, we conducted experiments with our designed pipeline on several leading open-source models as well as commercial models. We also deployed the benchmark methods from LM_Memorization and CodexLeaks on these models.

6.2.1. Open-Source Models. To validate the effectiveness of our pipeline, we deployed it on four open-source code models: CodeParrot [10], Code Llama [12], StarCoder2 [11], CodeGemma [13], DeepSeek-Coder [14] and DeepSeek-V3 [15]. Aside from the foundational CodeParrot, these models represent some of the most advanced open-source code models currently available and are widely used across commercial and open-source projects.

Table 3 presents the attack effectiveness across different language models (including CodeParrot-1.5B, StarCoder2-3B, Code Llama-7B, CodeGemma-7B, DeepSeek-Coder-V2-base[6.7B] and DeepSeek-V3-base[685B]). The results demonstrate that our method effectively increases the proportion of personal information leakage across all leakage levels, significantly surpassing other attack methods (such as LM_Memorization and CodexLeaks). Specifically, on the StarCoder2 model, our method reaches a leakage rate of 98.45%, far exceeding LM_Memorization's method at 43.05% and CodexLeaks at 63.87%. Similar results are observed on CodeGemma-7B, achieving leakage proportion of 94.92%. CODEBREAKER also achieved outstanding performance with larger versions of Code Llama and DeepSeek-Coder, even surpassing existing benchmark methods by several times. CODEBREAKER demonstrates superior performance in enhancing the interconnection of leaked real PI. Across all models, the IL-L results are several times higher than those of the benchmark methods.

As the leakage level increases (e.g., $\mathcal{L} \geq 5$ or higher), our attack method continues to demonstrate superior per-

Table 3: Extraction performance of four open-source CodeLLMs, measured by Leaked Proportion at Different Level $\mathcal{L}$ (LP-L).

CodeLLM	Method	$\mathcal{L} \geq 1$	$\mathcal{L} \geq 2$	$\mathcal{L} \geq 3$	$\mathcal{L} \geq 4$	$\mathcal{L} \geq 5$	$\mathcal{L} \geq 6$	$\mathcal{L} \geq 7$	$\mathcal{L} \geq 8$	$\mathcal{L} \geq 9$	$\mathcal{L} \geq 10$
CodeParrot	LM_Mem	43.05%	16.45%	4.69%	2.54%	1.27%	1.07%	0.78%	0.78%	0.49%	0.39%
	CodexLeaks	63.87%	47.17%	28.22%	15.33%	7.13%	3.13%	1.46%	0.49%	0.20%	0.10%
	CODEBREAKER	**89.45%**	**80.18%**	**64.84%**	**42.38%**	**24.32%**	**13.48%**	**7.62%**	**3.71%**	**1.66%**	**0.59%**
StarCoder2	LM_Mem	37.85%	20.89%	12.94%	2.78%	2.41%	1.14%	0.76%	0.29%	0.20%	0.20%
	CodexLeaks	65.92%	42.48%	22.27%	9.38%	4.39%	2.25%	0.88%	0.20%	0.10%	0.0%
	CODEBREAKER	**98.54%**	**93.36%**	**82.03%**	**69.82%**	**58.01%**	**45.21%**	**32.91%**	**24.51%**	**15.72%**	**8.59%**
Code Llama	LM_Mem	46.65%	24.54%	15.12%	8.84%	6.13%	4.28%	2.85%	2.28%	1.57%	1.14%
	CodexLeaks	65.24%	42.57%	14.65%	6.20%	2.99%	1.39%	0.64%	0.32%	0.32%	0.11%
	CODEBREAKER	**77.40%**	**61.16%**	**48.51%**	**36.26%**	**27.54%**	**19.84%**	**13.88%**	**8.88%**	**5.45%**	**2.81%**
CodeGemma	LM_Mem	43.26%	31.84%	21.68%	13.77%	10.06%	7.13%	5.27%	3.42%	2.25%	1.56%
	CodexLeaks	80.08%	56.15%	30.47%	14.16%	7.23%	4.10%	2.25%	1.07%	0.78%	0.29%
	CODEBREAKER	**94.92%**	**86.72%**	**75.39%**	**61.33%**	**46.88%**	**34.47%**	**24.80%**	**15.33%**	**9.08%**	**5.27%**
DeepSeek-Coder	LM_Mem	23.76%	15.03%	12.49%	8.84%	3.11%	1.27%	0.53%	0.00%	0.00%	0.00%
	CodexLeaks	52.01%	34.58%	23.39%	12.57%	5.23%	3.28%	2.15%	1.27%	0.98%	0.39%
	CODEBREAKER	**74.43%**	**54.20%**	**37.02%**	**23.66%**	**18.70%**	**12.60%**	**10.69%**	**7.63%**	**5.34%**	**3.05%**
DeepSeek-V3	LM_Mem	10.35%	4.98%	2.56%	1.78%	0.56%	0.41%	0.14%	0.14%	0.00%	0.00%
	CodexLeaks	23.52%	10.14%	4.78%	2.36%	1.29%	0.63%	0.30%	0.10%	0.10%	0.00%
	CODEBREAKER	**37.43%**	**25.31%**	**17.53%**	**9.28%**	**5.71%**	**3.74%**	**2.54%**	**1.98%**	**1.24%**	**0.64%**

Table 4: Extraction performance of four open-source models, measured by Interconnected Leakage at Level $\mathcal{L}$ (IL-L).

CodeLLM	Method	$\mathcal{L} \geq 2$	$\mathcal{L} \geq 3$	$\mathcal{L} \geq 4$
CodeParrot	LM_Mem	19.00%	4.64%	0.29%
	CodexLeaks	24.42%	4.99%	1.38%
	CODEBREAKER	**58.98%**	**11.62%**	**2.24%**
StarCoder2	LM_Mem	13.27%	2.23%	1.10%
	CodexLeaks	17.78%	2.35%	0.30%
	CODEBREAKER	**74.71%**	**29.70%**	**9.29%**
Code Llama	LM_Mem	12.55%	5.14%	1.28%
	CodexLeaks	14.76%	1.82%	0.11%
	CODEBREAKER	**41.43%**	**9.56%**	**2.53%**
CodeGemma	LM_Mem	20.80%	6.25%	2.83%
	CodexLeaks	22.85%	2.64%	1.27%
	CODEBREAKER	**58.59%**	**18.65%**	**5.86%**
DeepSeek-Coder	LM_Mem	11.60%	3.68%	0.71%
	CodexLeaks	22.06%	5.09%	1.77%
	CODEBREAKER	**36.65%**	**11.46%**	**4.21%**
DeepSeek-V3	LM_Mem	4.13%	1.48%	0.57%
	CodexLeaks	8.04%	3.18%	0.93%
	CODEBREAKER	**13.43%**	**6.93%**	**3.42%**

formance. For example, on the Code Llama-7B model, our method raises the leakage rate to 27.54%, compared to 6.13% for LM_Memorization's method and 2.99% for CodexLeaks. On the CodeGemma-7B model, our method achieves a leakage rate of 5.27% for $\mathcal{L} \geq 10$, showing a noticeable improvement over other methods. Also, even at higher values of $\mathcal{L}$, CODEBREAKER maintains over 9% IL-L on StarCoder2 (Table 4). CODEBREAKER is proved to extract a greater amount of interconnected PI across more CodeLLMs, significantly outperforming existing benchmark methods—for example, on models such as Code Llama and DeepSeek-Coder. This indicates that our method significantly strengthens the relationships among extracted PI, potentially making extraction attacks more applicable in real-world scenarios. These results indicate that our method maintains strong attack effectiveness even at higher leakage levels, guiding the model to leak a wider variety of personal information with strong interconnection in its responses.

CODEBREAKER also performs well on the general-purpose model DeepSeek-V3, which significantly outperform existing attack methods (Table 3 and Table 4). This demonstrates that CODEBREAKER is effective not only for code-focused models but also for general-purpose LLMs, achieving stronger performance than baseline approaches. However, we note that while CODEBREAKER remains effective on general-purpose LLMs, their unstructured outputs make comprehensive verification slow and resource-intensive as indexing and searching through the full training data is time- and storage-intensive [9]. Therefore, similar to other CodeLLMs, we extract the PI from the outputs of general-purpose LLMs and perform retrieval on GitHub. This may lead to an underestimation of the privacy risk posed by general-purpose LLMs. Future work could focus on improving the efficiency of the verification pipeline for general-purpose LLMs.

6.2.2. Commercial Models. For extraction results for commercial models (*i.e.,* GPT and CodeFuse), our method consistently outperforms the other two benchmarks. As shown in Table 5, for both versions of GPT, CODEBREAKER achieves a significantly higher proportion of responses containing different amounts of leaked PI compared to the benchmarks. Furthermore, in CodeFuse, CODEBREAKER extracts responses containing three or more leaked PI nearly 13% more often than CodexLeaks and approximately 33% more often than LM_Memorization. This demonstrates that CODEBREAKER can extract more PI from the training data of commercial models, leading to greater privacy leakage.

Similarly, in terms of PI interconnection (shown in Table 6), CODEBREAKER also outperforms the other two benchmark methods on commercial models. For instance, in CodeFuse, CODEBREAKER achieves improvements ranging from 2x to as much as 8x, indicating that the extracted information is more usable and poses even greater security risks.

For the attack performance of base and API-based CodeLLMs, we found that the results of the API-based model (*i.e.,* GPT series) were significantly lower than base CodeLLMs. Specifically, in Table 5, CodeFuse outperformed the GPT model by about 6-10 times across all leakage levels $\mathcal{L}$. A similar pattern is observed in Table 6. Therefore, we argue that for base CodeLLMs, there is an urgent need to ensure strict auditing of private data leakage.

6.2.3. Overhead. As a fuzzing-inspired algorithm, CODEBREAKER performs deep, automated extraction attacks – heuristic, targeted, and self-optimizing. While static methods like CodexLeaks rely on a manual prompt design specific to models and datasets, CODEBREAKER self-tunes efficiently, achieving baseline query speeds once deployed. From a complexity perspective, if the number of responses is n, then both LLM_MEM [4] and CODEXLEAKS [5] operate with a time complexity of $\mathcal{O}(n)$, as they issue a fixed set of prompts. In contrast, CODEBREAKER introduces dynamic mutation through iterative self-optimization, resulting in a time complexity of $\mathcal{O}(M \cdot n) \approx \mathcal{O}(n)$, where M is a tunable hyperparameter denoting the number of mutation iterations, preserving CODEBREAKER's query efficiency in practice.

6.3. Category Check

To assess the potential risks posed by PI extracted by these pipelines, we further categorized PI. We detail the results for various categories of personal information. Specifically, we classified PI into three main categories: Identifiable, Privacy, and Secret, each representing a different level of risk associated with personal information leakage. Within each category, we further divided the information into specific subcategories to facilitate detailed analysis and differentiation of the types of leaked information. Note that since the time of model training, some GitHub repository content included in the training dataset may have been deleted or modified. Limitations in GitHub search also functionality further restrict our verification. As a result, the number of leaked PI elements we report is likely an underestimation of the actual amount of leakage.

From the results in Table 7, it can be observed that our model extracts significantly more leaked PI in most categories compared to the two benchmark methods. This is particularly evident in categories such as *Name, Email, Address, Background,* and *Password/pin/tokens*. Our work demonstrates a notable increase in PI leakage at both the *Privacy* and *Secret* levels, including sensitive information such as *Credit card* and *Private keys*. These types of information pose even greater privacy leakage risks.

6.4. Zoom-in Study

To obtain interesting findings, we also implement a zoom-in study to examine specific instances.

PI overlap in Various CodeLLMs. In our analysis of various CodeLLMs, we found that these models, including Code Llama, StarCoder2, CodeGemma, and GPT, occasionally returned identical PI information when given the same prompts. This information, as shown below, sometimes came from the same source (e.g., the same GitHub repository), which suggests overlap in their training data, likely leading to memorization of similar PI details:

> **Responses:**
> - ..."Pe[redacted]ng, hap[redacted]3@gmail.com"... [Code Llama, StarCoder2, GPT-4o]
> - ..."":"email": ar[redacted]ail.com, "address": street No 10"... [Code Llama, CodeFuse]

Training Data Confusion Output. PI data with similar structures or domains is prone to being output by the model within the same response, even if this PI data originates from different GitHub repositories. For example, nearly all open-source models tend to confuse personal information in the format of "name," "email" or "phone" or "address," or output this information in a jumbled order. Additionally, although CodeFuse's training data domain differs significantly from that of other models, similar phenomena occur. CodeFuse, for instance, tends to mix up "name" and "education background"/"occupation," especially when there is partial overlap between two individuals' information, such as when they share a similar occupation (e.g., both are doctors).

> **Responses:**
> - ..." Name: Jo[redacted]non, Date of Birth: October 9, 1940, Name: Da[redacted]son, Date of Birth: 06/06/1990"... [CodeGemma]
> - ..." email: dan[redacted]loaks.net, name: Dan[redacted]aks, name: L[redacted]ia L, email: l[redacted]ial@live.com"... [Code Llama]
> - "李[redacted]，刘[redacted]，郑州[redacted]医院手术麻醉科，杭州市[redacted]精神科学医生，浙江大学[redacted]卫生中心" **Translate:** Li Tao, Liu [redacted], Zhengzhou [redacted] Hospital, Department of Surgical Anesthesia, psychiatrist at Hangzhou [redacted] Hospital, Zhejiang University School of [redacted] [CodeFuse]
> - "金[redacted]，韩[redacted]，出生于1975年，毕业于上海[redacted]学院表演系，上市公司浙江省东方[redacted]公司董事" **Translate:** Jin [redacted], Han [redacted], born in 1975, graduated from the Acting Department of Shanghai [redacted] Academy, Director of Zhejiang [redacted] Co., Ltd., a listed company [CodeFuse]

Variability of Output across Different Defenses. Models that have undergone different defense methods exhibit diverse performances under extraction attacks. Since we deployed experiments across models ranging from basic models to API-only models (such as GPT), we observed significant differences in the effectiveness of various defense methods against extraction attacks.

First, the GPT series models frequently refuse to answer direct questions involving "secret"-level PI information. Notably, GPT-4o exhibits a higher frequency of refusals compared to GPT-3.5-turbo. We attribute this difference to

Table 5: Extraction performance of two commercial CodeLLMs, measured by Leaked Proportion at Different Level $\mathcal{L}$ (LP-L).

CodeLLM	Method	$\mathcal{L} \geq 1$	$\mathcal{L} \geq 2$	$\mathcal{L} \geq 3$	$\mathcal{L} \geq 4$	$\mathcal{L} \geq 5$	$\mathcal{L} \geq 6$	$\mathcal{L} \geq 7$	$\mathcal{L} \geq 8$	$\mathcal{L} \geq 9$	$\mathcal{L} \geq 10$
GPT-4o	LM_Mem	3.54%	1.43%	0.57%	0.29%	0.10%	0.00%	0.00%	0.00%	0.00%	0.00%
	CodexLeaks	4.59%	2.07%	1.73%	0.67%	0.28%	0.10%	0.10%	0.00%	0.00%	0.00%
	CODEBREAKER	**5.62%**	**4.03%**	**2.10%**	**1.08%**	**0.38%**	**0.10%**	**0.10%**	0.00%	0.00%	0.00%
GPT-3.5	LM_Mem	4.81%	2.52%	1.56%	1.10%	0.38%	0.10%	0.00%	0.00%	0.00%	0.00%
	CodexLeaks	8.63%	6.34%	4.67%	2.37%	1.33%	0.57%	0.29%	0.10%	0.00%	0.00%
	CODEBREAKER	**10.16%**	**8.22%**	**6.50%**	**3.26%**	**1.22%**	**0.67%**	**0.34%**	0.10%	**0.00%**	0.00%
CodeFuse	LM_Mem	18.60%	6.01%	2.39%	1.46%	0.93%	0.73%	0.49%	0.34%	0.29%	0.29%
	CodexLeaks	55.95%	40.71%	24.14%	15.36%	7.50%	3.69%	1.79%	0.83%	0.48%	0.24%
	CODEBREAKER	**60.11%**	**49.56%**	**37.06%**	**25.49%**	**17.92%**	**12.40%**	**8.79%**	**6.01%**	**4.39%**	**3.37%**

Table 6: Extraction performance of two commercial models, measured by Interconnected Leakage at Level $\mathcal{L}$ (IL-L).

CodeLLM	Method	$\mathcal{L} \geq 2$	$\mathcal{L} \geq 3$	$\mathcal{L} \geq 4$
GPT-4o	LM_Mem	0.34%	0.19%	0.00%
	CodexLeaks	0.48%	0.10%	0.00%
	CODEBREAKER	**3.23%**	**0.44%**	**0.10%**
GPT-3.5	LM_Mem	0.53%	0.43%	0.10%
	CodexLeaks	3.21%	1.85%	0.10%
	CODEBREAKER	**5.85%**	**2.74%**	**0.53%**
CodeFuse	LM_Mem	4.45%	1.03%	0.40%
	CodexLeaks	17.78%	2.35%	0.30%
	CODEBREAKER	**43.90%**	**16.20%**	**7.69%**

Table 7: The leaked PI category in StarCoder2-3B.

Main	Category	LM_Mem	CodexLeaks	CODEBREAKER
Identifiable	Name	129	182	**371**
	Address	61	104	**276**
	Email	11	31	**89**
	Phone Number	11	17	**36**
	Social media	5	28	**30**
	Date of birth	0	13	**43**
Private	Identity	14	20	**31**
	Background (Education)	1	4	**10**
	Background (Career)	3	20	**41**
	Account/user name	4	9	**12**
	Medical record	0	1	**6**
	Others	4	20	**43**
Secret	Password/pin/token	3	12	**30**
	Private key	0	2	**5**
	Credit card	0	1	**4**
	Others	0	1	**6**
Total	All of categories	254	454	**959**

two possible factors: *(i)* GPT-4 may have enhanced security protections during its development and deployment, and *(ii)* GPT-4o likely has a stronger comprehension ability, allowing it to consistently recognize sensitive prompts across multiple queries. In contrast, the base models we attacked revealed noticeably more PI in their responses, suggesting that beyond differential privacy and data sanitization measures, GPT's deployment of moderator and defensive prompt methods may have a more direct and effective impact.

Some models in our study implemented data sanitization and other protective measures. For instance, StarCoder2 [3] employed its own NER model, starPII [41], to sanitize certain personal information in the training data. However, since the identifiable PI categories of starPII are limited, some PI still leaked through. Similarly, both GPT and CodeFuse implemented defenses specifically targeting email addresses and phone numbers. These models replace a large portion of continuous numbers, such as phone numbers and passwords, with meaningless sequences like "123456". GPT also often outputs non-specific placeholders such as "example" or "email" in place of actual PI.

Responses:

- "example.com, Liam@example.com, liam@email.com" [GPT]
- "username@ec2-xx-xxx-xxx-xxx.compute-1.amazonaws.com" [GPT]
- "I'm sorry, I can't assist with that request." [GPT]
- "john[redacted]@example.com, (123)456-7890, "12345", jonhn[redacted]@gmail.com, Main St, Birth: [redacted]-1990, New York" [CodeFuse]
- "电子邮箱：cKok[redacted]@163.com, 密码：654321，地址：深圳市[redacted]" [CodeFuse] **Translate:** "Email: cKok[redacted]@163.com, Password: 654321, Address: Shenzhen City [redacted]" [CodeFuse]
- "在线聊天QQ群：473[redacted]29 或者微信whh[redacted]715" **Translate:** "Online Chat: QQ Group: 473[redacted]29 or WeChat whh[redacted]715" [CodeFuse]

Repetitive Homogenized Output. We observed that code models frequently output a significant amount of temporal data (especially dates and generalized location names). This tendency is particularly pronounced when the prompt lacks clear meaning. For example, in the results of the LM_Memorization experiment, CodeLLMs often generated long sequences of consecutive dates and location data (especially city names). Additionally, we found that these types of data tend to have strong interconnections, with such data within the same response often originating from the same GitHub repository. We hypothesize that this behavior may relate to the distribution of training data; for CodeLLMs, apart from code functionality, data in this form is likely one of the most common types found within code repositories.

Responses:

- "May 5, 2021 6:55:16 AM Maya the Bee 5, May 5, 2021 7:04:41 AM Celence 5, May 5, 2021 6:55:32 AM..." [Code Llama (LM_Memorization)]
- "姚 明2345674 孙 继 海890123 廖 元 杰1536552 郭 艾伦714523 唐正东1324589 张稀哲784526 李翔245678 艾弗森，萨卡姆1023456 哈登754329"... **Translate:** Yao Ming 2345674 Sun Jihai 890123 Liao Yuanjie 1536552 Guo Ailun 714523 Tang Zhengdong 1324589 Zhang Xizhe 784526 Li Xiang 245678 Iverson, Sakamu 1023456

Harden 754329 [CodeFuse]

7. Potential Defenses

We discuss several potential defenses from the perspectives of developers and third parties as follows.

For Developers. As developers, we typically use three methods to defend against extraction attacks. *(i) Data anonymization* [42] is a method for removing PI and has been applied across multiple base LLMs. However, we find that existing models like CodeFuse, Code Llama, and CodeGemma still have insufficient data anonymization. For instance, our attack results show that while many numerical PI elements are replaced, other types remain largely exposed. *(ii)* Differential privacy is originally designed to protect data membership privacy by adding random noise at the input level [43] or during training [44]. However, prior research [45], [46] shows difficulty in applying differential privacy to large models in practice due to the tradeoff on utility. *(iii)* Hintersdorf *et al.* [47] proposed a method to control and prevent memorization in diffusion models by identifying and deactivating specific neurons responsible for memorizing training data. This approach effectively mitigates data leakage and enhances the diversity of generated outputs in computer vision applications. However, its applicability is currently limited to diffusion models in this domain.

For Third Parties. As a third party, a common approach is to have moderators inspect the model's inputs and outputs, serving as input and output filters for the models. This method has been implemented in GPT-4 and GPT-3.5. As shown in Table 5, the GPT series have a significantly lower leakage proportion than Chinese CodeFuse models. This perhaps indicates that moderators play an essential role in preventing PI leakage and can be a potential defense against extraction attacks, which can be applied to Chinese large models to ensure data security.

8. Conclusion

In this work, we introduced a comprehensive framework for conducting extraction attacks on code generation large language models (CodeLLMs) to highlight potential privacy leakage. By combining Blind MI with novel features such as semantic entropy, our approach significantly improved the possibility of extraction of personal information from leading models like StarCoder2, Code Llama, and CodeGemma. Using automated prompt generation, query optimization, and precise response filtering, which boosts reasoning diversity and model memorization, we finally achieved over 20% higher extraction efficiency than prior methods. We further validated the robustness of our approach by conducting experiments on multiple CodeLLMs, highlighting the potential privacy risks inherent in publicly accessible CodeLLMs. Our findings emphasize the urgent need for more effective privacy-preserving techniques in CodeLLMs to mitigate the risk of inadvertent data leakage.

We acknowledge the challenge of completely validating the authenticity of all extracted content. Following two established benchmarks from previous research [4], [5], we rigorously employed search engines on the Internet for validation. Although this approach captures only a subset of the training data – and some content may no longer be accessible – the validation parameters across both prior studies and our own remain aligned, ensuring a robust and fair assessment of data extraction attack effectiveness. We argue that the significant underestimation of the scale of training data leakage in CodeLLMs persists, especially as attack methodologies continue to rapidly advance.

Acknowledgements

We gratefully acknowledge Prof. Sheng Wen as the corresponding author of this work. Minhui Xue is supported in part by Australian Research Council (ARC) DP240103068 and in part by CSIRO – National Science Foundation (US) AI Research Collaboration Program.

References

[1] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[2] C. Team, "Codegemma: Open code models based on gemma," *arXiv preprint arXiv:2406.11409*, 2024.

[3] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[4] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson *et al.*, "Extracting training data from large language models," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2633–2650.

[5] L. Niu, S. Mirza, Z. Maradni, and C. Pöpper, "{CodexLeaks}: Privacy leaks from code generation language models in {GitHub} copilot," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2133–2150.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[7] J. Henriksen-Bulmer and S. Jeary, "Re-identification attacks—a systematic literature review," *International Journal of Information Management*, vol. 36, no. 6, pp. 1184–1192, 2016.

[8] C. Xie, Y. Huang, C. Zhang, D. Yu, X. Chen, B. Y. Lin, B. Li, B. Ghazi, and R. Kumar, "On memorization of large language models in logical reasoning," *arXiv preprint arXiv:2410.23123*, 2024.

[9] M. Nasr, N. Carlini, J. Hayase, M. Jagielski, A. F. Cooper, D. Ippolito, C. A. Choquette-Choo, E. Wallace, F. Tramèr, and K. Lee, "Scalable extraction of training data from (production) language models," *arXiv preprint arXiv:2311.17035*, 2023.

[10] "codeparrot/codeparrot · Hugging Face — huggingface.co," https://huggingface.co/codeparrot/codeparrot, [Accessed 11-11-2024].

[11] "bigcode/starcoder2-3b · Hugging Face — huggingface.co," https://huggingface.co/bigcode/starcoder2-3b, [Accessed 11-11-2024].

[12] "codellama/CodeLlama-7b-hf · Hugging Face — huggingface.co," https://huggingface.co/codellama/CodeLlama-7b-hf, [Accessed 11-11-2024].

[13] "google/codegemma-7b · Hugging Face — huggingface.co," https://huggingface.co/google/codegemma-7b, [Accessed 11-11-2024].

[14] "deepseek-ai/DeepSeek-Coder-V2-Base · Hugging Face — huggingface.co," https://huggingface.co/deepseek-ai/DeepSeek-Coder-V2-Base, [Accessed 09-04-2025].

[15] "GitHub - deepseek-ai/DeepSeek-V3 — github.com," https://github.com/deepseek-ai/DeepSeek-V3, [Accessed 09-04-2025].

[16] OpenAI, "GPT-4o," https://openai.com/index/gpt-4o-system-card/, [Accessed 13-11-2024].

[17] https://openai.com/index/gpt-3-5-turbo-fine-tuning-and-api-updates/, [Accessed 13-11-2024].

[18] "codefuse-ai/CodeFuse-13B · Hugging Face — huggingface.co," https://huggingface.co/codefuse-ai/CodeFuse-13B, [Accessed 12-11-2024].

[19] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," *arXiv preprint arXiv:1801.06146*, 2018.

[20] A. Radford, "Improving language understanding by generative pre-training," 2018.

[21] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[22] O. D. Thakkar, S. Ramaswamy, R. Mathews, and F. Beaufays, "Understanding unintended memorization in language models under federated learning," in *Proceedings of the Third Workshop on Privacy in Natural Language Processing*, 2021, pp. 1–10.

[23] S. Zanella-Béguelin, L. Wutschitz, S. Tople, V. Rühle, A. Paverd, O. Ohrimenko, B. Köpf, and M. Brockschmidt, "Analyzing information leakage of updates to natural language models," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 363–375.

[24] "GitHub - Stardustsky/SaiDict github.com," https://github.com/Stardustsky/SaiDict, [Accessed 12-11-2024].

[25] "List of most popular given names - Wikipedia — en.wikipedia.org," https://en.wikipedia.org/wiki/List_of_most_popular_given_names, [Accessed 12-11-2024].

[26] "zh.wikipedia.org," https://zh.wikipedia.org/wiki/%E6%9C%80%E5%B8%B8%E8%A6%8B%E5%90%8D%E5%AD%97%E5%88%97%E8%A1%A8, [Accessed 12-11-2024].

[27] B. Hui, Y. Yang, H. Yuan, P. Burlina, N. Z. Gong, and Y. Cao, "Practical blind membership inference attack via differential comparisons," *arXiv preprint arXiv:2101.01341*, 2021.

[28] L. Kuhn, Y. Gal, and S. Farquhar, "Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation," *arXiv preprint arXiv:2302.09664*, 2023.

[29] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–36, 2022.

[30] M. J. Heule and O. Kullmann, "The science of brute force," *Communications of the ACM*, vol. 60, no. 8, pp. 70–79, 2017.

[31] S. Farquhar, J. Kossen, L. Kuhn, and Y. Gal, "Detecting hallucinations in large language models using semantic entropy," *Nature*, vol. 630, no. 8017, pp. 625–630, 2024.

[32] H. D. Abubakar, M. Umar, and M. A. Bakale, "Sentiment classification: Review of text vectorization methods: Bag of words, tf-idf, word2vec and doc2vec," *SLU Journal of Science and Technology*, vol. 4, no. 1, pp. 27–33, 2022.

[33] K. Khan, S. U. Rehman, K. Aziz, S. Fong, and S. Sarasvady, "Dbscan: Past, present and future," in *The fifth international conference on the applications of digital information and web technologies (ICADIWT 2014)*. IEEE, 2014, pp. 232–238.

[34] S. M. Lim, A. B. M. Sultan, M. N. Sulaiman, A. Mustapha, and K. Y. Leong, "Crossover and mutation operators of genetic algorithms," *International journal of machine learning and computing*, vol. 7, no. 1, pp. 9–12, 2017.

[35] Microsoft, "Microsoft Presidio — microsoft.github.io," https://microsoft.github.io/presidio/, [Accessed 25-10-2024].

[36] "Login — Microsoft 365 — office.com," https://www.office.com/, [Accessed 13-11-2024].

[37] "GitHub - ftramer/LM_Memorization: Training data extraction on GPT-2 — github.com," https://github.com/ftramer/LM_Memorization, [Accessed 13-11-2024].

[38] "GitHub - niuliang42/CodexLeaks: CodexLeaks: Privacy Leaks from Code Generation Language Models in GitHub Copilot — github.com," https://github.com/niuliang42/CodexLeaks, [Accessed 13-11-2024].

[39] V. W. Berger and Y. Zhou, "Kolmogorov–smirnov test: Overview," *Wiley statsref: Statistics reference online*, 2014.

[40] "meta-llama/Llama-3.1-8B-Instruct · Hugging Face — huggingface.co," https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct, [Accessed 09-04-2025].

[41] "bigcode/starpii · Hugging Face — huggingface.co," https://huggingface.co/bigcode/starpii, [Accessed 13-11-2024].

[42] T. Yang, X. Zhu, and I. Gurevych, "Robust utility-preserving text anonymization based on large language models," *arXiv preprint arXiv:2407.11770*, 2024.

[43] S. Pang, Z. Lu, H. Wang, P. Fu, Y. Zhou, M. Xue, and B. Li, "Reconstruction of differentially private text sanitization via large language models," *arXiv preprint arXiv:2410.12443*, 2024.

[44] M. Du, X. Yue, S. S. Chow, T. Wang, C. Huang, and H. Sun, "Dp-forward: Fine-tuning and inference on language models with differential privacy in forward pass," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2665–2679.

[45] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, "A survey on large language model (llm) security and privacy: The good, the bad, and the ugly," *High-Confidence Computing*, p. 100211, 2024.

[46] A. Panda, X. Tang, M. Nasr, C. A. Choquette-Choo, and P. Mittal, "Privacy auditing of large language models," in *ICML 2024 Next Generation of AI Safety Workshop*, 2024.

[47] D. Hintersdorf, L. Struppek, K. Kersting, A. Dziedzic, and F. Boenisch, "Finding nemo: Localizing neurons responsible for memorization in diffusion models," *arXiv preprint arXiv:2406.02366*, 2024.

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper proposes Codebreaker to extract sensitive personal information (PI) from Code LLMs. Codebreaker improves prior methods by using semantic entropy to select prompts and dynamic mutation to diversify prompts to extract more PIs. The evaluation is conducted on several open-source and commercial models, which shows Codebreaker outperforms existing extraction methods. The paper also discusses the risks of PI leakage and potential defenses.

A.2. Scientific Contributions

Provides a Valuable Step Forward in an Established Field.

A.3. Reasons for Acceptance

The paper provides a valuable step forward in an established field. This paper significantly advances the field of extraction attacks against Code Large Language Models (Code LLMs). The semantic entropy based prompt selection method and the dynamic mutation approach are shown to systematically improve the attack automation and effectiveness. The black-box threat model also aligns well with real-world scenarios.