Peeking Inside the Black Box: Training Data Exposure in Code Language Models

Angelica Spina[a], Marco Russodivito[a], Simone Scalabrino[a], Rocco Oliveto[a]

[a] *University of Molise, Contrada Fonte Lappone, 86090, Pesche (IS), Italy*

Abstract

Large Language Models (LLMs) have demonstrated effective in tackling coding tasks, leading to their growing popularity in commercial solutions like GitHub Copilot and ChatGPT. These models, however, may be trained on proprietary code, raising concerns about potential leaks of intellectual property. A recent study indicates that LLMs can memorize parts of the source code, rendering them vulnerable to extraction attacks. However, it used *white-box* attacks which assume that adversaries have partial knowledge of the training set.

In this paper, we present a pioneering effort to conduct a *black-box* reconstruction attack on an LLM – CodeT5+ – trained to tackle a specific coding task – code summarization. We assume the adversary has no knowledge about the training set. We train an *inverse model*, *i.e.,* a model that, given a comment, aims to reconstruct the source code from the training set. Then, we try to understand to what extent such a model can reconstruct the code in the training set. Our results show that the attack through the inverse model does not allow an adversary to fully reconstruct training code instances, except for a minority of cases. On the other hand, an in-depth manual analysis of the reconstructed code reveals that some important information (such as the APIs adopted) can be extracted in several cases, showing the potential vulnerability of such models.

Keywords: LLMs for Coding Tasks, Security, Reconstruction Attacks.

1. Introduction

Large Language Models (LLMs) are playing an increasingly prominent role in software engineering research, showing particular effectiveness in tasks such as bug fixing [1, 2, 3], code summarization [1, 4, 5], code generation [6, 7], and many others. These models operate by processing sequences of textual tokens as input and producing new sequences as output, which may consist of source code or natural language, depending on the specific task. LLMs for coding tasks typically build upon the Transformer architecture [8] and are trained in two distinct phases. In the first phase, semi-supervised *pre-training*, the model is exposed to large-scale codebases to learn general language patterns and programming conventions, independent of any particular downstream application. In the second phase, called *fine-tuning*, the model is further trained in a supervised manner on data tailored to a specific coding task, learning to map inputs to the desired outputs for that task. This two-step approach has proven highly effective [9, 10], fueling the adoption of LLMs in commercial products such as GitHub Copilot[1] and ChatGPT[2], which

[1] https://copilot.microsoft.com/

[2] https://chat.openai.com

are rapidly becoming essential tools for software developers [11].

Although most general-purpose LLMs are fine-tuned primarily on open source code, due to its abundance and accessibility, there is a growing trend toward the development of specialized models [12, 13, 14]. These specialist LLMs are fine-tuned on proprietary datasets to achieve superior performance in niche domains or organization-specific contexts that public code repositories may not adequately cover. For example, a company might want to build a coding assistant tailored to its unique technology stack, leveraging data from thousands of internal projects and developers. Fine-tuning an LLM on such proprietary data enables the model to provide more relevant and context-aware assistance. However, fine-tuning on proprietary code introduces new risks, as it increases the likelihood that sensitive information from the training data could be memorized and, under certain conditions, exposed.

The risk becomes particularly acute if it is possible for an end user, without privileged access, to extract confidential information from the model simply by interacting with it. Let us denote the confidential dataset used for fine-tuning as T_p. If an attacker could extract private details about any instance $p \in T_p$, this would constitute a serious breach of confidentiality.

This concern is not merely theoretical. Increasing evidence suggests that LLMs – especially when fine-tuned – are susceptible to memorization, inadvertently retaining and reproducing specific training examples in their output behavior. This phenomenon has been highlighted in recent work by Al-Kaswan *et al.* [15], who showed that LLMs for code-related tasks can indeed memorize and regurgitate token sequences from their training data. Their study demonstrated that such memorization makes these models vulnerable to *white-box* extraction attacks, in which the adversary has partial access to the training dataset T_p.

Although these findings are significant, they rely on assumptions that may not hold in practice, namely, that an attacker has some prior knowledge of the training data. In many real-world scenarios, this is unlikely. This raises an important question: Are these models also vulnerable to *black-box attacks*, where the adversary has no visibility into the training data at all? If so, the implications for the safe deployment of fine-tuned LLMs, especially those trained in proprietary or sensitive code, would be even more concerning.

Prior studies have shown that other forms of privacy attacks are possible on deep neural networks (DNNs) [16, 17, 18, 19, 20, 21]. Among these, *Reconstruction Attacks* (or *Model Inversion Attack*) aim to recover one or more training samples from the model, assuming the adversary has no direct information about the original training data [16].

We preliminarily studied to what extent a *black-box* attack (*Reconstruction Attack*) can be used to reconstruct training instances from a T5-based LLM trained to generate code summaries (*i.e.,* documentation) from a given code snippet [22]. Our results showed that the attack failed in the large majority of the cases. However, we observed that such an attack allowed us to partially reconstruct code aimed to achieve very simple tasks (such as logging a statement or deleting a file).

In this paper, we extend our previous work [22] by presenting a comprehensive study on the feasibility of reconstruction attacks against an LLM trained for code summarization. Unlike our earlier study, which focused on a general-purpose T5-based model [1], this work targets a specialized LLM that has been explicitly pre-trained on source code: CodeT5+ [23].

Our attack is based on the construction of an inverse model, *i.e.,* a model trained to generate source code from natural language descriptions produced by the target model. The underlying idea is that learning to reverse the summarization process would potentially reproduce code samples from the proprietary fine-tuning set used to train the target model. To build the inverse model, we first collected an auxiliary set of approximately 700k open-source code snippets from

established datasets [24, 25, 26]. We then used the target CodeT5+ model to generate summaries for each snippet. This process produced an *attack set*, a collection of summary-code pairs, which was used to train the *inverse model* to reconstruct source code from a summary.

We evaluated whether the inverse model enables an adversary to extract information from the proprietary training set (T_p) of the target model. We designed two experimental settings: a *controlled* scenario, in which the input to the inverse model consists of the original summaries present in T_p, and a more *realistic* scenario, in which the attacker does not have access to the original comments and instead uses paraphrased versions generated with GPT-3.5.

We first conducted a quantitative evaluation using *CodeBLEU*[27], *ROUGE-L*[28], and *METEOR*[29] to assess how closely the reconstructed code matches the original training samples. This was complemented by an extensive manual analysis to investigate why reconstruction failed in most cases. Furthermore, we performed a qualitative analysis to determine whether sensitive information could still be recovered, even when the reconstructed code was not a close match. In particular, we annotated several code properties (*e.g.,* control flow, conditional logic, internal/external API calls, and variable initializations) to assess whether they were correctly inferred.

Our findings corroborate the results of our earlier study [22]: the inverse model is largely ineffective at fully reconstructing training instances, with only a few isolated successes. However, manual analysis revealed a more nuanced picture. Even when full reconstruction failed, partial leakage of sensitive information, such as internal API calls, was observed in a non-negligible portion of cases: approximately 3–17% in the controlled scenario and 3–9% in the realistic scenario. These results suggest that black-box reconstruction attacks, while limited in scope, can still pose a meaningful threat to the confidentiality of fine-tuned code LLMs.

In summary, this paper extends our previous work [22] in several key ways:

- We adopt a more modern and code-specialized LLM, *i.e., CodeT5+*, as both the target and the inverse model, replacing the general-purpose T5 used in our earlier study.
- We introduce a new and more realistic evaluation scenario, in which the adversary does not have access to the original summaries used to train the target model, an assumption that better reflects real-world conditions.
- We conduct an extensive manual analysis of reconstructed instances to (i) identify the causes behind reconstruction failures, and (ii) assess whether private or sensitive information can still be recovered by the inverse model, even when full reconstruction.

2. Background and Related Work

In the domain of machine learning security, *Reconstruction Attacks* or *Model Inversion Attacks* (MI) have gained significant attention as a means of recovering sensitive training data from a model. These attacks aim to reconstruct training data $\hat{X}$ from a trained model $f_\theta(\cdot)$ using auxiliary information D (*e.g.,* output labels or partial knowledge of some features), without direct access to the original data $\hat{X}$ [16, 30]. MI attacks are generally categorized into *white-box* and *black-box* attacks: the former assumes the adversary has full or partial access to the model's internal architecture, parameters, or gradients, while in the latter the adversary can only rely on query-response mechanisms, without insight into its internal structure [31]. A substantial amount of research has investigated MI attacks on models trained for various tasks, such as classification and generation, across different data types, with a particular focus on images and text [30]. The first MI attack was introduced by Fredrikson et al. [32], demonstrating how linear regression

models could be inverted to predict a patient's generic markers. Later, Fredrikson et al. [33] proposed a more generic algorithm designed to be used with various models such as decision trees and neural networks for face recognition, in both *white-box* and *black-box* settings.

2.1. Attacks on Image-Based Models

For image-based models, subsequent works have leveraged *Generative Adversarial Networks* (GANs) as a powerful tool to enhance the fidelity and realism of reconstructed images. For example, Hitaj et al. [34] demonstrated the potential of GANs to extract private training data in collaborative learning environments. Later, Chen et al. [35] proposed an inversion-specific GAN with a discriminator trained to classify the input into one of $K+1$ classes, where the first K classes correspond to the labels of the target network and the $(K+1)$-th class represents fake samples, modeling a private data distribution for each class. At the same time, Wang et al. [36] approached MI attacks as a variational inference problem, leveraging statistical divergence to improve reconstruction diversity and quality. Other approaches include the use of pseudo labeling-guided strategies via conditional GANs (cGANs) [37] and model augmentation techniques based on knowledge distillation [38]. *Black-box* attacks on image models remain particularly challenging. Yang et al. [39] pioneered a *learning-based black-box* MI attack, in which an inversion model is trained to reconstruct private images using only the output of the target model. Other approaches leverage model explanations in XAI-aware inversion attacks to enhance inversion performance [40], while some propose techniques that adjust the synthesized image to move it away from the decision boundary of the class, simulating the gradient-based optimization used in *white-box* setting [41].

2.2. Attacks on Text-Based Models

Research on MI attacks in Natural Language Processing (NLP) has progressed significantly. Early work focused on recovering unordered input words inverting embedding vectors [42], later improved using generative models to reconstruct structured text sequences [43]. When it comes to models trained for code-related tasks, existing research primarily focus on *Training Data Extraction Attacks* against models trained for the Code Completion task, which aim to assess the level of memorization of training data by querying the model with specific prefixes (*e.g.,* function definition statements [20] or generic token sequences from the training code [44]) and evaluating whether the suffix is something actually present in the training set. This is based on the concept of Type-1 clone search or they even use fuzzier metrics like BLEU [45]. Also Cheng et al. [21] present two attack scenarios against two Code Completion Tools (*i.e.,* GitHub Copilot and Amazon Q): an example of *Training Data Extraction Attack* and an example of *Jailbreaking attack*. To the best of our knowledge, the only Model Inversion attack executed on a code model is our previous work [22], which adapts the *black-box* MI attack of Yang et al. [39] - originally designed for image models - to a T5 model trained for *Code Summarization*. Unlike *Training Data Extraction Attacks*, this approach explores the possibility of inferring code representations using auxiliary information beyond the target code itself. Specifically, it involves training a new *Inverse model* to perform the inverse task of the target model using a dataset of open-source code and the corresponding comments generated by the target model. Once trained, the inverse model can be queried using the comments from the target model's training set, generating code snippets that are compared to the original training data. However this study has several limitations: (i) it employs a language models (T5) rather than a specialized code model, potentially affecting attack efficacy; (ii) it relies solely on BLEU metric for evaluation, which fails to fully capture the

semantic accuracy of the generated code; (iii) it assumes an unrealistic attack scenario where the adversary uses exact comments from the training set.

Building upon this work, our study extends Russodivito et al. [22] by addressing these key limitations: (i) we employ CodeT5+, a model specifically designed for code-related tasks, as both the target and inversion models; (ii) we introduce a more comprehensive evaluation framework, leveraging *CodeBLEU*, *ROUGE-L*, *METEOR* and manual analysis, offering a more precise assessment of reconstruction quality; (iii) we propose a more realistic adversarial scenario, replacing exact training comments with paraphrased versions.

3. Reconstructing Training Examples of LLMs for Coding Tasks

Let us assume that there is a company *Com* that is using its own proprietary code (T_P) to train an LLM for coding tasks, and that *Com* wants to share such an LLM with users who should not have access to T_P. In other words, the company is interested in keeping the *confidentiality* of T_P. *Com* may be interested in assessing to what extent it is possible to retrieve private information in T_P by simply querying the LLM. In this section, we describe our methodology for helping practitioners achieve this goal by simulating an attack aimed at violating the confidentiality of an LLM for coding tasks. In this simulation, an adversary tries to retrieve private information from T_P. We decided to work in a *black-box* scenario: The adversary is unaware of the model architecture or its hyperparameters. The adversary has some basic knowledge of the target model (that we will explicitly report below).

In the following, we first describe the specific task on which we focus, *i.e., code summarization*. Then, we present an overview of the generic attack methodology, inspired by previous work. Finally, we describe in details the single steps and assumptions used to build an inverse model for code summarization.

3.1. Target Task

Our methodology is generic and can be easily adapted to any coding task. In this study, however, we focus on a specific task, *i.e., code summarization* [4, 46]. Given a code snippet, a *code summarization* model automatically generates a natural language summary for such code, which could be used as documentation. We chose this task because the attack we will adopt is possibly more effective on such a type of task. As we will explain later, we will build the *inverse model* of the target model. The inverse of a *code summarization* model is a *code generation* model, which can reconstruct *source code* given a *natural language query*. An adversary could easily perform an attack with such a model, assuming they know what to look for. On the other hand, consider, for example, the *bug fixing* task, which, given the *buggy* version of a program generates its *fixed* version. Its inverse would be a *mutant generation* model, which given the *fixed* code introduces bugs into it. An attacker would need to provide the inverse model with the *fixed* code, which is what they want to retrieve in the first place. This makes such a task not particularly attackable in the first place.

3.2. Attack Methodology Overview

The attack is inspired by the work by Yang et al. [39] and is organized in two main stages. First, we query the target model using an *Auxiliary set* to collect the output sequence and build an *Attack set*. Second, we use such an *Attack set* to train an *Inverse model* which takes the prediction

sequence as input and outputs the reconstructed sequence that could be decoded to the original sample. The attack stages follow the *Kill Chain* model [47], executing the following five steps.

Reconnaissance. The purpose of this step is to analyze the target model and identify the semantics of the input. During this step, the adversary selects and inspects the target model to obtain information regarding the input and the output. In this phase, they also have to discover the tokenizers to use to interact with the model. For example, against a facial recognition system, the adversary needs to find out what kind of images are accepted by the model (e.g., frontal face, 224×224 RGB), and how to process raw photos in the correct format.

Weaponization. During this step, the attacker acquires the *Auxiliary set* that consists of samples that are semantically consistent with the target model's input. For example, against a facial recognition model, the adversary collects a large number of face photos from publicly available datasets or social media platforms.

Delivery. The attacker queries the target model with all the encoded *Auxiliary dataset* samples to construct the *Attack dataset*, which is made up of pairs of encoded *Auxiliary dataset* samples and the corresponding predicted sequence. For example, each face image is passed through the model and its output (*i.e.,* the associated identity) is recorded, resulting in a dataset of picture-identity pairs.

Exploitation. During the *Exploitation* phase, the attacker trains a new model, referred to as the inverse model, which learns to map the output embeddings of the target model back to the original sequence, so the original code snippet. The training set of the inverse model is the *Attack dataset* built in the previous step. For instance, in the case of a facial recognition system, this inverse model takes as input the embeddings produced by the facial recognition system (assigned identity) and attempts to reconstruct an approximation of the original face image.

Action on Objective. During this last phase, the attacker queries the inverse model in order to reconstruct some original training samples, *e.g.,* in the case of the facial recognition system, the attacker attempts to visualize or re-identify specific individuals, potentially violating their privacy.

3.3. Performing the Reconstruction Attack

We describe below how we declined the previously-mentioned Kill Chain model to attack a code summarization model. We first present the target model, *i.e.,* the model we aim to attack. Then, we present how we implemented the steps of the Kill Chain model.

3.3.1. Target Model

In our previous work [22], we carried out the reconstruction attack against the model proposed by Mastropaolo et al. [1], which is a T5 model trained for code-related tasks. T5 (Text-to-Text Transfer Transformer) is a transformer-based *sequence-to-sequence* model proposed by Google [48] that reformulates all NLP tasks into a text-to-text format and leverages a unified framework for both input and output representations. The T5 architecture [48] is an encoder-decoder transformer model. The encoder uses self-attention, multi-head attention modules, feed-forward networks, layer normalization, and residual connections for stable and efficient training. Dropout is applied throughout the architecture, including feed-forward networks and attention distributions. The decoder employs autoregressive self-attention, only attending to preceding tokens. Outputs are processed through a dense softmax layer for token probabilities. Positional encoding uses shared scalar values across layers for efficiency. T5 comes in five sizes with parameters ranging from 60 million to 11 billion. Due to resource constraints, Mastropaolo et al.

[1] adopted the smallest model variant, that we targeted in our previous work. Pre-training involved masked language modeling with 2.67 million samples, including Java methods, abstracted versions, and Javadoc comments from the CodeSearchNet dataset[24]. For fine-tuning, a multi-task learning strategy was applied to adapt the model to four downstream tasks using datasets from previous studies for comparison. The T5 model is pre-trained to perform natural language tasks and has been adapted and fine-tuned to perform coding tasks. Previous work introduced code-specific variants of T5 (*i.e.,* CodeT5 and CodeT5+). In this extension, we decided to use *CodeT5+* [23] as the target model since it is the most recent and advanced variant between the two. *CodeT5+* is specifically designed for code-related tasks, exploiting the original raw method, without the need to abstract the code. CodeT5+ is based on the encoder-decoder architecture [23] but enhances flexibility for various downstream tasks through a two-stage pre-training on unimodal and bimodal data. In the first stage, CodeT5+ was pre-trained with a massive code data and then, during the bimodal pre-training, they continued to pre-train the model with a smaller set of code-text data with cross-modal objectives, exposing the model to more diverse data to learn rich contextual representations. The tasks that they used are different based on the type of data used (*i.e.,* unimodal or bimodal). For the unimodal pre-training, they used (i) *Span Denoising*, randomly replacing 15% of the tokens with sentinel tokens in the encoder inputs, requiring the decoder to recover them via generating a combination of these spans; they then used two variants of (ii) *Causal Language Modeling*(CLM), in which they randomly select a pivot location in the text (concatenating a [CLM] token) and the context before becomes the source sequence and the context after is the target output, while the second variant is a decoder-only generation task by always passing the separating token [CLM] to the encoder input and require the decoder to generate the full code sequence. For the bimodal pre-training, they used three different tasks: (i) *Text-Code Contrastive Learning* aims to align natural language descriptions and code snippets in a shared semantic space by pulling matching pairs closer and pulling non-matching pairs apart, (ii) *Text-Code Matching* determines whether a natural language description semantically corresponds to a given code snippet; (iii) *Text-Code Causal LM* trains a model to generate code from natural language descriptions in a left-to-right manner, predicting each token conditioned on the preceding ones, and this is done through a dual multimodal conversion: text-to-code generation and code-to-text generation. Five different versions of *CodeT5+* have been proposed: (i) *CodeT5+* 220M, (ii) *CodeT5+* 770M, (iii) *CodeT5+* 2B, (iv)*CodeT5+* 6B, and (v) *CodeT5+* 16B. Taking into account the available computing power, we decided to use the smallest version of *CodeT5+* (*CodeT5+* 220M) for all experiments.

We initialized our model using the pre-trained version of CodeT5+, which was subsequently fine-tuned on our task-specific dataset. In particular, to replicate the study by Russodivito et al. [22], we decided to fine-tune the model to perform the *Code Summarization* task. We chose a state-of-the-art dataset, the one presented by Shi et al. [49], which is a cleaned version of the *Funcom* dataset [50]. This dataset comprises 1,184,438 training instances, where each sample is represented as a $\langle code, summary \rangle$ pair. Specifically, the code corresponds to Java methods while the summary corresponds to the first sentence of the method's Javadoc comment. This dataset is the target one, so we will refer to it as *Target dataset* (D_{target}). Indeed, our objective is to reconstruct some information belonging to the Java codes within this dataset, having access only to the model itself as a *black-box* system and to the comments in the pairs. We fine-tuned *CodeT5+* 220M for 15 epochs with a batch size of 16. The fine-tuning process was performed using the standard hyperparameters for *CodeT5+*, including the AdamW optimizer and a learning rate of $2e-5$. The training took about two weeks, during which we saved a checkpoint every epoch. To select the best-performing one, we employed a validation-based selection strategy,

evaluating the accuracy of each checkpoint on the validation set and retain the one with the highest accuracy.

3.3.2. Implementing the Kill Chain Model

Reconnaissance. We assume the adversary has some basic background knowledge about the target model. First of all, the adversary knows the semantics of the input data, even if they have no access to the original training data. This is possible since the adversary is free to query the model how many times they like, trying with different input data types, until they understand the semantics. Moreover, *sequence-to-sequence* models do not accept raw data as input, *e.g.,* sentences. They accept only large fixed-size vectors, obtained by using a so called *tokenizer*. A *tokenizer* is required to map every word with a numerical identifier and to map back the *sequence-to-sequence* output, *e.g.,* to associate numerical identifier to a word to build the sentence. Since *tokenizers* are deterministic functions, we assume that these are either publicly usable by legit users or reconstructible by statistical inference.

Weaponization. The most important requirement for this type of *Reconstruction Attack* is the availability of a pool of data, called *Auxiliary set* (D_{aux}). This dataset must contain samples that match the semantics of the original training dataset. For example, Yang et al. [39] targeted a facial recognition classifier. Thus, the *Auxiliary set* was composed by public facial images of random individuals from the Internet. Such samples keep generic visual traits like face edges and eye and nose placement, which are shared semantic features of the original training data [39]. Against a *sequence-to-sequence* model like T5 trained for *Code Summarization*, we decided to use a mixture of different state-of-the-art datasets. Specifically, we combine: (i) CodeSearchNet [24], (ii) TL-CodeSum, a collection of Java projects with at least 20 stars created from 2015 to 2016, and (iii) DeepCom, that was built from 9,714 open-source projects. CodeSearchNet [24] comprises functions with corresponding documentation in multiple programming languages, including Go, Java, JavaScript, PHP, Python, and Ruby, extracted from open-source GitHub projects. For our study, we used only the Java subset, which consists of 181,061 instances. Additionally, we incorporated all partitions (train, validation and test) of TL-CodeSum [25], resulting in 87,136 instances, and the complete DeepCom [26] dataset, which contains 588,108 instances. Combining these sources resulted in an initial dataset of 856,305 samples. We then applied data processing techniques to mitigate data leakage and ensure dataset integrity. Specifically, we performed clone detection within the dataset, checking whether there were clones of the same code-comment pair or different samples with same code or same comment, thus ensuring a one-to-one association between code and the summary.

To mitigate data leakage, we performed clone detection between the initial version of the auxiliary set (D_{aux}) and the target set D_{target}, removing any duplicated samples from D_{aux}. This step was crucial to avoid false positives during reconstruction —- i.e., cases where the inverse model appears to reconstruct target samples solely because they were already present in the auxiliary data used to train the inverse model itself. Another consideration is that these models are intended for use in corporate environments, where they may be trained on private data that must remain confidential and protected from unauthorized access. In a real-world scenario, it would be impossible for the adversary to find online or by other sources of information the same proprietary code.

Furthermore, we excluded all samples containing methods exceeding 512 tokens, aligning with the model's context window size. After these preprocessing steps, the final *Auxiliary set* (D_{aux}) comprised 404,452 samples.

Delivery. To build the Attack Set (D_{atk}), we assume the adversary already knows that the target model takes as input tokenized single-function Java methods with a maximum length of 512 tokens (*i.e.,* the maximum source length used during the pre-training of the model [23]). To build the final *Attack Set* we query the target model (fine-tuned *CodeT5+*) with the collected methods (Java source-code in D_{aux}). In other words, we define D_{atk} as $\langle M_{target}(\text{code}), \text{code}\rangle$ pairs. We then split the *Attack set* into training, validation, and test sets (80-10-10 ratio) that were used to train the inverse model and to select the best checkpoint.

Exploitation. We opted to use for the inverse model the same architecture of the target model (*i.e., CodeT5+*). We fine-tuned the inverse model using the *Attack Set* D_{atk}. We trained it for 20 epochs with a batch size of 16, and we saved a checkpoint each epoch. The best model checkpoint is selected using a simple early stopping criterion based on accuracy improvements. Starting from the first epoch, the method tracks the best accuracy achieved so far. A new epoch is considered better only if its accuracy exceeds the current best by at least a threshold $\delta = 0.001$. If no improvement is observed for a consecutive window of 5 epochs, the search stops. The last epoch showing a significant improvement is selected as the final checkpoint. This approach ensures both high accuracy and training stability. We used the same hyperparameters we adopt for the target model.

Action on Objective. Since we simulate the attack, we are not interested in performing any specific action. However, in Sec. 4 we study the effectiveness of such an attack.

4. Empirical Study Design

The *goal* of this study is to understand whether an LLM (*CodeT5+*) trained for tackling coding tasks (*Code Summarization*, specifically) is vulnerable to *Reconstruction Attacks*. In particular, our objective is to investigate the extent to which information memorized during training can be reconstructed using inversion techniques. To this end, we formulate the following research questions:

RQ_1 *To what extent can methods included in the training set of a Code Summarization model be reconstructed?*
This research question aims to quantify the number of training samples that can be recovered through the proposed reconstruction attack.

RQ_2 *To what extent can meaningful information be extracted from partially reconstructed samples?*
This research question investigates whether partial reconstructions still leak valuable or sensitive information.

4.1. Context Selection

The context of our study is composed of three *objects*: (i) the target model M_{target}, (ii) the inverse model $M_{inverse}$, and (iii) the test set of the *Target Dataset* (D_{target}). We already detailed in Section 3 how we chose/defined the two models: Briefly, we adopt the *CodeT5+* model fine-tuned to perform code summarization as M_{target}, and we train $M_{inverse}$ from it by using the same architecture. As previously mentioned, as for D_{Target}, we chose the dataset by Shi et al. [49].

```
// Original: tell whether the user has actually edited the string value or not
// Paraphrased: determine if the user has made changes to the string value.
public boolean valueChanged() {
    if (!(origEncoding.equals(""))) {
      String newEncoding = (String)combEncoding.getSelectedItem();
      if (!(newEncoding.equals(origEncoding))) {
        return true;
      }
    }
    if (combValue == null) {
      if (txtfValue == null) {
        if (!origString.equals(txtaValue.getText())) {
          return true;
        } else {
          return false;
        }
      } else {
        if (!origString.equals(txtfValue.getText())) {
          return true;
        } else {
          return false;
        }
      }
    } else {
      String newval = "";
      if (combValue.getSelectedIndex() < 0) {
        newval = (String)combValue.getSelectedItem();
      } else {
        newval = localVals.elementAt(combValue.getSelectedIndex());
      }
      if (!origString.equals(newval)) {
        return true;
      } else {
        return false;
      }
    }
  }
```

Figure 1: Example of non-trivial code in the target dataset.

4.2. Experimental Procedure

Our goal is to assess not only whether training samples can be reconstructed, but also how this depends on the level of prior knowledge about the original dataset available to the attacker. Thus, to answer to both our research questions, we design and evaluate two distinct attack settings:

- *Controlled Scenario*: We assume the adversary has (and uses) the exact comments that were part of D_{target}. This (highly unlikely) setup represents an idealized attack case, where the adversary has the highest amount of information about D_{target}.
- *Realistic Scenario*: We assume the adversary manually crafts the summaries from which they want the code to be generated. This setup models a more realistic attack, in which the adversary only knows that code implementing a given goal (Figure 1) is included in the dataset.

In the *Controlled Scenario*, we test the inverse model using the original summaries in D_{target} to evaluate the upper bound of reconstruction accuracy. In the *Realistic scenario*, we test the inverse model with paraphrased versions of the original summaries in D_{target}. To automatically generate the paraphrases of the summaries, we use GPT-3.5 with its default temperature parameter at the time we ran our experiments (*i.e.,* 0.7) [51, 52]. We opted for GPT-3.5 because, at the time of our experiments, it was substantially less expensive than newer GPT variants. Since we needed to generate paraphrases for 1,184,438 comments—and our goal was simply to obtain semantically equivalent comments with wording different from the original—GPT-3.5 offered a cost-effective and sufficient solution. We present a small-scale comparison with a more recent model and different temperature settings in Sec. 7.

In both scenarios, the inverse model and the auxiliary set remain unchanged: only the input queries differ between the two settings. This design allows us to isolate the effect of prior knowledge (whether access to original summaries or paraphrases) on the ability of the inverse model to reconstruct the target code.

To answer RQ_1, we employed *CodeBLEU* [27], *ROUGE-L* [28], and *METEOR* [29].

CodeBLEU is a metric specifically designed to work for code generation tasks for evaluating the similarity between generated code (*i.e.,* code reconstructed by the inverse model) and reference code (*i.e.,* the original code in the dataset). This metric allows us to quantitatively assess the reconstruction capabilities of the inverse model. Unlike traditional natural language metrics such as *BLEU* [45], *CodeBLEU* considers additional aspects of code that are crucial for evaluating program correctness, including *syntax matching* via Abstract Syntax Trees and *semantic matching* through data-flow analysis. By incorporating these additional factors, *CodeBLEU* is able to provide a more accurate evaluation of the reconstruction quality, which is critical when working with structured code rather than free-form text.

ROUGE-L [28] measures similarity based on the *Longest Common Subsequence* (LCS) between two sequences, rewarding in-order token matches regardless of whether they are consecutive. In our setting, we use *ROUGE-L* to compare the generated code with the reference one. This allows us to capture not only lexical overlap but also the preservation of structural ordering, which is important in assessing the fidelity of code reconstruction.

METEOR [29] compares the generated text with a reference using a combination of exact token matches, stemmed forms, synonyms, and paraphrase matches. Additionally, *METEOR* introduces a penalty for word order differences, which helps reflect both content similarity and fluency.

To address RQ_2, we manually analyzed a randomly-selected sample of 175 reconstructed instances (7.4% margin of error, 95% confidence level) for each scenario, resulting in 350 evaluations. Our objective was to understand whether some information about the reference code was successfully recovered. We evaluated such a number of samples to achieve a reasonable balance between the margin of error and the cost of analysis (note that such a manual evaluation required about 55 hours).

We defined a structured questionnaire aimed to systematically capture the type of information that was recovered from the training set. We report the complete list of questions in Table 1.

With the first question we want to evaluate whether the control flow of the original code was recovered. The second question concerns the reconstruction of the conditions of loops and control structures and aims to assess whether the method recovered such information. We use *Yes* when *all* the conditions have been correctly recovered, *No* when *none* of them have been reconstructed, *Partial* to indicate that *some* of the conditions have been recovered, and *NA* (Not Available) when no loops and control structures were available in the method in the first place. The third question evaluates whether the same internal APIs (*i.e.,* methods defined in the project itself) are used. Again, we use *NA* to indicate that no such API is invoked in the first place. Similarly, with the fourth question we want to evaluate whether the same external APIs (*e.g.,* JDK methods) are recovered. Again, NA applies if the original code makes no external API calls.

Finally, the last question concerns values used in variable assignments. We use *NA* when no assignment is done.

One of the authors performed an initial evaluation and marked the ambiguous cases in which they had doubt about the annotation (35 and 15 for the *controlled* and *realistic* scenarios, re-

ID	Question	Options
Q_{CF}	Does the reconstructed method execute the same sequence of instructions as the original method, in the same logical order?	*Y*, *N*
Q_{cond}	Do the reconstructed method's loops and control structures have similar conditions to those in the original method?	*Y*, *N*, *P*, *NA*
$Q_{API-int}$	Does the reconstructed method use the same developer-defined methods or classes as the original code?	*Y*, *N*, *NA*
$Q_{API-ext}$	Does the reconstructed method invoke the same Java standard library methods and external API calls as the original method?	*Y*, *N*, *NA*
Q_{var}	Are variables assigned with the same values in both the original and reconstructed methods?	*Y*, *N*, *NA*

Table 1: Questions used to answer RQ_2 and allowed options for each of them (**Y**es, **N**o, **P**artial, **N**ot **A**vailable).

spectively). Such cases (that were still originally labeled by the initial evaluator) were analyzed and discussed with another author, aiming to reach a consensus. We report the percentage of positive/negative answers to each question for both the previously reported scenarios.

4.3. Experimental Environment

The experiment was performed using Google Colaboratory (*i.e.,* Google Colab), a cloud-based platform that offers free access to powerful GPUs, enabling model training without the need for local computational resources. In particular, we leveraged NVIDIA A100 and NVIDIA L4 GPUs. The A100 is designed for high-performance training workloads and for this reason we used it to train all our models (*i.e.,* both target and inverse models). The L4, on the other hand, is optimized for AI inference, so we used it to evaluate models' checkpoints and for querying the inverse model during the attack. Overall, the training and inference phases and the generation of paraphrases required a cost of about €600.

5. Empirical Study Results

The analysis for each research question (RQ_1 and RQ_2) involved a direct comparison between the results obtained in our two different scenarios, allowing us to examine how variations in the level of prior knowledge of the attacker affect both the accuracy of code reconstruction (RQ_1) and the ability to extract sensitive information from partially reconstructed methods (RQ_2). We discuss the results obtained for the two RQs below.

5.1. RQ_1. Accuracy of Code Reconstruction

Figure 2 shows the *CodeBLEU* score [27] distributions for the reconstructed training instances, obtained (a) in the controlled scenario, *i.e.,* by querying the inverse model with the original comments from D_{target}, (b) in the realistic scenario, *i.e.,* by querying the inverse model with paraphrased versions of the comments. We observe that the reconstruction accuracy is generally low in both scenarios. The average and median CodeBLEU scores for the *controlled scenario* are approximately 0.24 and 0.25, respectively, while for the *realistic scenario* are around 0.23 and 0.24. These relatively low values suggest that, although the inverse model can partially reconstruct the original code, it fails to completely achieve this goal in basically all the cases.

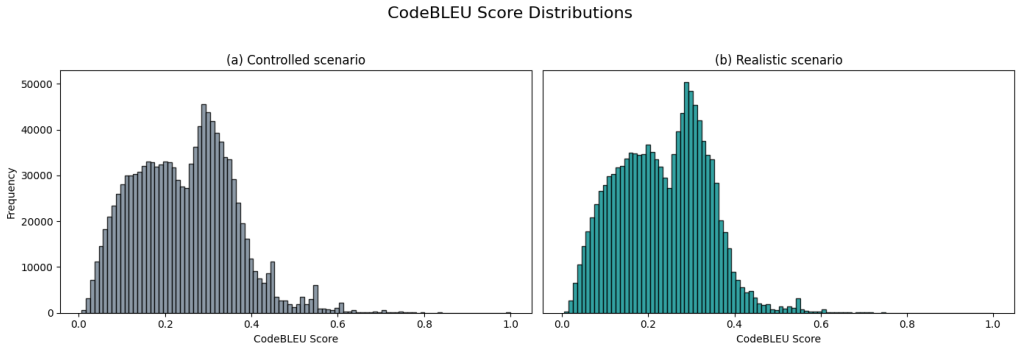

Figure 2: Histogram for the *CodeBLEU* scores evaluated in the controlled scenario (a) and in the realistic scenario (b).

Scenario	Mean	Median	#Outliers	#Maximum Score
Controlled	0.244	0.246	8,134	47
Realistic	0.233	0.236	3,808	7

Table 2: Summary statistics for the CodeBLEU scores.

We report in Figure 3 the comparison of the *CodeBLEU* results achieved in the two scenarios. As expected, the scores achieved in a more realistic scenario are lower. Surprisingly, however, the difference is marginal (almost non-existent). This result is confirmed when looking at the summary statistics of *CodeBLEU* achieved in both scenarios (Table 2).

It is worth noting that, despite the generally low scores, a number of (positive) outliers with higher *CodeBLEU* values can still be observed. As expected, the highest concentration appears in the *controlled scenario*, with 8,134 cases, compared to 3,808 in the *realistic* one.

Among these, 218 instances exceed a score of 0.8 in the first setting, indicating strong similarity to the reference code, whereas only 37 such cases are found in the second one. Remarkably, 47 of the controlled outliers achieve a perfect score of 1.0, while this occurs in just 7 instances under realistic conditions. The results in terms of *ROUGE-L* [28] and *METEOR* [29] are shown in Figure 4. Similarly to what happens for *CodeBLEU*, both *ROUGE-L* and *METEOR* exhibit higher reconstruction capabilities in the *controlled scenario* as compared to the *realistic* one. In both cases, however, the score distributions remain heavily skewed towards low values, confirming that full code reconstruction is a challenging task for the inverse model. At the same time, *ROUGE-L* and *METEOR* exhibit a larger proportion of samples with scores close to their maximum (e.g., 1) (Figure 4). Note that a score of 1 means that the generated code and the reference code are identical for what the metric at hand concerns. To understand why, in some cases, the code generated is identical to the reference one according to *METEOR* and *ROUGE-L* and not for *CodeBLEU*, we manually checked some examples. We found that this discrepancy is mainly due to the greater strictness with which *CodeBLEU* is computed. *CodeBLEU* is more sensitive to structural variations in the source code (*e.g.,* the introduction of additional parentheses or the explicit qualification of fields with `this`). Even a upper/lower case character or a single white space might significantly reduce *CodeBLEU*. This, instead, does not happen with *METEOR* and *ROUGE-L*, which work on normalized text. Figures 5 and 6 show two examples in which this happens.

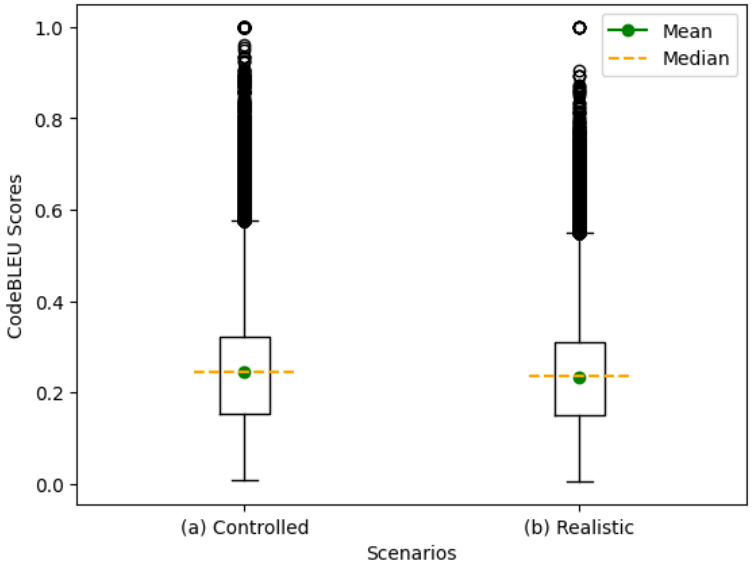

Figure 3: Box plot for the *CodeBLEU* scores evaluated in the controlled scenario (a) and in the realistic scenario (b).

5.2. RQ_2. *Data Reconstruction from Partially Reconstructed Methods*

We report the results of our manual analysis in Table 3. There is a clear distinction between the two scenarios. The *Controlled scenario* exhibits higher percentages of positive (*yes*) responses across all questions, except for the one related to the reconstruction of the conditions. This confirms that the access to the original comments significantly improves the likelihood of reconstructing meaningful structural elements. This gap is particularly evident for the analysis in terms of reconstruction of the APIs adopted, both internal ($Q_{API-int}$) and external ($Q_{API-ext}$). These results suggest that even small variations introduced in the query can substantially hinder the inverse model's ability to recover semantically aligned code fragments.

An example of instance correctly reconstructed in the *realistic scenario* but not in the *con-*

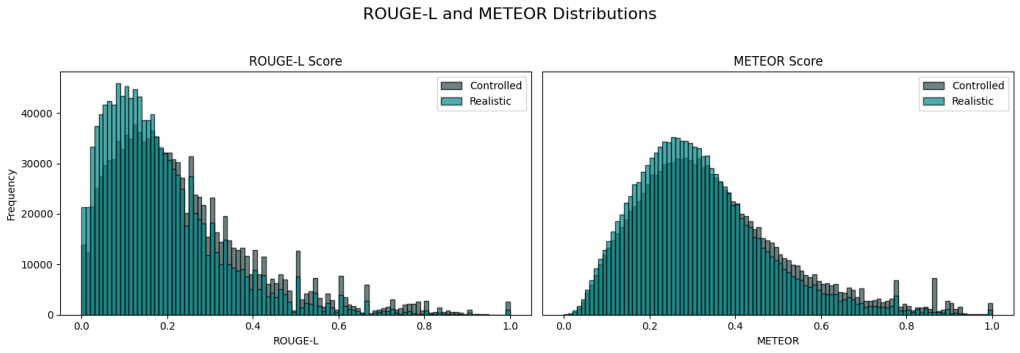

Figure 4: *ROUGE-L* and *METEOR* score distributions in both controlled and realistic scenarios

Code in the training set

```
public boolean isTrue() {
    return value == true;
}
```

Code reconstructed (controlled scenario)

```
public boolean isTrue(){
  return (value == true);
}
```

Figure 5: Example with *CodeBLEU* = 0.391 (determined by *ngram_match_score* = 0.065, *weighted_ngram_match_score* = 0.070, *syntax_match_score* = 0.430, *dataflow_match_score* = 0), *ROUGE-L* = 1.0, and *METEOR* = 0.922.

Code in the training set

```
public void inputMethodTextChanged(
      InputMethodEvent e) {
        ((InputMethodListener)a).
      inputMethodTextChanged(e);
        ((InputMethodListener)b).
      inputMethodTextChanged(e);
    }
```

Code reconstructed (realistic scenario)

```
public void inputMethodTextChanged(
      InputMethodEvent e){
  ((InputMethodListener)a).
      inputMethodTextChanged(e);
  ((InputMethodListener)b).
      inputMethodTextChanged(e);
}
```

Figure 6: Example with *CodeBLEU* = 0.620 (determined by *ngram_match_score* = 0.238, *weighted_ngram_match_score* = 0.240, *syntax_match_score* = 1.0, *dataflow_match_score* = 1.0), *ROUGE-L* = 1.0, and *METEOR* = 1.0.

trolled scenario is presented in Figure 7. This shows how the adversary, using a different sentence from the one in the training set, can actually better reconstruct the code snippet. In this particular case, we can see how the reconstructed code obtained with the paraphrased comment keeps the same sequence of instructions and also the same conditions. Indeed, the original code (i) checks if the queue is empty, (ii) if it is empty, the method returns null, otherwise (iii) it returns the element in position `start` of the queue, so at the beginning. The reconstructed code obtained in the *controlled scenario* also does the same task but without checking whether the queue is empty or not. Instead, the code obtained from the *realistic scenario* also performs this check.

We report in Figure 8 an example of a reconstruction of the internal APIs defined in the project. In this case, the attack only succeeded in the controlled scenario, while it failed in the realistic one. Specifically, the generated code suggests to an adversary that there is a class in the project named `ChangePasswordCriteria`. This piece of information might help the adversary figure out the internal design of the system and even hint at possible vulnerabilities that can be exploited.

Finally, we depict in Figure 9 an example in which the adversary successfully recovered the assignment values of the variables (variable `count`, specifically). It is worth noting that, also in this case, the attack only succeeded in the controlled scenario and failed in the realistic one.

		Q_{CF}	Q_{cond}	$Q_{API-int}$	$Q_{API-ext}$	Q_{var}
Contr.	Yes	**10.9%**	2.3%	**4.6%**	**18.9%**	**9.1%**
	Partial		5.1%			
	No	89.1%	42.3%	62.9%	44.0%	50.9%
Real.	Yes	6.3%	**4.0%**	3.4%	11.4%	5.7%
	Partial		3.4%			
	No	93.7%	42.3%	64.0%	51.4%	54.3%

Table 3: Results of RQ_2 in terms of the percentage of the response distribution for the questions across both scenarios. The instances no belonging to any category have received a NA evaluation (not reported in the table).

Code in the training set

```
synchronized public I Message peek(){
    if (numberOfElements == 0)
        return null;
    else
        return queue[start];
}
```

Code reconstructed (controlled scenario)

```
// peeks the message at the head of the queue
public Message peek(){
    return queue.peek();
}
```

Code reconstructed (realistic scenario)

```
// the code is accessing the message at the front of the queue
private void accessFront(){
    if (queue.size() == 0) {
        return;
    }
    Message m=queue.get(0);
    System.out.println("Message at front of the queue is " + m);
}
```

Figure 7: Example of code reconstructed with same control flow (Q_{CF}) only in the realistic scenario and not in the original one.

Code in the training set

```
public ChangePasswordCriteria getChangePasswordCriteria() {
    if ( changePasswordCriteria == null) {
        changePasswordCriteria = new ChangePasswordCriteria();
    }
    return changePasswordCriteria;
}
```

Code reconstructed (controlled scenario)

```
// gets the change password criteria
private static List<ChangePasswordCriteria> getChangePasswordCriteria(final User user) {
    val criteria = new ArrayList<ChangePasswordCriteria>();
    LOGGER.debug("Getting criteria for user []", user);
    val id = user.getId();
    if (StringUtils.isNotBlank(id)) {
        // ...
        criteria.add(new ChangePasswordCriteria(id, passwordSetCriteria));
    }
    LOGGER.debug("Returning criteria []", criteria);
    return criteria;
}
```

Code reconstructed (realistic scenario)

```
// retrieves the requirements for changing the password
private static List<String> getPasswordRequirements(final String password) {
    final List<String> requirements = new ArrayList<>();
    if (password != null && !password.isEmpty()) {
        final List<String> tokens = new ArrayList<>();
        for (final char c : password.toCharArray()) {
            //...
            if (!requiredTokens.isEmpty()) {
                requirements.add(String.format("* %s", StringUtils.join(requiredTokens, ",")));
            }
        }
    }
}
```

Figure 8: Example of reconstruction of developer-defined APIs in the controlled scenario ($Q_{API-int}$).

Code in the training set

```
public void incrementProfile(RuleClauseCode clause) {
    if (profile != null) {
        String index = clause.toString();
        Count count = profile.get(index);
        if (count == null) {
            profile.put(index, new Count(clause).inc());
        } else {
            count.inc();
        }
    }
}
```

Code reconstructed (controlled scenario)

```
// record a rule invocation in the profile count
public void recordRuleInvocation(String rule, int count) {
    if (rule != null) {
        int profileCount = this.profileCount.get();
        if (profileCount == 0) {
            this.profileCount.set(1);
        } else {
            profileCount++;
        }
        this.profileCount.set(profileCount);
    }
}
```

Code reconstructed (realistic scenario)

```
// keep track of the number of times a rule is called in the profile count
public static void ruleCalled(final String rule, final int count) {
    if (rule == null) {
        throw new IllegalArgumentException("rule cannot be null");
    }
    if (count <0) {
        throw new IllegalArgumentException("count must be >0");
    }
    INSTANCE.profileCount.put(rule, count);
}
```

Figure 9: Example of variable initialization reconstruction (Q_{var}) in the controlled scenario.

6. Discussion

In this work, we investigated the vulnerability of a code model — specifically, *CodeT5+* trained for the Code Summarization task — to Model Inversion Attacks. Our study specifically explored to what extent it is possible to reconstruct training samples accessing only the output of the target model, and whether partially reconstructed samples can still reveal sensitive or meaningful information.

Our results indicate that (i) a non-negligible portion of the training set can be recovered from the training set, especially in the *Controlled Scenario*, and (ii) even when the inverse model fails to recover the entire method, it frequently succeeds in reconstructing fragments of the original code (including API usage patterns and other project-specific information).

To further understand the reasons why the inverse model failed to extract any useful information, we performed a deeper manual analysis. We started from the same sample adopted to answer RQ_2, selected the instances for which we answered *No* or *NA* to all the questions in RQ_2, and manually checked those cases. Specifically, we first checked whether the input comment or paraphrase (depending on the scenario) was semantically consistent with the original code. Discrepancies between the input and the original method often explained reconstruction failures, as an inconsistent description would hinder the inverse model's recovery of the original code. If this was not a problem, we assessed whether the input description accurately reflected the original method, *i.e.,* if the comment was sufficiently informative. A correct alignment between

Scenario	Code-Com. Inc.	Gen. Com.	Model Error	Part. Rec.
Controlled	11.4%	49.7%	1.7%	37.1%
Realistic	17.1%	56.0%	3.4%	23.4%

Table 4: Results of the manual investigation in terms of percentage of *partially reconstructed* samples (*Part. Rec.*), instances with *code-comment inconsistencies* (*Code-Com. Inc.*) *generic or ambiguous comments* (*Gen. Com.*), and generic errors made by the model (*Model Error*).

```
// Original: remove all outputs from picking
// Paraphrased: delete all the results from the selection
public void removeOutputFromPicking(){
    for (int i=0; i<getOutputLength();i++)
        getOutput(i).setIsPickable(false);
}
```

Figure 10: Example of inconsistency between the paraphrased comment and the original code.

the comment and the original code does not necessarily imply that the comment is sufficiently informative for guiding the generation of the code itself (it could be too generic or overly abstract). Finally, we identified cases where the inverse model generated empty methods or entered generation loops, highlighting further limitations of the reconstruction process. We report the results of such an analysis in Table 4. We found that a *generic and/or ambiguous comment* caused the failed reconstruction in the majority of the failed cases. As expected, this phenomenon was more frequent in the *realistic setting*, where the generated paraphrases often replaced specific terms with broader or synonymous expressions. For example, in one of the instances the original comment was *"see if the digit needs to be checked"*, which clearly refers to a single digit. On the other hand, the paraphrased version uses the more generic term "number" (*"check if the number needs to be verified."*), potentially including numbers made up of multiple digits. The impact of this can be observed in a significantly higher percentage of inconsistencies between source code and comment in the realistic scenario (from ~11% to 17%) and in a slightly higher percentage of generic comments (from ~50% to almost 57%). An example is presented in Figure 10. Even if the *original* comment and the *paraphrased* one are semantically equivalent, the second one is not representative of the original code.

We also identified a case of perfect reconstruction in the *realistic scenario*. This is illustrated in Figure 11, where the only observed difference lies in the name of the array, while the logic and structure of the code remain unchanged.

Our results clearly suggest that there is concerning security risk for scenarios where training data contains proprietary or confidential code, as attackers could potentially infer the design logic, specific API usage, or control flow of unseen code simply by querying the target model with natural language inputs.

7. Threats to Validity

Threats to Construct Validity. We assumed the adversary could access the tokenizer to encode and decode raw input and output data. An adversary might not have such a piece of information, especially if the tokenizer is not open-source.

Code in the training set

```
protected boolean arrayContains(int[] ary, int value) {
    for (int i = 0; i < ary.length; i++) {
          if (ary[i] == value) {
            return true;
          }
    }
    return false;
}
```

Code reconstructed (controlled scenario)

```
// checks to see if the int array contains the specified value utility method
public boolean containsValueUtilityMethod(int[] values){
      for (  int valueUtilityMethod : values) {
            if (this.values.contains(valueUtilityMethod)) {
              return true;
            }
      }
      return false;
}
```

Code reconstructed (realistic scenario)

```
// this utility method verifies whether the integer array includes the specified value
public static boolean includes(int[] intArray, int value) {
    for (int i = 0; i < intArray.length; i++) {
        if (intArray[i] == value) {
            return true;
        }
    }
    return false;
}
```

Figure 11: Example of perfect reconstruction in the *realistic scenario*.

Threats to Internal Validity. A first threat involves the use of similar code distributions between the training set of the target model and the auxiliary set. If the inverse model is exposed to code that is structurally or semantically similar to the target set, this may inflate performance and overestimate the reconstruction ability of the model. Our experimental results are based on the fact that the adversary knows and can use the architecture of the target model (CodeT5+, in our case). It is possible that using different architectures for the target and inverse models negatively affects the results. The sample we manually analyzed to answer RQ_2 contains 175 instances, labeled for both the scenarios, leading to 350 evaluations. We decided not to use a larger sample due to the significant effort required to perform such an evaluation in both the scenarios. Analyzing different samples might have led to different results. Specifically, the margin of error is 7.4%. This means that a different sample might lead to a different distribution of the information retrieved (*e.g.,* more instances reconstructing the control flow and less reconstructing external APIs). However, we can confidently say that the number of instances that expose *at least* a problem is significant (32.0±7.4% for the controlled scenario and 18.9±7.4% for the realistic scenario), thus our main finding remains valid.

There are two threats related to the model we adopted to generate the paraphrases in the realistic scenario. First, we only used a single model (GPT-3.5), and second, we used the default temperature setting (0.7). There is a risk that this configuration is suboptimal to generate semantically-equivalent paraphrases of the original comments. To mitigate this threat, we replicated the attack on the same 175 samples used for the manual evaluation in RQ_2. We used two models, *i.e.,* GPT-3.5 and GPT-4.1, and two temperature settings for each of them: the default value suggested for the model — 0.7 for GPT-3.5 and 1 for GPT-4.1 — and one that allows to generate the output in a (quasi-)deterministic way — *i.e.*, 0. We also manually wrote paraphrases

Scenario	Temperature		Results				
			Q_{CF}	Q_{cond}	$Q_{API\text{-}int}$	$Q_{API\text{-}ext}$	Q_{var}
Controlled	–	Yes	**10.9**%	2.3%	**4.6**%	**18.9**%	**9.1**%
		Partial		5.1%			
		No	89.1%	42.3%	62.9%	44.0%	50.9%
GPT-3.5	Default (0.7)	Yes	6.3%	4.0%	3.4%	11.4%	5.7%
		Partial		3.4%			
		No	93.7%	42.3%	64.0%	51.4%	54.3%
	Deterministic (0)	Yes	6.9%	4.0%	2.9%	10.3%	8.6%
		Partial		1.7%			
		No	93.1%	44.0%	64.6%	52.6%	51.4%
GPT-4.1	Default (1)	Yes	6.9%	**4.6**%	2.3%	14.9%	6.9%
		Partial		2.9%			
		No	93.1%	42.3%	65.1%	48.0%	53.1%
	Deterministic (0)	Yes	5.7%	**4.6**%	2.3%	17.1%	7.4%
		Partial		1.7%			
		No	94.3%	43.4%	65.1%	45.7%	52.6%
Manual	–	Yes	8.6%	2.3%	2.3%	16.6%	6.9%
		Partial		2.3%			
		No	91.4%	45.1%	65.1%	46.3%	53.1%

Table 5: Outcomes of the manual analysis by model and temperature.

for each instance to have an upper bound for the results.
Table 5 reports the results of such a replication and also compare the results to the ones obtained in the controlled scenario (*i.e.,* with the original comment). No single configuration emerges as distinctly superior, and none is better than the reconstructions obtained from the controlled scenario. However, scenarios involving GPT-4.1 and the manually written paraphrases achieve slightly higher scores in $Q_{API\text{-}ext}$, while for all other evaluated categories, the performance differences as compared to the analyzed realistic scenario (GPT-3.5 with t=0.7) are negligible.
In Figure 12 there is an example where paraphrases generated with GPT-4.1 and handcrafted paraphrases let the inverse model to recreate the exact same function `contains()`. The paraphrases are very similar (it is not possible to identify a superior one), but these slight differences in wording lead to corresponding small variations in the reconstructed code.

Threats to External Validity. Our experimental results are related to a specific model (*i.e., CodeT5+*) and on an inverse model with the same architecture and hyperparameters of the previous one. Results may vary in scenarios in which a different model is adopted (*e.g.,* DeepSeek Coder [53]). Besides, our analysis focuses on Java methods. The results could not generalize to other programming languages or application domains.

8. Conclusion and Future Work

We presented an initial exploration of performing a *Reconstruction Attack* on an LLM designed for a coding task – specifically, code summarization – with the objective of reconstructing the source code used for training. The results indicate that the attack fails on most instances,

Target Source Code

```
public boolean containsAttribute(T attribute) {
  return attributes.contains(attribute);
}
```

Original Comment (Controlled Scenario)

```
//checks whether this description contains a
     given attribute
public boolean hasAttribute(Attribute attribute)
      {
  return attributes.containsKey(attribute);
}
```

Paraphrased (Manual)

```
//this function verifies if the given
     description includes a specified attribute
public static boolean containsAttribute(String
     description,String attributeName) {
  return description.contains(attributeName);
}
```

Paraphrased (GPT-3.5, t = 0.7)

```
//verifies if the description includes a
     specified attribute.
public boolean includesAttribute(String
     attributeName) {
  return attributes.containsKey(attributeName);
}
```

Paraphrased (GPT-3.5, t = 0)

```
//Verifies if the description includes a
     specified attribute.
public boolean verify(String attr){
  return attributes.containsKey(attr);
}
```

Paraphrased (GPT-4.1, t = 1)

```
//verifies if this description includes a
     specified attribute
public boolean includesAttribute(Attribute
     attribute) {
  return attributes.contains(attribute);
}
```

Paraphrased (GPT-4.1, t = 0)

```
//verifies if this description includes a
     specified attribute
public boolean includesAttribute(Attribute
     attribute) {
  return attributes.contains(attribute);
}
```

Figure 12: Example in which GPT-4.1–generated paraphrases and manually crafted paraphrases enable the reconstruction of a method closer to the target implementation.

suggesting that state-of-the-art LLMs for coding tasks largely maintain confidentiality and do not fully disclose source code. However, we observed that some private information (such internal APIs adopted) can be successfully reconstructed. Our future research agenda includes simulating such attacks in a more realistic scenario (*i.e.,* by manually crafting the attack queries instead of using the original comments), testing the effectiveness of training methods for LLMs that further reduce the likelihood of successful reconstruction attacks, and extending the evaluation to programming languages other than Java, in order to assess whether language-specific characteristics may influence the feasibility and impact of the attack.

Another interesting direction is to investigate whether existing defense strategies could be effective in our specific setting [54, 30]. These include mechanisms that perturb or obfuscate feature representations or modify gradients ([55, 56]) to reduce the leakage of sensitive information. Another well-studied class is differential privacy (DL) [57, 58, 59, 60], which injects carefully calibrated noise during training or inference to limit the risk of data reconstruction. Beyond these, there are also approaches based on cryptographic encryption ([61, 62]), which secure data and model parameters during storage or transmission, techniques that work on a more robust training process ([63, 64, 65]) or on secure fine-tuning ([66, 67]). Evaluating the adaptability of these categories to our scenario could yield valuable insights into practical privacy-preserving strategies for code-processing models.

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used ChatGPT in order to improve language and readability. After using this tool, the authors reviewed and edited the content as needed and take

full responsibility for the content of the publication.

Data Availability

We publicly release our replication package [68], in which we provide our datasets, the scripts for building and everything needed to replicate all the results of our experiment.

Acknowledgements

This work was supported by the Italian Ministry of University and Research through the project "SOP: Securing sOftware Platform" (CUP: H73C22000890001), as part of the initiative "SERICS: Security and Rights in CyberSpace" (Project No. PE00000014 - CUP: B43C22000750006), and by the PRIN 2022 project "QualAI: Continuous Quality Improvement of AI-based Systems" (CUP: H53D23003520006).

References

[1] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, G. Bavota, Studying the usage of text-to-text transfer transformer to support code-related tasks, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 336–347.

[2] I. Bouzenia, P. Devanbu, M. Pradel, Repairagent: An autonomous, llm-based agent for program repair, arXiv preprint arXiv:2403.17134 (2024).

[3] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, X. Yang, Thinkrepair: Self-directed automated program repair, in: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024, pp. 1274–1286.

[4] W. Sun, Y. Miao, Y. Li, H. Zhang, C. Fang, Y. Liu, G. Deng, Y. Liu, Z. Chen, Source code summarization in the era of large language models, in: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), IEEE Computer Society, 2024, pp. 419–431.

[5] T. Ahmed, K. S. Pai, P. Devanbu, 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] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, arXiv preprint arXiv:2109.00859 (2021).

[7] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, Y. Ma, Exploring and evaluating hallucinations in llm-powered code generation, arXiv preprint arXiv:2404.00971 (2024).

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Advances in neural information processing systems 30 (2017).

[9] W. Hou, Z. Ji, Comparing large language models and human programmers for generating programming code, Advanced Science 12 (2025) 2412279.

[10] F. F. Xu, U. Alon, G. Neubig, V. J. Hellendoorn, A systematic evaluation of large language models of code, in: Proceedings of the 6th ACM SIGPLAN international symposium on machine programming, 2022, pp. 1–10.

[11] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, E. Aftandilian, Measuring github copilot's impact on productivity, Communication of ACM 67 (2024) 54–63.

[12] P. Lepagnol, T. Gerald, S. Ghannay, C. Servan, S. Rosset, Small language models are good too: An empirical study of zero-shot classification, arXiv preprint arXiv:2404.11122 (2024).

[13] M. J. J. Bucher, M. Martini, Fine-tuned'small'llms (still) significantly outperform zero-shot generative ai models in text classification, arXiv preprint arXiv:2406.08660 (2024).

[14] M. Hassid, T. Remez, J. Gehring, R. Schwartz, Y. Adi, The larger the better? improved llm code-generation via budget reallocation, arXiv preprint arXiv:2404.00725 (2024).

[15] A. Al-Kaswan, M. Izadi, A. Van Deursen, Traces of memorisation in large language models for code, in: IEEE/ACM International Conference on Software Engineering (ICSE), 2024, pp. 1–12.

[16] M. Rigaki, S. Garcia, A survey of privacy attacks in machine learning, ACM Computing Surveys 56 (2023) 1–34.

[17] Y. Liu, G. Deng, Y. Li, K. Wang, Z. Wang, X. Wang, T. Zhang, Y. Liu, H. Wang, Y. Zheng, et al., Prompt injection attack against llm-integrated applications, arXiv preprint arXiv:2306.05499 (2023).

[18] P. Maini, H. Jia, N. Papernot, A. Dziedzic, Llm dataset inference: Did you train on my dataset?, Advances in Neural Information Processing Systems 37 (2024) 124069–124092.
[19] P. T. Nguyen, C. Di Sipio, J. Di Rocco, M. Di Penta, D. Di Ruscio, Adversarial attacks to api recommender systems: Time to wake up and smell the coffee?, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2021, pp. 253–265.
[20] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, D. Lo, Unveiling memorization in code models, in: IEEE/ACM International Conference on Software Engineering (ICSE), 2024, pp. 856–856.
[21] W. Cheng, K. Sun, X. Zhang, W. Wang, Security attacks on llm-based code completion tools, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 39, 2025, pp. 23669–23677.
[22] M. Russodivito, A. Spina, S. Scalabrino, R. Oliveto, Black-box reconstruction attacks on llms: A preliminary study in code summarization, in: International Conference on the Quality of Information and Communications Technology, Springer, 2024, pp. 391–398.
[23] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, S. C. Hoi, Codet5+: Open code large language models for code understanding and generation, arXiv preprint arXiv:2305.07922 (2023).
[24] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt, Codesearchnet challenge: Evaluating the state of semantic code search, preprint (arXiv:1909.09436) (2019).
[25] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, Z. Jin, Summarizing source code with transferred api knowledge, in: International Joint Conference on Artificial Intelligence (IJCAI), 2018, p. 2269–2275.
[26] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin, Deep code comment generation, in: IEEE/ACM International Conference on Program Comprehension (ICPC), 2018, pp. 200–210.
[27] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, S. Ma, Codebleu: a method for automatic evaluation of code synthesis, arXiv preprint arXiv:2009.10297 (2020).
[28] C.-Y. Lin, Rouge: A package for automatic evaluation of summaries, in: Text summarization branches out, 2004, pp. 74–81.
[29] S. Banerjee, A. Lavie, Meteor: An automatic metric for mt evaluation with improved correlation with human judgments, in: Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization, 2005, pp. 65–72.
[30] H. Fang, Y. Qiu, H. Yu, W. Yu, J. Kong, B. Chong, B. Chen, X. Wang, S.-T. Xia, K. Xu, Privacy leakage on dnns: A survey of model inversion attacks and defenses, arXiv preprint arXiv:2402.04013 (2024).
[31] X. Wu, M. Fredrikson, S. Jha, J. F. Naughton, A methodology for formalizing model-inversion attacks, in: 2016 IEEE 29th computer security foundations symposium (CSF), IEEE, 2016, pp. 355–370.
[32] M. Fredrikson, E. Lantz, S. Jha, S. Lin, D. Page, T. Ristenpart, Privacy in pharmacogenetics: An {End-to-End} case study of personalized warfarin dosing, in: USENIX Security Symposium, 2014, pp. 17–32.
[33] M. Fredrikson, S. Jha, T. Ristenpart, Model inversion attacks that exploit confidence information and basic countermeasures, in: ACM SIGSAC Conference on Computer and Communications Security (CCS), 2015, pp. 1322–1333.
[34] B. Hitaj, G. Ateniese, F. Perez-Cruz, Deep models under the gan: information leakage from collaborative deep learning, in: ACM SIGSAC Conference on Computer and Communications Security (CCS), 2017, pp. 603–618.
[35] S. Chen, M. Kahla, R. Jia, G.-J. Qi, Knowledge-enriched distributional model inversion attacks, in: Proceedings of the IEEE/CVF international conference on computer vision, 2021, pp. 16178–16187.
[36] K.-C. Wang, Y. Fu, K. Li, A. Khisti, R. Zemel, A. Makhzani, Variational model inversion attacks, Advances in Neural Information Processing Systems 34 (2021) 9706–9719.
[37] X. Yuan, K. Chen, J. Zhang, W. Zhang, N. Yu, Y. Zhang, Pseudo label-guided model inversion attack via conditional generative adversarial network, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 37, 2023, pp. 3349–3357.
[38] N.-B. Nguyen, K. Chandrasegaran, M. Abdollahzadeh, N.-M. Cheung, Re-thinking model inversion attacks against deep neural networks, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2023, pp. 16384–16393.
[39] Z. Yang, J. Zhang, E.-C. Chang, Z. Liang, Neural network inversion in adversarial setting via background knowledge alignment, in: ACM SIGSAC Conference on Computer and Communications Security (CCS), 2019, pp. 225–240.
[40] X. Zhao, W. Zhang, X. Xiao, B. Lim, Exploiting explanations for model inversion attacks, in: Proceedings of the IEEE/CVF international conference on computer vision, 2021, pp. 682–692.
[41] M. Kahla, S. Chen, H. A. Just, R. Jia, Label-only model inversion attacks via boundary repulsion, in: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2022, pp. 15045–15053.
[42] C. Song, A. Raghunathan, Information leakage in embedding models, in: Proceedings of the 2020 ACM SIGSAC conference on computer and communications security, 2020, pp. 377–390.
[43] H. Li, M. Xu, Y. Song, Sentence embedding leaks more information than you expect: Generative embedding inversion attack to recover the whole sentence, arXiv preprint arXiv:2305.03010 (2023).
[44] F. Salerno, A. Al-Kaswan, M. Izadi, How much do code language models remember? an investigation on data

extraction attacks before and after fine-tuning, arXiv preprint arXiv:2501.17501 (2025).
[45] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, BLEU: a method for automatic evaluation of machine translation, in: Annual Meeting of the Association for Computational Linguistics (ACL), 2002, pp. 311–318.
[46] R. Haldar, J. Hockenmaier, Analyzing the performance of large language models on code summarization, arXiv preprint arXiv:2404.08018 (2024).
[47] I. Tarnowski, How to use cyber kill chain model to build cybersecurity?, European Journal of Higher Education IT (2017).
[48] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu, Exploring the limits of transfer learning with a unified text-to-text transformer, Journal of Machine Learning Research (JMLR) 21 (2020) 5485–5551.
[49] L. Shi, F. Mu, X. Chen, S. Wang, J. Wang, Y. Yang, G. Li, X. Xia, Q. Wang, Are we building on the rock? on the importance of data preprocessing for code summarization, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 107–119.
[50] A. LeClair, S. Jiang, C. McMillan, A neural model for generating natural language summaries of program subroutines, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 795–806.
[51] K. Krishna, Y. Song, M. Karpinska, J. Wieting, M. Iyyer, Paraphrasing evades detectors of ai-generated text, but retrieval is an effective defense, Advances in Neural Information Processing Systems 36 (2023) 27469–27500.
[52] L. Hickman, P. D. Dunlop, J. L. Wolf, The performance of large language models on quantitative and verbal ability tests: Initial evidence and implications for unproctored high-stakes testing, International Journal of Selection and Assessment 32 (2024) 499–511.
[53] 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).
[54] W. Yang, S. Wang, D. Wu, T. Cai, Y. Zhu, S. Wei, Y. Zhang, X. Yang, Z. Tang, Y. Li, Deep learning model inversion attacks and defenses: a comprehensive survey, Artificial Intelligence Review 58 (2025) 242.
[55] S. Gade, N. H. Vaidya, Privacy-preserving distributed learning via obfuscated stochastic gradients, in: 2018 IEEE Conference on Decision and Control (CDC), IEEE, 2018, pp. 184–191.
[56] J. Geng, Y. Mou, Q. Li, F. Li, O. Beyan, S. Decker, C. Rong, Improved gradient inversion attacks and defenses in federated learning, IEEE Transactions on Big Data 10 (2023) 839–850.
[57] C. Dwork, Differential privacy, in: International colloquium on automata, languages, and programming, Springer, 2006, pp. 1–12.
[58] C. Dwork, Differential privacy: A survey of results, in: International conference on theory and applications of models of computation, Springer, 2008, pp. 1–19.
[59] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, L. Zhang, Deep learning with differential privacy, in: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, 2016, pp. 308–318.
[60] T. Ha, T. K. Dang, T. T. Dang, T. A. Truong, M. T. Nguyen, Differential privacy in deep learning: an overview, in: 2019 International Conference on Advanced Computing and Applications (ACOMP), IEEE, 2019, pp. 97–102.
[61] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, J. Wernsing, Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy, in: International conference on machine learning, PMLR, 2016, pp. 201–210.
[62] P. Prakash, J. Ding, H. Li, S. M. Errapotu, Q. Pei, M. Pan, Privacy preserving facial recognition against model inversion attacks, in: GLOBECOM 2020-2020 IEEE Global Communications Conference, IEEE, 2020, pp. 1–6.
[63] T. Wang, Y. Zhang, R. Jia, Improving robustness to model inversion attacks via mutual information regularization, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 35, 2021, pp. 11666–11673.
[64] X. Peng, F. Liu, J. Zhang, L. Lan, J. Ye, T. Liu, B. Han, Bilateral dependency optimization: Defending against model-inversion attacks, in: Proceedings of the 28th ACM SIGKDD conference on knowledge discovery and data mining, 2022, pp. 1358–1367.
[65] L. Struppek, D. Hintersdorf, K. Kersting, Be careful what you smooth for: Label smoothing can be a privacy shield but also a catalyst for model inversion attacks, arXiv preprint arXiv:2310.06549 (2023).
[66] X. Gong, Z. Wang, S. Li, Y. Chen, Q. Wang, A gan-based defense framework against model inversion attacks, IEEE Transactions on Information Forensics and Security 18 (2023) 4475–4487.
[67] D. Yu, S. Naik, A. Backurs, S. Gopi, H. A. Inan, G. Kamath, J. Kulkarni, Y. T. Lee, A. Manoel, L. Wutschitz, et al., Differentially private fine-tuning of language models, arXiv preprint arXiv:2110.06500 (2021).
[68] A. Spina, M. Russodivito, S. Scalabrino, R. Oliveto, Replication Package of "Peeking Inside the Black Box: Training Data Exposure in Code Language Models", 2025. URL: `https://figshare.com/s/cde6e04ec67f24fc3ee5`.