Effective Code Membership Inference for Code Completion Models via Adversarial Prompts

Yuan Jiang†, Zehao Li†, Shan Huang†, Christoph Treude§, Xiaohong Su†, Tiantian Wang†
†Harbin Institute of Technology §Singapore Management University
{jiangyuan, sxh, wangtiantian}@hit.edu.cn, {2021110768, 2022110145}@stu.hit.edu.cn, ctreude@smu.edu.sg

***Abstract*—Membership inference attacks (MIAs) on code completion models offer an effective way to assess privacy risks by inferring whether a given code snippet was part of the training data. Existing black- and gray-box MIAs rely on expensive surrogate models or manually crafted heuristic rules, which limit their ability to capture the nuanced memorization patterns exhibited by over-parameterized code language models. To address these challenges, we propose AdvPrompt-MIA, a method specifically designed for code completion models, combining code-specific adversarial perturbations with deep learning. The core novelty of our method lies in designing a series of adversarial prompts that induce variations in the victim code model's output. By comparing these outputs with the ground-truth completion, we construct feature vectors to train a classifier that automatically distinguishes member from non-member samples. This design allows our method to capture richer memorization patterns and accurately infer training set membership. We conduct comprehensive evaluations on widely adopted models, such as Code Llama 7B, over the APPS and HumanEval benchmarks. The results show that our approach consistently outperforms state-of-the-art baselines, with AUC gains of up to 102%. In addition, our method exhibits strong transferability across different models and datasets, underscoring its practical utility and generalizability.**

***Index Terms*—Code LLMs, Membership Inference Attacks, Adversarial Prompts, Robustness**

I. Introduction

Large language models (LLMs) have shown remarkable success in natural language processing by learning complex semantic and syntactic patterns from large-scale text corpora [1], [2]. This success has extended to the domain of source code, where code-specific LLMs (code LLMs) trained on billions of lines of code [3] now support tasks such as code completion [4], code summarization [5], [6], and vulnerability detection [7], and are integrated into tools like GitHub Copilot [8] and AWS CodeWhisperer [9].

Despite their impressive capabilities, code LLMs remain vulnerable to a variety of security and privacy threats, including adversarial perturbations [10], data poisoning [11], [12], and privacy leakage [13]–[15]. Among these, privacy leakage is particularly concerning due to its implications for sensitive information exposure and potential legal violations, often stemming from the memorization behavior of code LLMs [16]–[18]. MIAs, which infer whether a specific code sample was included in the training set of a target code LLMs [19], provide a means to uncover and quantify such memorization, thereby helping mitigate this risk.

The practical value of successful MIAs on code LLMs is multifaceted. From a privacy perspective, leaked membership information may expose sensitive code snippets containing API keys, credentials, or personal identifiers [13]. MIAs can assess the extent of such leakage in code LLMs and help mitigate it by guiding the design of effective defenses. From a legal perspective, memorization and unauthorized reproduction of code from open-source repositories may violate license agreements (e.g., GPL, CC-BY-SA) if proper attribution is omitted [20], [21]. Employing MIAs can promote the use of policy-compliant training datasets. From an accountability perspective, membership leakage may serve as forensic evidence for identifying unauthorized training or data misuse. MIAs can thus support data governance and regulatory compliance [22].

Therefore, advancing research on MIAs is essential for promoting responsible use of code LLMs and better mitigating the privacy risks associated with model memorization. While membership inference techniques can be misused (e.g., to probe models for memorized credentials), our work is aimed exclusively at understanding and mitigating privacy risks in code models. The methods we propose are intended to support research, auditing, and regulatory compliance efforts.

Existing black- and gray-box MIAs for code models generally fall into three paradigms [19]. The first trains a substitute model to approximate the target and performs shadow-model attacks based on its behavior [23]. The second directly compares the model's generated output with the ground-truth completion, flagging exact matches or high semantic similarity as evidence of membership [14], [24]. The third analyzes differences in internal representations when the model is queried with syntactic variants of the same input (e.g., lowercase versus uppercase code), assuming that large representational shifts signal memorization [25]. However, due to their high computational overhead, fragile reliance on output similarity, and inability to capture richer memorization patterns, these strategies still leave considerable room for improvement in reliably detecting memorization in large-scale code LLMs.

In this study, we propose AdvPrompt-MIA, which infers code membership status by leveraging a series of semantics-preserving and code-specific adversarial prompts to expose model memorization. The core insight is that code completion models tend to produce more stable outputs for training samples than for unseen inputs. This behavioral stability is particularly pronounced when the input is modified with small, functionality-preserving perturbations, such as inserting dead loops or renaming variables. That means, if the input is a memorized training sample, the model's output remains largely consistent even after applying code perturbations, staying close to the original completion. In contrast, for non-member inputs, the same perturbations often lead to noticeably different outputs. This distinction between members and non-members under perturbations forms the basis of our approach, and similar patterns have also been observed in prior work on adversarial machine learning [26]–[29].

AdvPrompt-MIA operationalizes this insight by introducing five types of semantics-preserving perturbations and quantifying the model's behavioral differences in response to perturbed versus original inputs. These differences are measured using both similarity and perplexity metrics, from which we construct feature vectors that characterize the model's output patterns. The resulting features are used to train a deep learning classifier that automatically learns discriminative signals indicative of membership. This design enables our method to detect subtle behavioral shifts and reliably infer whether a given input-output pair was part of the model's training data.

We evaluate our method on the Code Llama 7B model [30] and observe substantial AUC improvements ranging from 63.8% to 102.0% over state-of-the-art baselines on APPS and HumanEval. We further demonstrate that our approach generalizes well to other code models, including Deepseek-Coder 7B [31], StarCoder2 7B [32], Phi-2 2.7B [33], and WizardCoder 7B [34]. Additionally, we conduct transferability experiments in which the MIA classifier is trained and tested across different models or datasets, showing that our method maintains strong performance under cross-model and cross-dataset settings. The main contributions are as follows.

- We propose a novel MIA framework, AdvPrompt-MIA, which leverages semantics-preserving adversarial prompts to perturb input code and employs a deep learning classifier to automatically learn discriminative patterns from the model's behavioral variations.
- We design five types of semantics-preserving perturbations, each crafted to elicit measurable differences in model behavior. Empirical results demonstrate that these perturbations effectively amplify membership signals and enhance inference accuracy.
- We conduct comprehensive experiments on multiple code LLMs and two widely used benchmarks (APPS and HumanEval), showing that our approach consistently outperforms state-of-the-art baselines across a range of models and evaluation settings.

II. Motivation and Background

A. MIA Definition and Representative Attack Methods

MIAs aim to determine whether a given data sample was used during the training of a target machine learning model. In the context of code completion, each data point is represented as a pair (x, y), where x is a partial code snippet (i.e., the input prefix), and y is its corresponding expected completion. Given black-box access to a trained code completion model M, the adversary can query the model with x and obtain the generated output $\hat{y} = M(x)$. The objective is to infer whether the pair (x, y) was present in the training dataset D_{in} of M. Formally, an MIA constructs a binary classifier G, which takes as input the triple $(x, y, \hat{y})$ and predicts a membership label:

$$G(x, y, \hat{y}) \in \{0, 1\},$$

where a prediction of 1 indicates that (x, y) is a member sample (i.e., $(x, y) \in D_{\text{in}}$), and 0 otherwise.

A commonly used baseline defines G as a heuristic rule that compares the ground-truth completion y with the model output $\hat{y}$. One typical example is GT-Match, which infers membership by checking whether $\hat{y}$ exactly matches y:

$$G_{\text{exact}}(x, y, \hat{y}) = \mathbb{1}[\hat{y} = y],$$

where $\mathbb{1}[\cdot]$ denotes the indicator function. This method assumes that if the model reproduces y exactly given x, it is likely that (x, y) was memorized during training. However, this approach often suffers from low true positive rates, particularly for large code models that may generalize to produce semantically correct but non-identical completions. In such cases, even if x was included in the training set, the model may generate $\hat{y} \neq y$ [24].

To address these limitations, many prior studies adopt a shadow model strategy [35]. In this setting, the adversary trains a local surrogate model $\hat{M}$ on data sampled from the same or a similar distribution as the target model's training set. Since the attacker has full control over $\hat{M}$, they can label any sample (x, y) as a member or non-member based on whether it was in $\hat{M}$'s training data. The resulting dataset $\mathcal{D}_{\text{attack}}$ is then used to train the attack classifier G. Formally, the adversary constructs:

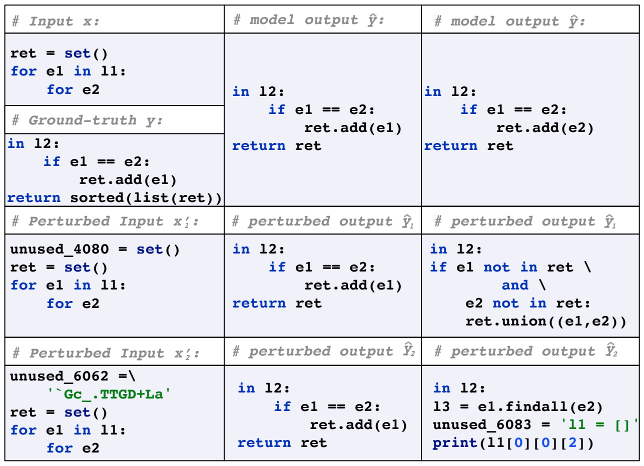
$$\mathcal{D}_{\text{attack}} = \{(x_i, y_i, \hat{M}(x_i), m_i)\}_{i=1}^{n},$$

where $m_i = 1$ if (x_i, y_i) was used to train $\hat{M}$, and $m_i = 0$ otherwise. This approach has been shown to be effective against small-scale code LLMs [19]. However, its application to larger models remains limited. Training shadow models that approximate large commercial code models is computationally expensive, and small-scale surrogates often fail to capture the behavior of large models, thereby reducing the transferability and effectiveness of the attack [13].

B. Motivation for the Proposed Method and How It Differs from Prior Work

The methods described in Section II-A share a common intuition: if a model has seen an input-output pair (x, y) during training, it is more likely to generate an output $\hat{y}$ that closely matches y. For example, GT-Match directly reflects this intuition by inferring membership based on exact string matching between $\hat{y}$ and y. Shadow-model–based approaches also follow this principle by learning to classify membership using feature representations derived from x, y, and $\hat{y}$, implicitly modeling the alignment between the generated and reference outputs. However, this intuition tends to be less pronounced for larger code LLMs. As these models are trained on massive and diverse datasets, they exhibit strong generalization and may generate completions that differ from the ground truth even for training samples. Conversely, for common functional code, they may produce outputs identical or similar to the ground truth despite having never seen the exact pair during training.

Our method is motivated by a different yet complementary observation: membership signals are more robust under semantics-preserving perturbations [29]. Specifically, when a sample (x, y) is part of the training data, the model tends to generate completions that maintain a stable relationship with y across small, functionality-preserving changes to x. This behavioral consistency reflects underlying memorization. In contrast, for non-member samples, the model's outputs are more sensitive to such perturbations, resulting in greater variation in their alignment with y. This divergence under controlled perturbations offers a fine-grained signal for distinguishing members from non-members.



```
# Input x:
ret = set()
for e1 in l1:
    for e2
```

```
# Ground-truth y:
in l2:
    if e1 == e2:
        ret.add(e1)
return sorted(list(ret))
```

```
# model output ŷ:
in l2:
    if e1 == e2:
        ret.add(e1)
return ret
```

```
# model output ŷ:
in l2:
    if e1 == e2:
        ret.add(e2)
return ret
```

```
# Perturbed Input x'_1:
unused_4080 = set()
ret = set()
for e1 in l1:
    for e2
```

```
# perturbed output ŷ_1
in l2:
    if e1 == e2:
        ret.add(e1)
return ret
```

```
# perturbed output ŷ_1
in l2:
if e1 not in ret \
        and \
    e2 not in ret:
    ret.union((e1,e2))
```

```
# Perturbed Input x'_2:
unused_6062 =\
    '`Gc_.TTGD+La'
ret = set()
for e1 in l1:
    for e2
```

```
# perturbed output ŷ_2
in l2:
    if e1 == e2:
        ret.add(e1)
return ret
```

```
# perturbed output ŷ_2
in l2:
l3 = e1.findall(e2)
unused_6083 = 'l1 = []'
print(l1[0][0][2])
```

Fig. 1. An illustrative example showing the code pair (x, y), along with the model's original and perturbed outputs when (x, y) is a memorized sample (middle column) and when it is a non-memorized sample (right column).

To further clarify this intuition, Fig. 1 presents a representative example illustrating the victim model's responses to an input x and its perturbed variants, conditioned on whether the pair (x, y) appears in the training set. As shown, the model tends to generate similar completions for the original input x in both member and non-member cases, which limits the effectiveness of GT-Match and shadow-model–based methods. In contrast, when semantics-preserving perturbations are applied, the outputs for non-member samples exhibit greater deviation from the ground truth y (right column), whereas member samples remain stable (middle column), indicating stronger memorization and reduced sensitivity to input changes.

Motivated by this example, we propose a deep learning-based method that captures both the consistency and variation in model responses to semantics-preserving perturbations, enabling more effective identification of subtle memorization patterns through aggregated observations.

III. Overview of Our Attack Methodology

This section defines the threat model in terms of the adversary's goals, knowledge, and capabilities [36], and then briefly describes our method's workflow under these assumptions.

A. Threat Model

a) Adversary's Goal: Let M be a code completion model trained on a private dataset $D_{in} = \{(x_i, y_i)\}_{i=1}^{n}$, where each pair (x_i, y_i) represents an input code snippet and its corresponding ground truth completion. Given a query input x, M produces a completion $\hat{y} = M(x)$ in a black-box fashion, without revealing any internal parameters or gradients. The attacker's goal is to determine whether a particular pair (x, y) was used in training M.

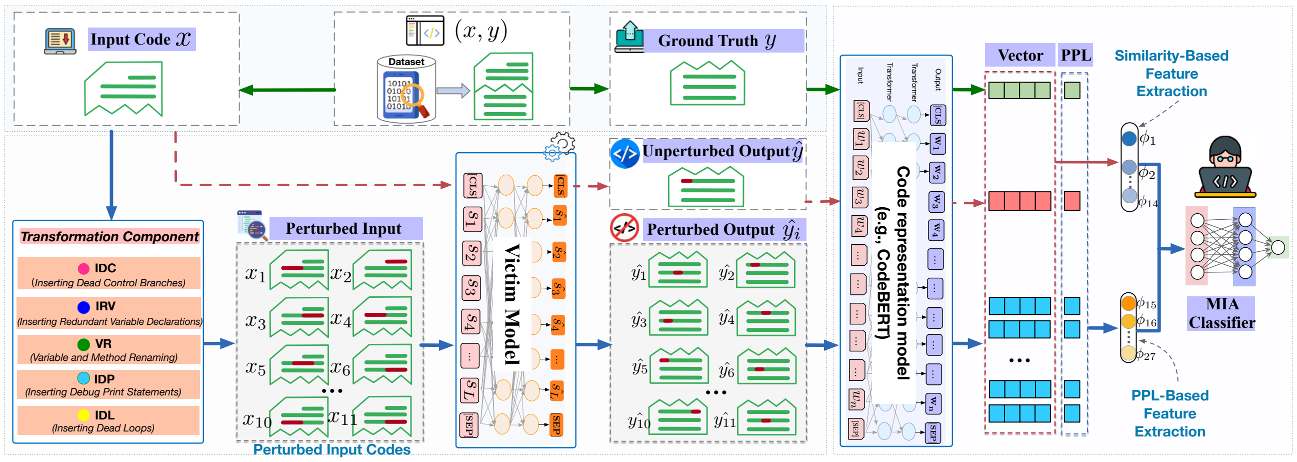
b) Adversary's Knowledge: To ensure fair comparison with prior work [23], we assume a partially informed adversary who has access to a small portion of the training dataset. Specifically, we follow the standard shadow training assumption used in [23], where the adversary knows approximately 20% of the samples in D_{in}. This setting reflects practical scenarios where models are trained on a mix of publicly available and proprietary code, making partial data exposure plausible in real-world deployments.

c) Adversary's Capabilities: The adversary is assumed to have black-box access to the target model M, which means that it can query the model with arbitrary code inputs and observe the corresponding outputs, but cannot access internal model details such as architecture, parameters, or training configuration. In our attack setting, the adversary leverages this black-box access to submit clean and perturbed versions of x and analyze the resulting completions to infer membership.

B. Overall Approach

Unlike prior methods that depend on heuristic rules or require training a surrogate model to approximate M [37], [38], our approach directly analyzes the victim model's responses to perturbed queries, thereby simplifying the overall workflow while maintaining strong attack performance. Specifically, we apply adversarial modifications to the input x and examine the variations in M's predictions to reveal membership signals.

The workflow of our method is illustrated in Fig. 2 and consists of the following steps. First, the victim model M is queried with the original input x to obtain the corresponding prediction $\hat{y}$. Next, a set of perturbed inputs x'_i is generated by applying small, semantics-preserving modifications to x, such as inserting dead code or renaming variables, ensuring that the program's intended functionality remains unchanged. Each perturbed input x'_i is then used to query M, resulting in a set of predictions $\hat{y}_i$. Subsequently, a pre-trained code embedding model (i.e., CodeBERT [3]) is employed to transform y, $\hat{y}$, and each $\hat{y}_i$ into vector representations. From these embeddings, similarity- and perplexity-based features are extracted to quantify how closely and consistently M's completions align with the true suffix y. Finally, a binary classifier G is trained on the extracted features using labeled member and non-member samples. During inference, G predicts the membership status of new queries based on their extracted features.



Input Code x
(x, y)
Dataset
Ground Truth y
Unperturbed Output $\hat{y}$
Perturbed Output $\hat{y}_i$
Transformation Component
IDC (Inserting Dead Control Branches)
IRV (Inserting Redundant Variable Declarations)
VR (Variable and Method Renaming)
IDP (Inserting Debug Print Statements)
IDL (Inserting Dead Loops)
Perturbed Input
Perturbed Input Codes
Victim Model
Code representation model (e.g., CodeBERT)
Vector
PPL
Similarity-Based Feature Extraction
PPL-Based Feature Extraction
MIA Classifier

Fig. 2. Workflow of the proposed MIA framework, AdvPrompt-MIA

IV. Details of the Proposed Method

A. Adversarial Perturbations for Membership Exposure

Adversarial perturbations are key to revealing the memorization behavior of LLMs. As discussed in Section II-B, if (x, y) is a training sample, the model's predictions remain stable under small perturbations to x; otherwise, the outputs vary more due to weaker learned associations. This behavioral divergence motivates the construction of *adversarial perturbations* that maintain program semantics while amplifying the prediction stability gap between member and non-member samples, thereby exposing memorization signals that can be exploited by a downstream classifier.

To this end, we design the following five semantics-preserving perturbations. Unlike prior adversarial attack methods that focus on imperceptible changes, our goal is to observe model behavior under meaningful input variations, regardless of whether the modifications are minimal.

1) Inserting Dead Control Branches (IDC): This transformation inserts a conditional branch with a statically false predicate at a randomly selected line in the code. The inserted branch optionally wraps either an assignment to a non-existent variable or an existing statement from x. Although this modifies the control flow and dependency structure, the runtime behavior remains unaffected since the branch condition is always false and thus never executed. We define four construction strategies, as illustrated in Fig. 3, and randomly apply two per input during our experiments, as applying all four did not lead to further performance improvements.

Taking Case 3 as an example, the IDC transformation is formally defined as follows.

$$T_{\text{IDC}} = \left\{ t_{\text{IDC}} \;\middle|\; \forall x \in \mathcal{X},\ \forall s \in S(x),\ t_{\text{IDC}}(x, s) \in \mathcal{X}' \right\} \quad (1)$$

where $S(x)$ denotes the set of statements in x.

2) Inserting Redundant Variable Declarations (IRV): This transformation injects a variable declaration that is never used in the subsequent code. The declared variable may be

# Case 1:	# Case 2:

`if "key" != "key":`
`    void_array = [''] * 50`
`    void_array[10] = 'A'`
	`if False:`
	`    void_array = [''] * 50`
	`    void_array[10] = 'A'`

# Case 3:	**# Case 4:**

`if "key" != "key":`
`    statement`
	`if False:`
	`    statement`

Fig. 3. Four strategies for constructing IDC branches

optionally initialized using either an existing variable (Case 1) or a randomly selected constant (Case 2), as shown in Fig. 4. The insertion is performed after the line where the referenced variable appears; if a constant is used, the insertion position is selected randomly. The formal definition of the IRV transformation for Case 1 is provided in Equation 2, where $V(x)$ denotes the set of variables in x.

# Case 1:	# Case 2:
`new_var = exist_var`	`new_var = 739`

Fig. 4. Two forms of the IRV transformation

$$T_{\text{IRV}} = \left\{ t_{\text{IRV}} \;\middle|\; \forall x \in \mathcal{X},\ \forall v \in V(x) :\ t_{\text{IRV}}(x, v) \in \mathcal{X}' \right\}, \quad (2)$$

3) Variable and Method Renaming (VR): This transformation replaces all occurrences of a variable or method name with a new syntactically valid identifier, typically generated by appending a randomized numeric suffix to the original name. The formal definition of the VR transformation is as follows:

$$T_{\text{VR}} = \left\{ t_{\text{VR}} \;\middle|\; \forall x \in \mathcal{X},\ \forall v \in V(x) :\ t_{\text{VR}}(x, v) \in \mathcal{X}' \right\}. \quad (3)$$

where $V(x)$ denotes the set of variable and method identifiers in x, and $v \in V(x)$ is a selected identifier. This renaming operation perturbs the token-level representation of the code while preserving its semantics, which has been shown to be effective in prior adversarial attack studies [39].

4) Inserting Debug Print Statements (IDP): This transformation, with two cases as shown in Fig. 5, inserts print statements for logging or debugging purposes, thereby increasing lexical diversity and introducing benign data-flow dependencies. In Case 1, the print statement is inserted at the beginning of the method body; in Case 2, it is placed after a variable declaration. The formal definition of the IDP transformation, corresponding to Case 2, is as follows:

`# Case 1:`	`# Case 2:`
`print("Debug: Entering method foo()")`	`print("Debug: Variable", v)`

Fig. 5. Two forms of the IDP transformation

$$T_{\text{IDP}} = \Big\{ t_{\text{IDP}} \;\Big|\; \forall x \in \mathcal{X},\ \forall v \in V(x):\ t_{\text{IDP}}(x, v) \in \mathcal{X}' \Big\}, \quad (4)$$

where v is a variable from x used to establish a data-flow dependency through the inserted print statement.

5) Inserting Dead Loops (IDL): This transformation introduces three types of loop conditions that are statically false (i.e., the loops will never execute), optionally populated with a no-op body such as a `print statement`, `pass`, or an `unused assignment`, as illustrated in Fig. 6. The loop is inserted at a randomly selected line in the code to preserve semantics while increasing structural complexity.

`# Loop conditions:`	`# Loop bodies:`
`for _ in range(5, 3):`	`print("Debug: Entering loop")`
`while "key" != "key":`	`pass`
`while False:`	`unused_var = 0`

Fig. 6. Loop conditions and bodies used in the IDL transformation

The IDL transformation is formally defined as follows:

$$T_{\text{IDL}} = \Big\{ t_{\text{IDL}} \;\Big|\; \forall x \in \mathcal{X},\ t_{\text{IDL}}(x) \in \mathcal{X}' \Big\}. \quad (5)$$

Among the above five perturbation strategies, some are inspired by prior adversarial attack works (e.g., VR in ALERT [40] and IDP in DIP [41]). Based on the five perturbation methods, we construct a set of 11 semantically equivalent variants $\mathcal{X}'$ for each input x using Algorithm 1. As shown in the algorithm, each variant is perturbed with one single transformation. In particular, the resulting set $\mathcal{X}'$ includes two variants each from IDC, IRV, VR, and IDP, and three from IDL. All variants preserve the original program semantics, as they do not alter the program's functional behavior. This is confirmed by our manual verification of representative examples from each transformation.

Algorithm 1: Generate the 11 perturbations of x.

```
Input  : code $x$ with lines $L$, variables $V$, methods $M$
Output: perturbed code $x'$
copy $x_{\text{orig}} \leftarrow x$; let $\mathcal{X}' \leftarrow \emptyset$
foreach type in {IDC, IRV, VR, IDP, IDL} do
    if type = IDC then
        pick $f_1, f_2 \in$ IDC_forms; for each $f \in \{f_1, f_2\}$,
          apply $t^f_{\text{IDC}}$ to $x_{\text{orig}}$, appending result to $\mathcal{X}'$
    else if type = IRV then
        for each $g \in$ IRV_forms, apply $t^g_{\text{IRV}}$ to $x_{\text{orig}}$,
          appending result to $\mathcal{X}'$
    else if type = VR then
        pick $h_1, h_2 \in$ VR_forms; for each $h \in \{h_1, h_2\}$,
          apply $t^h_{\text{VR}}$ to $x_{\text{orig}}$, appending result to $\mathcal{X}'$
    else if type = IDP then
        pick $p_1, p_2 \in$ IDP_forms; for each $p \in \{p_1, p_2\}$,
          apply $t^p_{\text{IDP}}$ to $x_{\text{orig}}$, appending result to $\mathcal{X}'$
    else
        for each $q \in$ IDL_forms, apply $t^q_{\text{IDL}}$ to $x_{\text{orig}}$,
          appending result to $\mathcal{X}'$
end
return $\mathcal{X}'$
```

B. Training MIA Classifiers

1) Feature Extraction: For each original input x and its perturbed variants $\{x'_i\}_{i=1}^{11}$, we obtain the model outputs y and $\{\hat{y}_i\}_{i=1}^{11}$. We then compute two types of deviations from the reference output y: (i) similarity scores and (ii) perplexity values. These measures capture how the model's predictions vary in response to perturbations and form the basis of the feature vector used for membership inference.

a) Textual Similarity: We tokenize and embed each code snippet using CodeBERT [3] to obtain a token sequence of length L and an embedding matrix of size $L \times 768$. Applying average pooling yields a single 768-dimensional vector. Let $v(\cdot)$ denote this embedding process; for any two snippets y and $\hat{y}$, their cosine similarity is defined as:

$$\operatorname{sim}(y, \hat{y}) \;=\; \frac{v(y) \cdot v(\hat{y})}{\|v(y)\| \|v(\hat{y})\|}. \quad (6)$$

b) Perplexity: To quantify how anomalous a model's output is under perturbation, we employ perplexity. For a token sequence $w_1^N = (w_1, \ldots, w_N)$, the model assigns conditional probabilities $P(w_i \mid w_{<i})$, and the perplexity of the sequence is defined as:

$$\text{PPL}(w_1^N) \;=\; \exp\!\Big(-\tfrac{1}{N} \sum_{i=1}^{N} \ln P(w_i \mid w_{<i})\Big) \quad (7)$$

where lower PPL indicates higher confidence. To evaluate the relative change in perplexity under perturbations, we normalize each perplexity score with respect to the unperturbed output:

$$\text{PPL}'(\hat{y}_i) \;=\; \frac{\text{PPL}(\hat{y}_i) \;-\; \text{PPL}(\hat{y})}{\text{PPL}(\hat{y})} \quad (8)$$

c) Feature Vector Construction: For each sample (x, y), we construct a 27-dimensional feature vector that captures both similarity- and perplexity-based cues relevant to membership. Specifically, the vector includes the similarity between the ground-truth suffix y and the model's unperturbed output $\hat{y}$, followed by the similarities between y and the 11 adversarially perturbed completions $\hat{y}_i$. To summarize these 12 similarity values, we include their mean $\mu(\text{sim})$ and standard deviation $\sigma(\text{sim})$. In addition, the vector incorporates the standardized perplexity scores $\text{PPL}'(\hat{y}_i)$ for each of the 11 perturbed outputs, along with their mean $\mu(\text{PPL}')$ and standard deviation $\sigma(\text{PPL}')$. Formally, the resulting feature vector is defined as:

$$\phi(x, y) = \begin{bmatrix} \text{sim}(y, \hat{y}), \\ \text{sim}(y, \hat{y}_1),\ \ldots,\ \text{sim}(y, \hat{y}_{11}),\ \mu(\text{sim}),\ \sigma(\text{sim}), \\ \text{PPL}'(\hat{y}_1),\ \ldots,\ \text{PPL}'(\hat{y}_{11}),\ \mu(\text{PPL}'),\ \sigma(\text{PPL}') \end{bmatrix}.$$

The feature vector captures behavioral differences, serving as the central mechanism that drives our method's performance.

2) Model Training and Inference: To perform membership inference, we train a multilayer perceptron (MLP) classifier using the high-dimensional feature representations $\phi(x, y)$ constructed in the previous step. Let $D = 27$ denote the input layer dimension, which matches the dimensionality of the feature vector. The network comprises three hidden layers of size 512 and an output layer that produces logits in $\mathbb{R}^C$, where $C = 2$ for binary membership classification. Each hidden layer performs a linear mapping with ReLU activation:

$$\mathbf{h}_i = \text{ReLU}\big(\mathbf{W}_i\, \mathbf{h}_{i-1} + \mathbf{b}_i\big), \quad \mathbf{h}_0 = \phi(x, y), \tag{9}$$

followed by a dropout operation to mitigate overfitting. After the final hidden layer $\mathbf{h}_3$, the output logits are computed as

$$\mathbf{z} = \mathbf{W}_{\text{out}}\, \mathbf{h}_3 + \mathbf{b}_{\text{out}}, \quad \mathbf{z} \in \mathbb{R}^C. \tag{10}$$

The classifier is trained using standard cross-entropy loss.

At inference time, given a data pair (x, y), we first obtain the unperturbed output $\hat{y} = M(x)$ from the victim model. We then apply semantics-preserving perturbations to x to generate a set of modified inputs $\{x'_i\}$ and collect their corresponding outputs $\{\hat{y}_i\}$. Based on these outputs and the reference y, we compute the feature vector $\phi(x, y)$ as described in Section IV-B. Finally, this feature vector is fed into the trained MLP classifier to determine the membership status. This end-to-end pipeline relies solely on the victim model's outputs under controlled perturbations and the feature vectors designed to capture output variations indicative of memorization.

V. Experiment Design

We aim to answer the following research questions (RQs)[1]:

RQ1: To what extent does the proposed MIA method outperform state-of-the-art baselines?

RQ2: Which perturbation strategies and feature components contribute most to attack performance?

RQ3: To what extent can the proposed method generalize to other code LLMs beyond Code Llama 7B?

RQ4: Can the proposed method train an MIA classifier on one model that transfers effectively to other target models?

RQ5: To what extent does the proposed method generalize with varying or no knowledge of target training datasets?

[1]Code is available at https://github.com/YuanJiangGit/MIA_Adv

A. Dataset

Code generation (or code completion) is a core capability of code LLMs and provides a representative setting for studying membership inference [23]. We adopt APPS and HumanEval, two widely used benchmarks, as evaluation datasets for assessing MIA performance. To investigate potential leakage between our evaluation benchmarks and the pre-training corpora of the target models, we perform a text-based matching analysis. Among the five victim models considered in this paper (introduced in Section V-D), StarCoder2 is the only one with publicly traceable pre-training data (The Stack v2, 32.1TB, 1.15B files) [32]. We therefore conduct matching experiments against this corpus and find no exact overlaps with the code samples from HumanEval and APPS, reducing the likelihood of data leakage. Moreover, both benchmarks are widely adopted as official test sets for evaluating the functional correctness of target models [30]–[32], further supporting their exclusion from pre-training data to ensure fair evaluation.

APPS [42] contains 5 000 training tasks and 5 000 testing tasks. For each task, we extract a single reference implementation and discard any tasks without valid solutions. This yields 5 000 training examples for constructing the victim model's training dataset D_{in}, and 3,765 testing examples, which serve as the basis for the non-member dataset D_{out}. HumanEval [43] consists of 164 hand-crafted programming problems. Following a similar setup, we randomly divide the problems into two disjoint subsets of equal size. One subset (82 problems) is used to construct D_{in}, while the remaining 82 problems form D_{out}. Each problem is associated with a single reference solution. Note that D_{in} is used to fine-tune publicly released code LLMs rather than train from scratch.

We adopt a *partial knowledge* threat model, in which the adversary has access to only a small portion of the training data. Specifically, 20% of D_{in} is assumed to be known to the adversary and is used as the positive class when training the MIA classifier. An equal number of non-member samples is randomly drawn from D_{out} to form a balanced training set. This setting, including the exact ratio (i.e., 20%), is consistent with prior work [23]. For evaluation, the same sampling strategy is applied to the remaining 80% of D_{in} and to the rest of D_{out}, ensuring that training and evaluation are performed on disjoint data splits derived from the original benchmarks.

The data preparation process is illustrated in Fig. 7, where D_{in} and D_{out} denote the training and testing sets, used to simulate member and non-member samples, respectively.

B. Evaluation Metrics

Consistent with prior work [23], [38], we adopt three metrics to evaluate the performance of the MIA classifier: True Positive Rate (TPR), False Positive Rate (FPR), and Area Under the ROC Curve (AUC). These are defined as: TPR = $\text{TP}/(\text{TP} + \text{FN})$, FPR = $\text{FP}/(\text{FP} + \text{TN})$, and AUC = $\int_0^1 \text{TPR}(\tau)\, \text{dFPR}(\tau)$, where TP, FP, TN, and FN represent true positives, false positives, true negatives, and false negatives, respectively, and τ denotes the decision threshold.

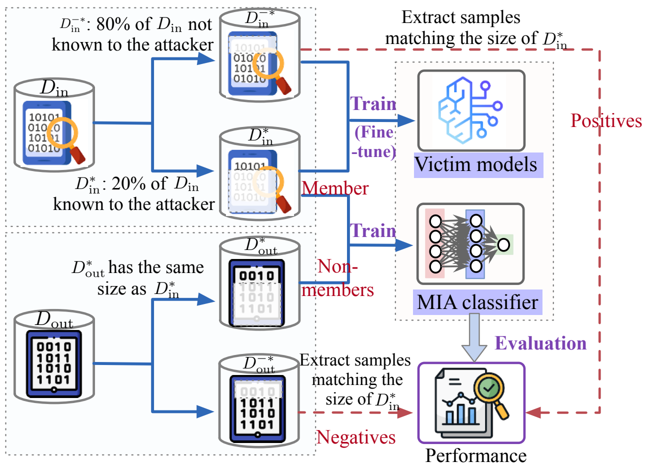

D_{in}^{-*}: 80% of D_{in} not known to the attacker
D_{in}^{-*}
D_{in}
D_{in}^{*}
D_{in}^{*}: 20% of D_{in} known to the attacker
Extract samples matching the size of D_{in}^{*}
Train (Fine-tune)
Victim models
Positives
Member
Train
MIA classifier
Non-members
D_{out}^{*}
D_{out}^{*} has the same size as D_{in}^{*}
D_{out}
D_{out}^{-*}
Extract samples matching the size of D_{in}^{*}
Evaluation
Negatives
Performance

Fig. 7. Workflow for dataset construction used in experiments

We choose these metrics because TPR reflects the attacker's ability to correctly identify member samples (i.e., attack power), FPR captures the rate of incorrect membership predictions (i.e., attack error), and AUC provides a threshold-independent summary of the trade-off between them, offering an overall assessment of classifier performance.

C. Baseline Methods

We compare our method against four representative black- or gray-box MIA baselines recently proposed for evaluating membership inference risks in target code LLMs, where GOTCHA and GT-Match have been detailed in Section II-A.

GOTCHA [23] trains a shadow model on a small known portion of the training data, uses its outputs to train a binary classifier, and transfers it to the victim model for inference.

GT-Match [20] labels a sample as a member if the model generates a suffix that exactly matches the ground truth, leveraging verbatim memorization.

PPL-Rank [44] ranks samples by token-level perplexity and predicts the top 50% as members, assuming lower perplexity for training data.

BUZZER-b [45] applies lowercase perturbations and measures the change in representation, detecting memorization based on casing sensitivity.

D. Evaluation Setting

We evaluate our proposed MIA method on the widely used Code Llama 7B model [30]. To examine its generalizability, we further conduct experiments on additional code LLMs, including Deepseek-Coder 7B [31], StarCoder2 7B [32], [46], Phi-2 2.7B [33], and WizardCoder 7B [34]. Following prior work and common practices [47], we fine-tune all code LLMs for 5 epochs using a learning rate of 2e-5, Adam optimizer (weight decay 1e-2, ϵ=1e-8), batch size of 2, and 4-step gradient accumulation. Phi-2 (2.7B) is fully fine-tuned, while all 7B models are fine-tuned using LoRA ($r = 16$, α=32, dropout=0.1). We also perform a random search over alternative settings and find that this configuration consistently yields the best overall functional correctness across all models.

Our MIA classifier is implemented as a fully connected three-layer neural network with a hidden size of 512. It is optimized using the Adam optimizer [48], with a learning rate of 1e-3 and no weight decay, and trained for 25 epochs using randomly shuffled mini-batches of size 4. The model architecture and hyperparameters are selected via random search on the HumanEval dataset. All experiments are conducted on two NVIDIA A6000 GPUs, each equipped with 48 GB of memory.

For baseline comparisons, we use the original implementations where available (e.g., GOTCHA and BUZZER-b), and re-implement the methods when necessary. For rank-based approaches such as PPL-Rank and BUZZER-b, we classify the top 50% of the samples as members, consistent with the balanced member/non-member split in the evaluation datasets and the setting used in prior work [23].

VI. Experiment Results

A. RQ1: To what extent does the proposed MIA method outperform state-of-the-art baselines?

To assess the effectiveness of our proposed MIA method, we compare it against four baselines using the Code Llama 7B model. All experiments are conducted on the APPS and HumanEval datasets (see Section V-A for dataset details), and the baselines are described in Section V-C. For each method, we report TPR, FPR, and AUC.

Table I presents the performance comparison between our method, AdvPrompt-MIA, and existing baselines. As shown, our method consistently outperforms all baselines across both datasets, demonstrating its effectiveness regardless of dataset scale. Specifically, on the relatively large-scale APPS dataset, AdvPrompt-MIA achieves AUC improvements of 67.2%, 86.3%, 63.8% and 90.0% over GOTCHA, GT-Match, PPL-Rank and BUZZER-b, respectively. The performance gap is even more pronounced on the smaller HumanEval dataset, where our method achieves 69.6%, 73.2%, 102.1% and 94.0% higher AUC than the same baselines.

TABLE I
Comparison of the Proposed Method and Baselines for Membership Inference Attacks on Code Llama 7B

Dataset	Method	TPR↑	FPR↓	AUC↑
HumanEval	GOTCHA	0.35	0.40	0.58
	GT-Match	0.55	0.40	0.56
	PPL-Rank	0.40	0.55	0.48
	BUZZER-b	0.48	0.50	0.50
	AdvPrompt-MIA	**0.85**	**0.05**	**0.97**
APPS	GOTCHA	0.36	0.40	0.56
	GT-Match	0.56	0.58	0.51
	PPL-Rank	0.58	0.42	0.58
	BUZZER-b	0.52	0.50	0.50
	AdvPrompt-MIA	**0.90**	**0.14**	**0.95**

Our method achieves strong performance primarily by leveraging a series of semantics-preserving perturbations to amplify membership signals. It then constructs feature representations that capture the consistency or variability in the model's responses to these perturbations. These features enable the deep learning classifier to distinguish member from non-member instances by identifying robust behavioral patterns, even in the absence of exact output matches.

In addition, we observe that all four baseline methods exhibit limited effectiveness on both datasets. GOTCHA suffers from severe overfitting because only 20% of the victim's training data is used as member samples; the resulting shadow model fails to mimic the victim model's behavior, which prevents the MIA classifier from learning meaningful features and leads to a high FPR and limited effectiveness. GT-Match exhibits similar performance to GOTCHA and remains ineffective. Although this method has been widely used to detect the leakage of sensitive information such as API keys or credentials [24], it struggles to accurately infer the membership status of functional code, which is the primary content of the APPS and HumanEval datasets. PPL-Rank also fails to distinguish members from non-members, likely because perplexity alone cannot reliably indicate memorization; in fact, recent work shows that LLMs can generate code with higher quality than human-written samples [49]. BUZZER-b, while introducing input perturbations to detect memorization, applies only a single perturbation, i.e., converting tokens to uppercase, which may be insufficient to trigger observable differences, especially in models robust to such surface input changes.

Conclusion: AdvPrompt-MIA outperforms state-of-the-art baselines on both datasets, demonstrating that semantics-preserving perturbations, together with features derived from the resulting outputs, effectively amplify memorization signals and enable more accurate MIA.

B. RQ2: Which perturbation strategies and feature components contribute most to attack performance?

To assess the contribution of each component in our approach, we perform an ablation study on both the five semantics-preserving perturbation strategies and the 27-dimensional feature vector introduced in Section IV. Specifically, we evaluate two settings: (i) removing one feature dimension at a time, and (ii) disabling one perturbation strategy at a time, while retraining the MIA classifier for each variant. All other experimental configurations are kept consistent with those used in Section VI-A. The effect of omitting individual feature dimensions is illustrated in Fig. 8, and the performance impact of excluding each perturbation is shown in Fig. 9.

As shown in Fig. 8, our full method consistently outperforms all ablated variants in terms of AUC, with performance drops ranging from 2.1% to 10.3% on HumanEval and from 1.1% to 4.2% on APPS when individual features are removed. This result demonstrates the contribution of each feature dimension to the overall effectiveness of the MIA classifier. Notably, the 14th feature, $\sigma(\text{sim})$, is the most impactful, as its removal leads to the largest average performance drop across both datasets. This finding suggests that the variability in similarity scores plays a key role in distinguishing member

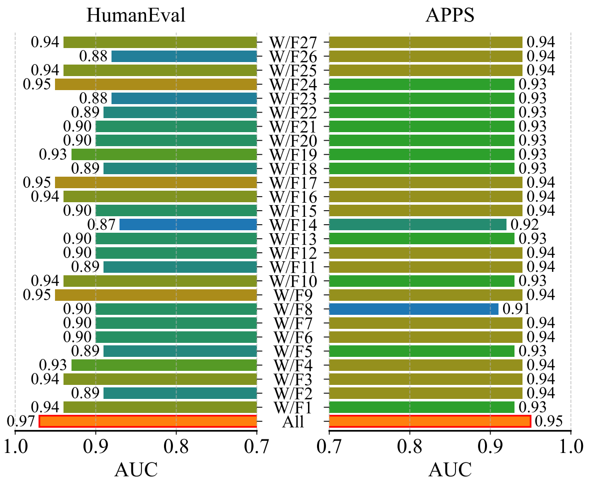


Fig. 8. AUC between our method (using all features) and its variants with each of the 27 features removed, where W/Fi denotes that the i-*th* feature is excluded when constructing the feature vectors for the MIA classifier.

from non-member samples, as it reflects the degree of output stability. The result is consistent with our hypothesis that member samples yield more stable completions under perturbations, while non-member samples exhibit greater variability.

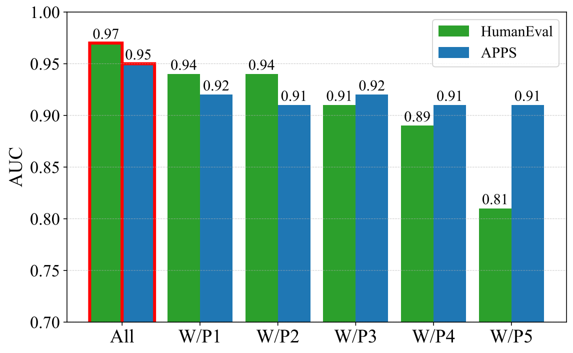


Fig. 9. AUC between our method (i.e., All) and its variants with each of the five perturbations removed, where W/Pi denotes that the i-*th* perturbation is excluded during adversarial prompt generation.

Fig. 9 further illustrates the contribution of each perturbation strategy by reporting the AUC when each is individually removed. The results show performance drops ranging from 3.1% to 16.5% on HumanEval and from 3.2% to 4.2% on APPS. Among all perturbations, the fifth strategy (i.e., IDL) leads to the most significant degradation when removed. This may be because IDL introduces structurally valid yet semantically inert code blocks, which more effectively expose discrepancies in the model's generalization behavior between member and non-member inputs. These findings underscore the importance of both perturbation diversity and feature expressiveness in enabling reliable membership inference.

Conclusion: The high performance of our method is attributed to the individual contributions of each feature dimension in the designed vector and the effectiveness of the proposed perturbation strategies.

C. *RQ3: To what extent can the proposed method generalize to other code LLMs beyond Code Llama 7B?*

In RQ1, we demonstrate that our method achieves superior performance compared to state-of-the-art baselines on the Code Llama 7B model. To further assess its generalizability, we evaluate the proposed approach on four additional widely used code LLMs: Deepseek-Coder 7B [31], StarCoder2 7B [32], Phi-2 2.7B [33], and WizardCoder 7B [34]. For each victim model, we conduct experiments on both the HumanEval and APPS benchmarks, following the same data preparation and evaluation protocol described in Section V-A. The experimental results are summarized in Table II.

TABLE II
EFFECTIVENESS OF THE PROPOSED METHOD ACROSS MULTIPLE CODE LLMS ON HUMANEVAL AND APPS

Dataset	**Victim model**	**TPR↑**	**FPR↓**	**AUC↑**
HumanEval	Deepseek-Coder 7B	0.80	0.10	0.91
	StarCoder2 7B	0.75	0.10	0.92
	Phi-2 2.7B	0.90	0.25	0.92
	WizardCoder 7B	0.75	0.05	0.95
APPS	Deepseek-Coder 7B	0.82	0.19	0.89
	StarCoder2 7B	0.84	0.16	0.91
	Phi-2 2.7B	0.79	0.21	0.85
	WizardCoder 7B	0.76	0.25	0.81

As shown in Table II, our method, AdvPrompt-MIA, consistently exhibits strong attack performance across all evaluated code models. For example, on HumanEval, it achieves AUC scores above 0.91, indicating robust generalization. This robust performance further supports our core assumption: for diverse code LLMs, memorized training samples tend to yield stable model outputs under small, semantics-preserving perturbations, whereas non-member samples result in more significant variations. These behavioral differences provide discriminative signals that enable effective membership inference.

AdvPrompt-MIA is explicitly designed to leverage this assumption by capturing perturbation-induced behavioral cues. Through a combination of semantics-preserving code transformations and a learning-based classification framework, our method automatically identifies internal consistency signals indicative of memorization, without requiring access to the model's architecture or training details. This design enables AdvPrompt-MIA to generalize well and maintain stable performance across a wide range of code LLMs.

Conclusion: AdvPrompt-MIA generalizes well across diverse code LLMs, consistently achieving strong performance regardless of the target model's internal design.

D. *RQ4: Can the proposed method train an MIA classifier on one model that transfers effectively to other target models?*

To further assess the transferability of our method and examine whether perturbation-induced features generalize across different code LLMs, we conduct cross-model experiments. Specifically, we select pairs of models from Code Llama and those evaluated in RQ3. For each experiment, we use one model to generate training vectors for the MIA classifier using our method, and directly apply the trained classifier to perform membership inference on another model, without any retraining or adaptation. All experiments are conducted on the HumanEval and APPS benchmarks, following the same data preparation process described in Section V-A. For each source-target model pair, we report AUC scores, as shown in Fig. 10.

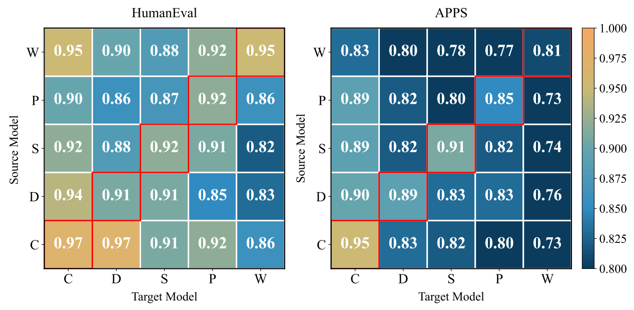


Fig. 10. Cross-model AUC scores of MIA classifiers trained on source models and evaluated on target models. C, D, S, P, and W refer to Code Llama, Deepseek-Coder, StarCoder2, Phi-2, and WizardCoder, respectively.

As shown in Fig. 10, our method consistently achieves strong performance across all model pairs on both datasets, with the best AUC reaching 0.97 when the MIA classifier is trained on Code Llama and evaluated on Deepseek-Coder on HumanEval and 0.90 when trained on Deepseek-Coder and evaluated on Code Llama on APPS. These results suggest that the perturbation-induced feature space learned from one model transfers well to others, even when the models differ significantly in architecture or training details.

We attribute this strong transferability to the shared behavioral property of code LLMs. Specifically, for member samples, semantics-preserving perturbations result in stable outputs that remain close to the ground-truth completion, whereas for non-members, the same perturbations cause greater variability. Our method captures these behavioral patterns by constructing feature vectors from the outputs of perturbed inputs. The fact that a classifier trained on one model can accurately infer membership on another suggests that these feature vectors, despite originating from different models, occupy a similar space. This similarity arises not from architectural alignment, but from the underlying consistency in how different LLMs respond to membership under perturbation.

The similar behavioral patterns exhibited by different models under perturbations allow AdvPrompt-MIA to generalize well across models. However, subtle differences in how individual models respond to the same perturbations can cause certain models to achieve higher performance. For example, the proposed semantics-preserving perturbations tend to induce more regular and stable output behaviors on Code Llama, likely owing to its large-scale and diverse pre-training data, whereas such effects are comparatively weaker on other models. These clearer behavioral patterns, when encoded into our feature vectors, allow the MIA classifier to more easily distinguish whether its input is a member sample, thereby improving inference performance even when the classifier is trained on other models. This explains Fig. 10, where (1) some target models (e.g., Code Llama) consistently outperform others, and (2) in rare cases—1 of 20 on HumanEval and 3 on APPS—the transfer setting yields higher AUC than the corresponding non-transfer baseline (e.g., Deepseek-Coder→Code Llama vs. Deepseek-Coder→Deepseek-Coder).

To verify whether the favorable performance of our method on certain target models stems from the specific perturbations employed, we take the IDL transformation as an example and revise it by replacing the simple loop body with more sophisticated code snippets, provided in the replication package[2] for brevity. The revised IDL remains semantics-preserving, as all conditions are false and loop bodies are never executed. We then compare the performance changes of our method on two target models, Code Llama and WizardCoder, before and after this modification. These two are chosen because they represent the overall best- and worst-performing target models on APPS. As shown in Fig. 11, performance decreases on the target model Code Llama but improves significantly on WizardCoder. These findings show that although our method performs well across all victim models, perturbations affect them differently, highlighting adaptive model-specific perturbation optimization as a promising direction for future work.

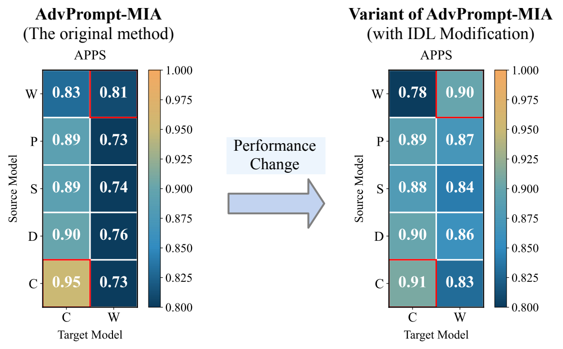


Fig. 11. Cross-model performance changes of our method after modifying only the IDL transformation (C, D, S, P, and W refer to Code Llama, Deepseek-Coder, StarCoder2, Phi-2, and WizardCoder, respectively).

Conclusion: AdvPrompt-MIA effectively captures transferable behavioral patterns, enabling a classifier trained on one model to infer membership on unseen models with strong performance. However, its transferability on specific targets is affected by the applied transformations.

E. RQ5: To what extent does the proposed method generalize with varying or no knowledge of target training datasets?

Our method AdvPrompt-MIA assumes that approximately 20% of the member samples used to train the victim model are available to the attacker. This assumption is plausible in real-world settings, where code LLMs are often trained on a mixture of public repositories and proprietary code. Moreover, this partial knowledge setup is consistent with prior work [23]. To evaluate the sensitivity of our method to the adversary's knowledge, we vary the ratio of known data from 5% to 25% in increments of 5% and conduct experiments on Code Llama 7B using APPS, which is larger than HumanEval and more suitable for small known-data settings. The results in Fig. 12 show that performance improves by about 0.05 AUC on average for every 5% increase in known data between 5% and 20%, then stabilizes beyond 20%. Even with only 5% known data, AdvPrompt-MIA achieves 0.79 AUC, demonstrating its effectiveness under limited adversary knowledge. This robustness arises from the clear behavioral differences exhibited by code LLMs when handling member and non-member samples under the designed perturbations, which can be reliably captured even with a small amount of known data.

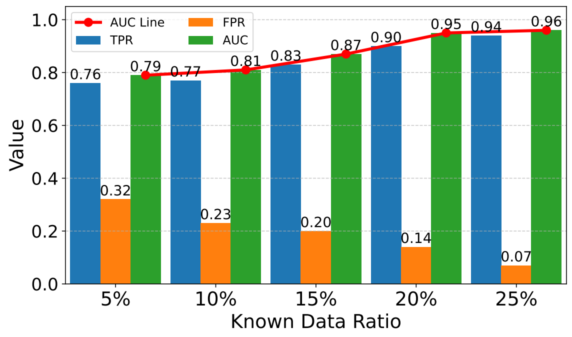


Fig. 12. Sensitivity of AdvPrompt-MIA to Known Data Proportions.

To further assess the generalizability of our method, we also consider a more challenging setting in which the attacker has no access to member samples from the target dataset. Specifically, we evaluate whether an MIA classifier trained on one dataset via our method can be transferred to a different dataset, while keeping the target code LLM unchanged. For each victim model, the classifier is trained using member and non-member samples from one dataset (e.g., HumanEval), and evaluated on another (e.g., APPS). The AUC scores across all training–evaluation dataset pairs are summarized in Table III.

As shown in Table III, our method remains effective even without access to member samples from the target dataset. For example, when targeting Code Llama, training the MIA classifier on APPS and evaluating it on HumanEval yields an AUC of 0.94, close to the within-dataset result (i.e., training and evaluating on HumanEval). This demonstrates the strong cross-dataset transferability of AdvPrompt-MIA.

Moreover, we observe an asymmetric trend: when the MIA classifier is trained on APPS and tested on HumanEval, performance is consistently better than the reverse. This may be attributed to the larger size and diversity of the APPS dataset, which provides a richer training signal. These findings suggest that, in practice, when the target dataset is unknown, training the MIA classifier on a larger and more diverse dataset can lead to better cross-dataset inference performance.

[2]https://github.com/YuanJiangGit/MIA_Adv/blob/dev/Modified_IDL.md

TABLE III
AUC PERFORMANCE OF MIA CLASSIFIER WITH TRAINING AND EVALUATION PERFORMED ON DIFFERENT DATASETS

Victim models	Training Dataset	Evaluation Dataset	AUC↑
Code Llama 7B	HumanEval	HumanEval	0.97
	HumanEval	APPS	0.78
	APPS	HumanEval	0.94
Deepseek-Coder 7B	HumanEval	HumanEval	0.91
	HumanEval	APPS	0.78
	APPS	HumanEval	0.86
StarCoder2 7B	HumanEval	HumanEval	0.92
	HumanEval	APPS	0.74
	APPS	HumanEval	0.95
Phi-2 2.7B	HumanEval	HumanEval	0.92
	HumanEval	APPS	0.73
	APPS	HumanEval	0.85
WizardCoder 7B	HumanEval	HumanEval	0.95
	HumanEval	APPS	0.68
	APPS	HumanEval	0.87

Conclusion: AdvPrompt-MIA demonstrates robustness under limited adversary knowledge and shows strong cross-dataset transferability, which can be further enhanced by training the MIA classifier on larger datasets.

VII. Related Work

Existing work [16]–[18], [50], [51] has investigated the memorization behavior of code LLMs to reveal potential risks of data leakage. Although MIAs are well explored in machine learning [29], [52], they are still emerging for code LLMs. Following [19], we classify them as gray- or black-box attacks.

a) Gray-Box MIA: Gray-box adversaries cannot access the model's architecture, parameters, or gradients [25], [53], but may have partial access to training data (e.g., from public code repositories). A common strategy in this setting is the *shadow model* technique: the attacker trains one or more surrogate models on datasets with known membership labels, collects features (e.g., code representations) from these models, and then trains a binary inference classifier [23], [25]. At inference time, the classifier uses analogous features extracted from the victim model to predict membership. The success of this approach hinges on how well the shadow models mimic the victim; training large shadow models is computationally expensive, and smaller surrogates may poorly approximate a large model, limiting transferability [13].

b) Black-Box MIA: In the black-box scenario, adversaries can only query the victim model and observe generated completions [14], [24]. Existing black-box MIAs employ methods that analyze completion fidelity, such as exact or fuzzy matching of predicted suffixes [24], and statistical metrics including perplexity, perplexity ratios, and average perplexity, leveraging the observation that member examples typically yield lower perplexity and more accurate completions [14]. Other approaches probe model sensitivity to small perturbations: for instance, masking individual tokens and verifying whether the masked predictions match the original tokens [54], or comparing outputs on semantically equivalent code variants (e.g., lowercase versus uppercase) to detect disparities indicative of memorization [25], [29]. Additionally, carefully crafted prompts can induce privacy leaks by coaxing the model to reproduce memorized code snippets, although such prompt engineering often requires manual intervention and may not generalize across different models or tasks [13].

AdvPrompt-MIA adopts a gray-box setting with limited training data access [23]. Unlike existing gray-box methods that rely on surrogate models, our method leverages deep learning to directly capture and exploit behavioral differences under perturbations and achieve higher inference performance.

VIII. Threats to Validity

Threats to Internal Validity. Internal validity concerns the extent to which the study results are free from bias. Our method leverages five predefined semantics-preserving code perturbations to induce model output variations. The findings may be specific to these transformations, and other perturbations could yield different results. To mitigate this threat, we selected several perturbations (e.g., VR and IDP) inspired by and commonly used in prior adversarial code analysis work. Additionally, we use CodeBERT to compute similarity between perturbed and original outputs, a key step in feature vector construction. We acknowledge that alternative code models or similarity metrics may affect the attack's effectiveness and will explore them in future work.

Threats to External Validity. External validity concerns the generalizability of our findings. Our evaluation focuses on widely used code models up to 7B parameters due to hardware constraints, though larger models (e.g., 34B) are increasingly adopted and remain unexplored. In addition, all experiments are conducted on unprotected models, whereas those with defense mechanisms may behave differently. Furthermore, our threat model targets the common real-world scenario of private, task-specific fine-tuning, where memorization risk is acute and existing MIAs underperform. Extending to pretraining or post-training (e.g., RLHF/RLVR) is important but nontrivial due to different training paradigms. Future work will extend and refine AdvPrompt-MIA to apply to these scenarios.

IX. Conclusion

In this paper, we propose AdvPrompt-MIA, a novel MIA method that leverages a series of semantics-preserving adversarial perturbations to probe the model's behavior. By extracting and learning from perturbation-induced output variations, our method achieves robust and accurate membership inference across different models and datasets. Experimental results show that AdvPrompt-MIA consistently outperforms state-of-the-art baselines and generalizes well in both cross-model and cross-dataset settings.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This research was supported by the National Natural Science Foundation of China under Grants No. 62302125 and 62272132, and by a Tier 1 grant funded by the Ministry of Education in Singapore (22-SIS-SMU-099 [C220/MSS23C017]).

References

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019, pp. 4171–4186.

[2] C. Raffel, N. Shazeer, A. Roberts *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020.

[3] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," 2020.

[4] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[5] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," in *Proceedings of the IEEE/ACM 46th international conference on software engineering*, 2024, pp. 1–13.

[6] J. Zhu, Y. Miao, T. Xu, J. Zhu, and X. Sun, "On the effectiveness of large language models in statement-level code summarization," in *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2024, pp. 216–227.

[7] Y. Jiang, Y. Zhang, X. Su, C. Treude, and T. Wang, "Stagedvulbert: Multi-granular vulnerability detection with a novel pre-trained code model," *IEEE Transactions on Software Engineering*, pp. 1–18, 2024.

[8] "GitHub Copilot: Your ai pair programmer," https://copilot.github.com/, 2021, accessed: 2024-12-04.

[9] "Amazon CodeWhisperer: Free ai coding companion," https://aws.amazon.com/codewhisperer/, 2023, accessed: 2024-12-04.

[10] S. Liu, D. Cao, J. Kim, T. Abraham, P. Montague, S. Camtepe, J. Zhang, and Y. Xiang, "{EaTVul}:{ChatGPT-based} evasion attack against software vulnerability detection," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 7357–7374.

[11] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1559–1575.

[12] D. Cotroneo, C. Improta, P. Liguori, and R. Natella, "Vulnerabilities in ai code generators: Exploring targeted data poisoning attacks," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 280–292.

[13] 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.

[14] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, "Unveiling memorization in code models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[15] C. Han, Z. Deng, W. Ma, X. Zhu, M. Xue, T. Zhu, S. Wen, and Y. Xiang, "Codebreaker: Dynamic extraction attacks on code language models," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2025, pp. 522–538.

[16] M. R. I. Rabin, A. Hussain, M. A. Alipour, and V. J. Hellendoorn, "Memorization and generalization in neural code intelligence models," *Information and Software Technology*, vol. 153, p. 107066, 2023.

[17] N. Carlini, D. Ippolito, M. Jagielski, K. Lee, F. Tramer, and C. Zhang, "Quantifying memorization across neural language models," in *The Eleventh International Conference on Learning Representations*, 2022.

[18] Y. Huang, Y. Li, W. Wu, J. Zhang, and M. R. Lyu, "Your code secret belongs to me: neural code completion tools can memorize hard-coded credentials," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2515–2537, 2024.

[19] Y. Yang, H. Fan, C. Lin, Q. Li, Z. Zhao, C. Shen, and X. Guan, "A survey on adversarial machine learning for code data: Realistic threats, countermeasures, and interpretations," *arXiv preprint arXiv:2411.07597*, 2024.

[20] A. Al-Kaswan, M. Izadi, and A. Van Deursen, "Traces of memorisation in large language models for code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[21] Z. Sun, X. Du, F. Song, M. Ni, and L. Li, "Coprotector: Protect open-source code against unauthorized training usage with data poisoning," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 652–660.

[22] Z. Sun, X. Du, F. Song, and L. Li, "Codemark: Imperceptible watermarking for code datasets against neural code completion models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1561–1572.

[23] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, "Gotcha! this model uses my code! evaluating membership leakage risks in code models," *IEEE Transactions on Software Engineering*, 2024.

[24] A. Al-Kaswan, M. Izadi, and A. Van Deursen, "Traces of memorisation in large language models for code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[25] S. Zhang, H. Li, and R. Ji, "Code membership inference for detecting unauthorized data use in code pre-trained language models," pp. 10 593–10 603, Nov. 2024. [Online]. Available: https://aclanthology.org/2024.findings-emnlp.621/

[26] T. Tanay and L. Griffin, "A boundary tilting persepective on the phenomenon of adversarial examples," *arXiv preprint arXiv:1608.07690*, 2016.

[27] S. Tian, G. Yang, and Y. Cai, "Detecting adversarial examples through image transformation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[28] S. Hu, T. Yu, C. Guo, W.-L. Chao, and K. Q. Weinberger, "A new defense against adversarial images: Turning a weakness into a strength," *Advances in neural information processing systems*, vol. 32, 2019.

[29] C. A. Choquette-Choo, F. Tramer, N. Carlini, and N. Papernot, "Label-only membership inference attacks," in *International conference on machine learning*. PMLR, 2021, pp. 1964–1974.

[30] 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.

[31] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[32] 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.

[33] M. Javaheripi, S. Bubeck, M. Abdin, J. Aneja, S. Bubeck, C. C. T. Mendes, W. Chen, A. Del Giorno, R. Eldan, S. Gopi *et al.*, "Phi-2: The surprising power of small language models," *Microsoft Research Blog*, vol. 1, no. 3, p. 3, 2023.

[34] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," *arXiv preprint arXiv:2306.08568*, 2023.

[35] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE Symposium on Security and Privacy*, 2017, pp. 3–18.

[36] G. Apruzzese, H. S. Anderson, S. Dambra, D. Freeman, F. Pierazzi, and K. Roundy, ""real attackers don't compute gradients": bridging the gap between adversarial ml research and practice," in *2023 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, 2023, pp. 339–364.

[37] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1482–1493.

[38] S. Hisamoto, M. Post, and K. Duh, "Membership inference attacks on sequence-to-sequence models: Is my data in your machine translation system?" vol. 8. MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info . . . , 2020, pp. 49–63.

[39] H. Zhang, Z. Fu, G. Li, L. Ma, Z. Zhao, H. Yang, Y. Sun, Y. Liu, and Z. Jin, "Towards robustness of deep program processing models—detection, estimation, and enhancement," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–40, 2022.

[40] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1482–1493.

[41] C. Na, Y. Choi, and J.-H. Lee, "Dip: Dead code insertion based black-box attack for programming language model," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 7777–7791.

[42] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, "Measuring coding challenge competence with apps," *arXiv preprint arXiv:2105.09938*, 2021.

[43] 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.

[44] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, “Unveiling memorization in code models,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[45] N. Carlini, F. Trämer, E. Wallace *et al.*, “Extracting training data from large language models,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2633–2650.

[46] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, “Starcoder: may the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.

[47] J. He, M. Vero, G. Krasnopolska, and M. Vechev, “Instruction tuning for secure code generation,” *arXiv preprint arXiv:2402.09497*, 2024.

[48] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[49] Y. Jiang, Y. Zhang, L. Lu, C. Treude, X. Su, S. Huang, and T. Wang, “Enhancing high-quality code generation in large language models with comparative prefix-tuning,” *arXiv preprint arXiv:2503.09020*, 2025.

[50] Z. Chen and L. Jiang, “Promise and peril of collaborative code generation models: Balancing effectiveness and memorization,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 493–505.

[51] A. Finkman, E. Bar-Kochva, A. Shapira, D. Mimran, Y. Elovici, and A. Shabtai, “Codecloak: A method for evaluating and mitigating code leakage by llm code assistants,” *arXiv e-prints*, pp. arXiv–2404, 2024.

[52] Y. He, B. Li, L. Liu, Z. Ba, W. Dong, Y. Li, Z. Qin, K. Ren, and C. Chen, “Towards label-only membership inference attack against pre-trained large language models,” in *USENIX Security*, 2025.

[53] Y. Wan, G. Wan, S. Zhang, H. Zhang, P. Zhou, H. Jin, and L. Sun, “Does your neural code completion model use my code? a membership inference approach,” *arXiv preprint arXiv:2404.14296*, 2024.

[54] V. Majdinasab, A. Nikanjam, and F. Khomh, “Trained without my consent: Detecting code inclusion in language models trained on code,” *arXiv preprint arXiv:2402.09299*, 2024.