Empirical Software Engineering (2027) 32:50
https://doi.org/10.1007/s10664-026-10966-5

RESEARCH PAPERS

The More You Tune, the More They Spill?

An empirical study of data leakage from code models under fine-tuning, quantization, and parameter-efficient adaptation

Fabio Salerno[1] · **Ali Al-Kaswan**[1] · **Maliheh Izadi**[1]

Received: 30 September 2025 / Accepted: 4 September 2026
© The Author(s) 2026

Abstract

Although previous research has established that *pre-trained* code models can regurgitate their training data via extraction attacks, the privacy implications of the increasingly common practice of *fine-tuning* remain underexplored. Fine-tuning enables entities to adapt models to specialised tasks using smaller, curated datasets that may contain proprietary assets or sensitive information. This study provides a comparative investigation into data extractability, before and after this fine-tuning stage. We develop a benchmark to systematically assess the vulnerability of pre-training and fine-tuning samples to extraction attacks. Our experiments on the Starcoder2 models yield two key findings. First, fine-tuning significantly reduces the extractability of the model's original pre-training data, decreasing the extraction rate from 54.9% to 23.5%. Second, this benefit is offset by the increased vulnerability in the fine-tuning data. Smaller models, in particular, show a strong propensity to leak this data. Extending this analysis, we find that moderate *model compression* generally preserves the same memorization patterns, whereas int4 quantization reduces absolute pre-training memorization but amplifies relative fine-tuning leakage. We further find that parameter-efficient fine-tuning mitigates fine-tuning data memorization compared to full fine-tuning, while exhibiting smoother forgetting of pre-training data. A qualitative analysis of 2,000 samples reveals that *data structures* and *licensing information* are the most frequently memorized categories. Our work highlights that fine-tuning introduces a privacy trade-off, shifting the risk from vast public pre-training data to smaller, higher-value proprietary fine-tuning data. Model deployers should evaluate these risks and explore strategies to mitigate leakage while preserving performance.

Keywords Security & privacy · LLMs for code · Memorization · Fine-tuning · Parameter-efficient fine-tuning

Fabio Salerno and Ali Al-Kaswan are the authors contributed equally to this work.

Communicated by: Ayushi Rastogi, Olga Baysal, Bram Adams.

✉ Ali Al-Kaswan
a.al-kaswan@tudelft.nl

[1] Delft University of Technology, Delft, The Netherlands

1 Introduction

Large Language Models (LLMs) have become increasingly popular for their ability to generate human-like text and their potential applications across various fields, including Software Engineering (Hou et al. 2024; Lu et al. 2021; Izadi et al. 2024). Despite their immense popularity, LLMs for code are commonly trained on *unsanitized* datasets of source code scraped from the Internet. Previous work has revealed that the content of these datasets can be memorized and extracted by attackers through various data extraction techniques (Carlini et al. 2021; Biderman et al. 2023; Al-Kaswan et al. 2024; Ishihara 2023; Yang et al. 2024).

While LLM pre-training typically occurs infrequently due to their enormous size, fine-tuning is becoming the dominant approach for end-users to customize models. Fine-tuning often involves small curated datasets (Bogomolov et al. 2022), which may contain proprietary or sensitive information (Al-Kaswan and Izadi 2023; Al-Kaswan et al. 2024). This raises concerns about the potential for sensitive data within these datasets to be inadvertently extracted from the fine-tuned models (Mireshghallah et al. 2022b; Zeng et al. 2024; Sun et al. 2023). To our knowledge, previous studies have investigated data memorization and extraction attacks on *pre-trained* LLMs for code (Al-Kaswan et al. 2024; Zeng et al. 2024), but there has been no empirical investigation into the extractability of *fine-tuned* code models. In this work, we explore the extent to which fine-tuned LLMs for code leak data from their pre-training and fine-tuning corpora.

Directly measuring memorization in large language models is challenging, as memorized training data may not always appear verbatim in generated outputs. In this work, we distinguish between *memorization*, namely the internal retention of training examples by the model, and *data leakage*, which occurs when memorized content can be reproduced through model outputs. Since memorization is not directly observable, we estimate it through *extractability*: the ability to recover training samples. While memorization of training data can manifest as non-exact duplication (Hartmann et al. 2023), measuring the rate of data extraction provides a lower bound on data leakage in LLMs.

We first develop a custom benchmark to assess the vulnerability of pre-training and fine-tuning samples to extraction attacks and identify vulnerable samples. We then test the extractability of pre-training and fine-tuning data on the Starcoder2 model family (Lozhkov et al. 2024). Finally, two authors individually and manually categorize the extractable samples to identify specific data categories that are more exposed to memorization.

Our key results: *Fine-tuned LLMs for code tend to "forget" samples that were extractable from the pre-trained models. Additionally, smaller models are more vulnerable to fine-tuning data extraction attacks compared to larger models. Data carriers and license information are the most likely to be memorized, while the latter is the most likely to be forgotten after fine-tuning, leaving data carriers still exposed to extraction. Beyond fine-tuning, we find that lightweight model precision settings leave memorization behavior largely unchanged, while int4 precision alters this trend by lowering pre-training memorization and intensifying leakage of fine-tuning data. Finally, parameter-efficient LoRA fine-tuning mitigates fine-tuning data memorization compared to full fine-tuning, while exhibiting smoother forgetting of pre-training data.*

To summarize, the main contributions of our study are:

- An empirical assessment of pre-training data memorization in fine-tuned code models which reveals that fine-tuning reduces the extractability of certain pre-training data from the pre-trained models.
- An empirical assessment of fine-tuning data memorization in code models which shows that (1) code models memorize fine-tuning data; (2) smaller models are more affected by fine-tuning memorization; (3) increasing the duplication rate of samples from the training corpus significantly drives data leakage; (4) similar to pre-training memorization, the input prefix length has a considerable impact on fine-tuning extractability. (5) Model compression generally preserves memorization patterns, with int4 quantization in contrast, reducing pre-training memorization while amplifying fine-tuning leakage. (6) Parameter Efficient Fine-Tuning methods memorize fine-tuning data at a lower rate compared to their full fine-tuned counterparts.
- We release our code, evaluation scripts, and data used in this study to enable replication of our results (Salerno et al. 2025).[1]

2 Background and Related Work

Memorization Memorization refers to the ability of Language Models to recall details from the training data (Carlini et al. 2021, 2022a, c; Al-Kaswan and Izadi 2023; Yang et al. 2024; Hu et al. 2022b). Recent studies have highlighted this risk of regurgitating training data, some of which might contain sensitive information (Al-Kaswan and Izadi 2023; Carlini et al. 2021). For example, GitHub Copilot and Amazon CodeWhisper can be coaxed to emit hardcoded credentials captured during training, using so-called data extraction attacks (Huang et al. 2024).

Membership Inference Attacks *Memorization enables membership inference attacks (Al-Kaswan et al. 2023b; Carlini et al. 2022a, c). With a membership inference attack, an adversary aims to predict if a sample was part of the training data of a machine learning model. The adversary is only given access to the model, and no access to the training data. Membership inference attacks are a primitive of more elaborate and dangerous attacks against machine learning models, such as data extraction attacks. Shokri et al. proposed the first membership inference attack in which the authors targeted classification models (Shokri et al. 2017). Since then the field has expanded and proposed attacks target generative models, including LLMs (Carlini et al. 2022a; Mireshghallah et al. 2022a; Oh et al. 2023). In the code domain, an additional risk is raised by the presence of copyleft and other non-permissively licensed codes in training datasets (Al-Kaswan and Izadi 2023). Katzy et al. found that datasets used to train code LLMs contain non-permissively licensed code, even those that claim otherwise (Katzy et al. 2024).*

Untargeted Data Extraction Attacks *Yang et al. studied memorization in pre-trained code LLMs using an untargeted attack against CodeParrot (Yang et al. 2024). In untargeted attacks, the model is prompted with an empty string or a random token. The completion of the model is then checked for memorized sequences. The memorized sequence can be from any part of the training corpus (Carlini et al. 2021; Oh et al. 2023; Carlini et al. 2022c).*

[1] GitHub repo: https://github.com/AISE-TUDelft/LLM4Code-memtune-ext

This attack requires knowledge of the complete training data, as any random sequence can be emitted, as there is no control of the target (Al-Kaswan et al. 2023b; Henderson et al. 2023; Huang et al. 2022; Mireshghallah et al. 2022a).

Targeted Data Extraction Attacks *Carlini et al. first proposed a targeted data extraction attack and demonstrated that LMs tend to memorize and regenerate segments of training data. Kandpal et al. and Lee et al. revealed that duplicated training data is more vulnerable to memorization, and de-duplication can effectively reduce memorization (Kandpal et al. 2022; Lee et al. 2022). Carlini et al. further quantified memorization effects, exhibiting that memorization grows with model scale, data duplication, and prompt input length (Carlini et al. 2022b). Al-Kaswan et al. also measured memorization in pre-trained code LLMs but with a targeted attack. In a targeted attack, the model is supplied with partial information to complete. By sampling common code from Python files and identifying potentially extractable samples, the authors constructed a benchmark dataset (Al-Kaswan et al. 2024).*

Model Compression *Quantization techniques are increasingly adopted to make LLMs more practical for deployment, reducing memory footprint and accelerating inference. According to Wei et al. , int8 quantization enables running the full-sized CodeGen-16B model on a standard laptop CPU, reducing the time to generate a 20-token sequence from roughly 70 seconds to approximately 25 seconds, while maintaining comparable accuracy (Wei et al. 2023). However, model compression techniques can introduce new security implications. Egashira et al. demonstrated that adversaries can deliberately fine-tune and manipulate model weights so that the full-precision version exhibits benign behavior, while the corresponding quantized model deterministically activates malicious behavior (Egashira et al. 2024). For example, the full-precision StarCoder-7B generated secure code in 77.1% of cases, whereas its int8-quantized counterpart produced secure outputs in only 12.7%, as measured by CodeQL static analysis. This result illustrates that quantization, while beneficial for efficiency, may simultaneously expose models to hidden vulnerabilities (Egashira et al. 2024).*

Fine-Tuning *Many code LLMs are deployed using the pretrain-finetune paradigm. Under this paradigm, an LLM is first pre-trained with a large unlabeled dataset in an unsupervised manner. Then, the LLM is fine-tuned with a smaller task-specific dataset (Devlin et al. 2019; Wang et al. 2021; Van Dam et al. 2023). Use cases for fine-tuning include adapting a model to a specific project (Ahmed and Devanbu 2022), organization, or a different field (Al-Kaswan et al. 2023a). Additionally, LLMs can be fine-tuned with instruction data (Zhang et al. 2023).*

Parameter Efficient Fine-Tuning *Parameter Efficient Fine-Tuning (PEFT) techniques have emerged as a resource-efficient alternative to full fine-tuning of large language models, enabling the adaptation of massive pre-trained models by updating only a small fraction of parameters, thereby reducing computational costs and memory requirements (Han et al. 2024). Common PEFT methods include Low-Rank Adaptation (LoRA) (Hu et al. 2022a; Han et al. 2024), which injects low-rank matrices into the model's layers to approximate weight updates, and adapter modules, which add lightweight bottleneck layers for task-specific learning while keeping the base model frozen. These approaches address the challenges of*

traditional full fine-tuning, which involves adjusting all parameters and demands substantial computational resources, especially for adapting general-purpose LLMs to downstream tasks like those in software engineering (Han et al. 2024; Zhuo et al. 2024). By minimizing the number of trainable parameters, PEFT methods significantly lower memory usage and training time, making deep learning more accessible across domains (Han et al. 2024).

In the context of code language models, PEFT has shown particular promise in customising models to domain-specific software engineering (SE) tasks without extensive retraining (Zhuo et al. 2024). For example, the Astraios framework applies PEFT to instruction-tune code LLMs, such as a suite of 28 models based on OctoCoder (Muennighoff et al. 2023), demonstrating improved performance on code generation and completion while maintaining efficiency in resource-constrained environments (Zhuo et al. 2024). Zhuo et al. found that LoRA (Hu et al. 2022a) offers the best trade-off between model accuracy and cost, and in some cases, outperforms full fine-tuning. Similarly, Weyssow et al. compared various PEFT techniques for code completion tasks across different models, concluding that LoRA is the most effective (Weyssow et al. 2025). Notably, their work highlights that In-Context Learning methods, such as few-shot prompting, alone are less performant than PEFT methods and incur additional inference overhead, whereas PEFT avoids such runtime costs (Weyssow et al. 2025).

Memorization After Fine-Tuning *Research on the impacts of fine-tuning on memorization is restricted and it has largely focused on text-based models. Mireshghallah et al. experimented with different fine-tuning methods on GPT-2 to investigate their impact on memorization (Mireshghallah et al. 2022b). Zeng et al. found that the fine-tuning task also significantly influences the memorization rate (Zeng et al. 2024).*

Sun et al. investigate the fine-tuning API offered by GPT-3 (Brown et al. 2020). A fine-tuned GPT-3 text completion model can leak Personal Identifiable Information (PII) from its pre-training data (Sun et al. 2023).

Research on code-based LLMs is even more limited. Pappu et al. investigated memorization in Reinforcement Learning with Human Feedback (RLHF) for code completion models (Pappu et al. 2024). Memorization of pre-training data can persist through RLHF, but the memorization rate of the RLHF data is much lower (Pappu et al. 2024). To the best of our knowledge, no prior studies have examined how fine-tuning affects memorization in code LLMs.

Our Contribution *In our study, we examine fine-tuning and its impact on memorization in code models. We build on previous findings by exploring multiple factors well studied for pre-trained code models but not yet for fine-tuned ones, including input prefix length, duplication rate, and model size. We additionally investigate the extractability of pre-trained and fine-tuned models under model compression techniques and parameter-efficient fine-tuning. To complement this quantitative evaluation, we manually investigate a set of extracted data to understand the type of memorized information. While previous research primarily focused on the memorization of Python code, our work uniquely explores the memorization of Java code, which has not been studied before.*

3 Problem Definition

Previous studies have shown that LLMs trained on code tend to memorize parts of their training data (Al-Kaswan et al. 2024; Yang et al. 2024; Pappu et al. 2024). Fine-tuning these models often uses small, curated datasets, which are often proprietary and valuable. This poses additional risks of sensitive information being unintentionally exposed through the fine-tuned models (Mireshghallah et al. 2022b; Sun et al. 2023; Zeng et al. 2024). In this work, using data extractability techniques, we investigate how much code LLMs memorize the data they have seen before and after *fine-tuning*.

We examine this phenomenon for the automatic code completion task, which has emerged as a highly prevalent and valuable application of LLMs (Hou et al. 2024). Code completion also represents a setting where leakage of user data can raise legal, commercial, and privacy concerns (Al-Kaswan and Izadi 2023). To this end, we fine-tune the StarCoder2 model (Lozhkov et al. 2024) and its various available parameter sizes into a downstream task; line completion in Java. We focus on Java for several reasons. Java is widely used in enterprise environments, making it a relevant and practical choice. Furthermore, prior studies on code memorization have predominantly focused on Python and suggested that more structured languages might show different patterns of extractability (Al-Kaswan et al. 2024; Yang et al. 2024; Pappu et al. 2024). Following their recommendations, we chose to investigate Java as a new language.

In addition, we extend our analysis to techniques commonly used in practice when deploying or adapting large models. First, we investigate *model compression*, comparing the already evaluated baseline fp32 with bfloat16, int8, and int4 quantization levels. Such techniques are increasingly adopted to reduce memory footprint and accelerate inference (Wei et al. 2023). While prior work has highlighted their efficiency benefits and potential security implications (Egashira et al. 2024), their effect on memorization in pre-trained and fine-tuned code models remains unexplored. Second, we examine *parameter-efficient fine-tuning*, focusing on LoRA (Hu et al. 2022a). LoRA enables adapting large models by training only a small number of additional parameters, making it a popular alternative to full fine-tuning. However, its implications for memorization in code models remain largely unexplored. By analyzing both compression and parameter-efficient fine-tuning, we widen existing research on memorization in code models and contribute new insights into how practical deployment choices shape extractability.

Figure 1 provides an overview of the configuration of the experiments. We adopt a setup introduced by Carlini et al. (Carlini et al. 2022c) to conduct targeted data extraction attacks. Targeted attacks are more security-critical as they allow for targeting specific information from the training corpus. We perform two types of targeted attacks: a *pre-training code attack* to evaluate the extractability of pre-training data, and a *fine-tuning code attack* to quantify the vulnerability of recent data encountered by the model.

3.1 StarCoder2 and TheStack v2

For this study, we utilized StarCoder2 (Lozhkov et al. 2024). StarCoder2 is a transparently trained open code LLM and comes in three sizes: 3B, 7B, and 15B parameters. All models use Grouped Query Attention, a context window of 16384 tokens with sliding window attention of 4096 tokens, and were trained using the Fill-in-the-Middle objective.

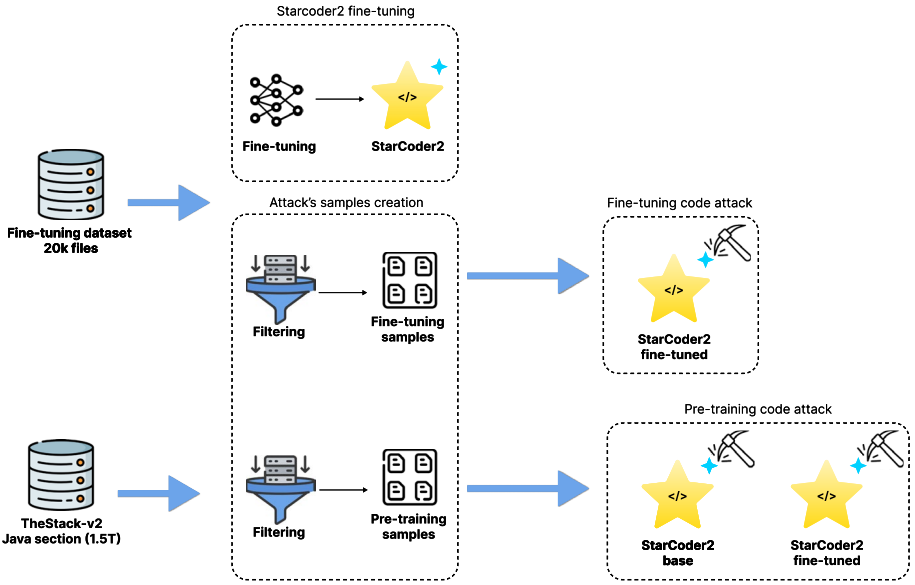


Fig. 1 Overview of the experiments' configuration

StarCoder2-15B was trained on 600+ programming languages from The Stack v2 on 4+ trillion tokens (Lozhkov et al. 2024).

The Stack v2 is the largest open code dataset suitable for LLM pre-training; it builds on Software Heritage's extensive source code archive, which covers over 600 programming languages. In addition to code repositories, The Stack v2 includes curated data from high-quality open sources such as GitHub issues, pull requests, Kaggle and Jupyter notebooks, code documentation, and other natural language datasets related to mathematics, coding, and reasoning. The complete dataset comprises approximately 67.5 TB and around 900 billion tokens.[2] We chose StarCoder2 as it is one of the most recent open-source code models. Starcoder2 and its derivatives are the best-performing models with both open weights and open data on the EvalPlus Leaderboard (Liu et al. 2023).[3] Its open-data nature provides us with complete access to the training corpus, which is essential for conducting targeted attacks and assessing the extractability of pre-training data (Yang et al. 2024; Al-Kaswan et al. 2024).

3.2 Mellum

To assess the generalizability of our findings beyond a single model family, we extend our analysis to Mellum-4B (Pavlichenko et al. 2025), a code language model developed by JetBrains. Mellum adopts a LLaMA-style architecture with 4 billion parameters, a context window of 8192 tokens, and was trained using Automatic Mixed Precision (AMP) in bfloat16 precision (Pavlichenko et al. 2025). Unlike StarCoder2, which uses Grouped

[2] TheStack v2: https://huggingface.co/datasets/bigcode/the-stack-v2

[3] EvalPlus Leaderboard: https://evalplus.github.io/leaderboard.html

Query Attention and a Fill-in-the-Middle training objective, Mellum follows a decoder-only autoregressive design and is specifically optimized for code completion in IDE environments. Mellum was pre-trained on approximately 4 trillion tokens of multi-language code sourced exclusively from permissively licensed repositories. Its training corpus draws from The Stack (Allal et al. 2023) and The Stack v2 (Lozhkov et al. 2024), among other curated sources. This partial overlap with the StarCoder2 pre-training data enables us to apply the same fine-tuning attack framework, as both models have not seen the fine-tuning dataset from The Heap (Katzy et al. 2025) during pre-training.

Mellum was pre-trained on approximately 4 trillion tokens of multi-language code sourced exclusively from permissively licensed repositories. Its training corpus draws from The Stack (Allal et al. 2023) and The Stack v2 (Lozhkov et al. 2024), among other curated sources. This partial overlap with the StarCoder2 pre-training data enables us to apply the same fine-tuning attack framework, as both models have not seen the fine-tuning dataset from The Heap (Katzy et al. 2025) during pre-training.

3.3 Research Questions

RQ1: **To what extent are pre-training data extractable from pre-trained models?** We conduct a data extraction attack on the base version of StarCoder2 to evaluate the extractability of its training data. We examine two factors that might influence extractability: model size and input prefix length (Al-Kaswan et al. 2024; Carlini et al. 2021).

RQ2: **To what extent are pre-training data extractable from fine-tuned models?** After fine-tuning the model on the Java line completion task, we repeat the pre-training code attack to compare the extractability before and after fine-tuning.

RQ3: **To what extent are fine-tuning data extractable from fine-tuned models?** We conduct a data extraction attack on the fine-tuned StarCoder2 to evaluate the extractability of recent fine-tuning data. Factors analyzed include model size, training epochs, duplication rate of samples, and input prefix length variations (Al-Kaswan et al. 2024; Carlini et al. 2021).

RQ4: **Does the type of code data impact extractability?** We investigate whether certain data categories are more or less extractable after fine-tuning. We analyze samples from both fine-tuning and pre-training data extraction attacks, constructing a classification to evaluate their extractability.

RQ5: **To what extent do fine-tuning memorization and forgetting patterns generalize across model architectures?** We fine-tune Mellum-4B (Pavlichenko et al. 2025) on the same Java line completion task and repeat the pre-training and fine-tuning code attacks to investigate the memorization patterns of a different architecture under the same experimental conditions.

RQ6: **To what extent does model compression affect data extractability?** We evaluate StarCoder2 models under standard compression techniques (bfloat16, int8, and int4), using the default configuration (full-precision fp32) as the baseline. We repeat the pre-training and fine-tuning code attacks to investigate the impact of these techniques on memorization.

RQ7: **To what extent do PEFT techniques affect the extractability of pre-training and fine-tuning data compared to full fine-tuning?** We fine-tune the StarCoder2 models using prominent PEFT methods (Hu et al. 2022a; Zhuo et al. 2024) on the Java line completion task and repeat the pre-training and fine-tuning code attacks to evaluate changes in

memorization rates.

4 Experimental Setup

4.1 Fine-Tuning Dataset

The fine-tuning dataset consists of 20k `.java` files randomly sampled from the Java subset of The Heap (Katzy et al. 2025).[4] This dataset was created by scraping 10.5k public repositories on GitHub. The Heap consists of repositories released under strong copyleft licenses, as these licenses are excluded from the StarCoder2 training corpus (Lozhkov et al. 2024; Katzy et al. 2024, 2025). The scraped files are near-deduplicated against The Stack to remove any possibility of training-data contamination (Katzy et al. 2024, 2025).

We further apply several filters to the dataset in-line with the StarCoder2 pre-processing pipeline (Lozhkov et al. 2024). We remove files that are too long, have too few alphabetic characters, and those consisting of encoded or autogenerated data (Lozhkov et al. 2024). All the filter's implementations can be found inside the replication package (Salerno et al. 2025). Figure 2 schematizes the fine-tuning dataset construction process.

[4] The Heap Dataset: https://huggingface.co/datasets/AISE-TUDelft/the-heap

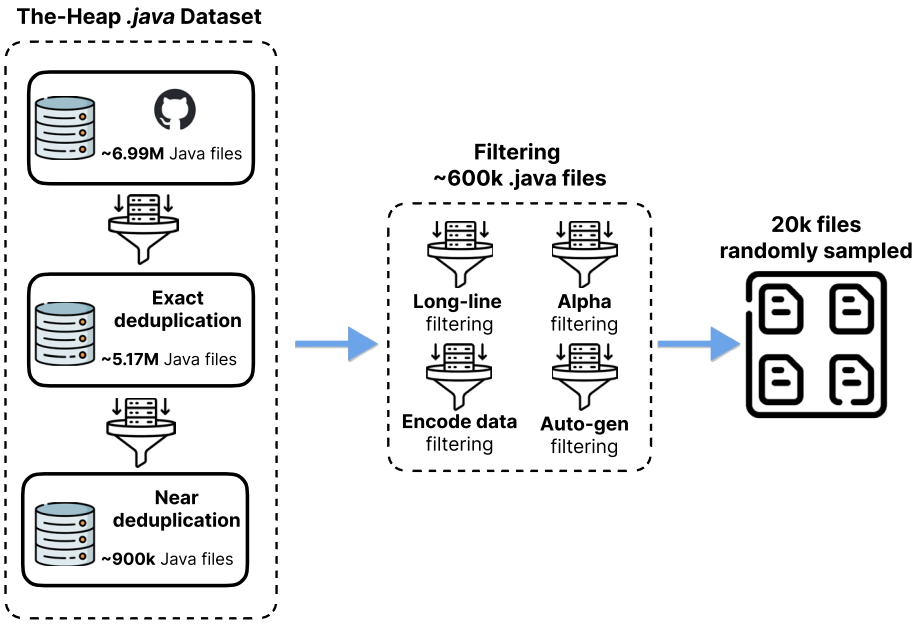


Fig. 2 Fine-tuning dataset construction process

4.2 Fine-Tuning

The fine-tuning of each model was conducted using the resources provided by the Delft High-Performance Compute Cluster (DHPC 2024). We utilize 32 CPU cores with 32GB of RAM and a set of Nvidia A100 GPUs, each with 80GB of memory. The experiments are performed on various sizes of the StarCoder2 model, with the specific number of GPUs allocated based on the model's parameter count to fit within memory constraints. Specifically, for StarCoder2-3B, 7B, and 15B, we employ two, four, and six GPUs, respectively. This results in approximate training times of 25, 55, and 110 hours, respectively. The GPUs are running Nvidia driver version 555.42.02 and CUDA 12.5. The training utilizes Transformer version 4.41.1 on Torch 2.3.0+cu121.

To ensure consistent comparison across the different sizes of StarCoder2, we maintain the same training setup for each model size. The effective training batch size (including gradient accumulation) is 24, 24, and 25, respectively. The context window is set to 1024 tokens. We use a learning rate of $3e-5$ and utilize a linear scheduler and the Adafactor optimizer (Shazeer and Stern 2018) to reduce GPU memory requirement and to make reproducibility easier. Each model was trained for three epochs, with checkpoints saved at the end of each epoch. To avoid overfitting, we monitored the validation loss after each fine-tuning epoch to ensure it did not increase. This approach confirms that the models are not overfitting.[5] For Mellum-4B, we follow the same fine-tuning procedure described above to ensure a fair comparison across architectures. The training is conducted on a single Nvidia A100 GPU (80GB), with a per-device batch size of 2 and 6 gradient accumulation steps, resulting in an effective batch size of 12.

Detailed training configurations and results are available in the replication package (Salerno et al. 2025).

4.3 Model Compression

To study the impact of model compression techniques on data extractability (RQ5), we evaluated three precision settings on a single model size, StarCoder2-15B: bfloat16, int8 quantization, and int4 quantization. These settings were applied at inference-time model loading. We selected these methods as they represent the standard precision-reduction techniques supported across modern deep learning frameworks and hardware.

In the bfloat16 setting, weights and activations were loaded in reduced precision by specifying the dtype during Torch model initialization, which provides a smaller memory footprint while preserving a dynamic range comparable to fp32. For quantization, we rely on the Hugging Face bitsandbytes library. We use the standard post-training quantization configurations. In this setup, model weights are quantized in a block-wise manner with a group size of 128. For the 8-bit setting, *load_in_8bit=True* is enabled, which stores weights in int8 while keeping key computations in higher precision (*fp16*/*bf16*) to maintain stability. For the 4-bit setting, *load_in_4bit=True* is enabled with the default NormalFloat (*nf4*) quantization type (Dettmers et al. 2023), which is designed for weight distributions and provides better accuracy than uniform 4-bit quantization.

[5] The loss graphs are available in the replication package (Salerno et al. 2025): training/train-stats

4.4 Parameter Efficient Fine-Tuning

To investigate the impact of Parameter Efficient Fine-Tuning (PEFT) techniques on data extractability (RQ6), we fine-tune the StarCoder2 models using Low-Rank Adaptation (LoRA) (Hu et al. 2022a; Zhuo et al. 2024). We select LoRA due to its demonstrated effectiveness in code-related tasks, and because it offers a strong trade-off between accuracy and efficiency (Zhuo et al. 2024). The fine-tuning was performed on the same 20k Java dataset from The Heap as in full fine-tuning, with identical preprocessing filters to ensure consistency.

We use the Hugging Face PEFT library integrated with Transformers. We used a single model size, StarCoder-7B to reduce the computational cost of our experiments. We use de default LoRA configuration with rank $r = 8$, scaling factor $\alpha = 16$, dropout of 0.05, targeting projection modules in the attention and MLP layers. The detailed training configurations and the loss graphs are available in the replication package (Salerno et al. 2025).

5 Data Extraction Attacks

We first introduce a precise definition of memorization proposed by Carlini et al. (Carlini et al. 2022b).

Definition 1 (memorization) A string s is *extractable with k tokens of context* from a model f if there exists a (k-length) string p, such that the concatenation $[p||s]$ is contained in the training data for f, and f produces s when prompted with p using greedy decoding.

This is defined as verbatim memorization. Ippolito et al. proposed a relaxed definition of memorization using the Bilingual Evaluation Understudy (BLEU) score (Ippolito et al. 2023). To measure memorization, we first formally define the data extraction game and then describe the approach used to obtain the pre-train and fine-tuning code samples for performing the data extraction attacks.

5.1 Data Extraction Security Game

In our experiments, we consider the models as black-box systems. We define a security game inspired by the membership inference attack security game and the notion of k-extractability (Carlini et al. 2021, 2022b; Al-Kaswan et al. 2024).

Definition 2 (Data Extraction Security Game) Given a challenger $\mathcal{C}$, an adversary $\mathcal{A}$, a data distribution $\mathbb{D}$ and a model f the game is defined as follows:

1. The challenger samples a training dataset $D \leftarrow \mathbb{D}$ and trains a model $f_\theta \leftarrow \mathcal{T}(D)$ on the dataset D.
2. $\mathcal{C}$ samples a sample $D_n = (p, s)$ where $D_n \in D$. The prefix p is provided to the adversary $\mathcal{A}$.
3. $\mathcal{A}$ is allowed query access to the model f_θ and may perform any other polynomial-time operations

4. $\mathcal{A}$ outputs his prediction sequence $\hat{s}$
5. If $\hat{s} = s$, $\mathcal{A}$ wins, otherwise $\mathcal{C}$ wins

In other words, given a prefix p (1), the adversary is challenged to extract the correct suffix from the model's training data. The adversary can query the language model f_θ(2) but cannot inspect the weights or hidden states (e.g., attention vectors). The adversary then predicts the suffix s (3) and wins if it matches the actual suffix in the training data (Carlini et al. 2021, 2022b; Al-Kaswan et al. 2024).

Some factors may affect the difficulty of the challenge:

1. The selection of the sample $D_n \in D$. As noticed by previous works, not all training samples are as hard to extract as others. In particular, highly duplicated or outlier samples are more vulnerable to attacks (Kandpal et al. 2022; Lee et al. 2022; Carlini et al. 2022b).
2. The size of the model f_θ. Some models are more likely to memorize samples than others, i.e., larger models have been observed to memorize more pre-training data than smaller models (Carlini et al. 2022b; Al-Kaswan et al. 2024; Yang et al. 2024).
3. The length of the prefix p. It has been found that longer prefixes induce more memorization. Carlini et al. (2022b); Yang et al. (2024).
4. The victory condition $\hat{s} = s$. Using the exact match might exclude samples that are entirely extractable but have meaningless differences e.g., a space character. So, a fuzzy match can also be considered (Ippolito et al. 2023).

In this work, we exploit the security game and the factors (1,3,4) described above to construct a set of extractable samples to evaluate different model sizes (2). We also measure fuzzy match scores (4) and compare them with the exact match rate.

We note that our evaluation operationalizes memorization through targeted extractability in a black-box security game. This choice does not aim to cover all possible forms of memorization. Models may memorize training content approximately or in forms that are not recoverable through greedy exact-match extraction from a fixed prefix. Accordingly, extractability should be interpreted as a lower-bound proxy for memorization; successful extraction provides direct evidence that training data has been retained in a recoverable form, whereas failed extraction does not imply the absence of memorization. We adopt this setup because it is well grounded in prior work (Carlini et al. 2021, 2022b; Al-Kaswan et al. 2024; Yang et al. 2024) and because it reflects a realistic deployment scenario where an adversary interacts with a model through black-box API access, without visibility into model parameters or internal states.

5.2 Data Extraction Dataset

To measure memorization in LLMs for code, we first need to construct a dataset analogous to the one used in the SaTML23 Language Model Data Extraction Challenge. [6] We can divide the extractions into *pre-training code attack* (extracting pre-training data from the model) and *fine-tuning code attack* (extracting fine-tuning data from the model). In our experiments, we use different sizes of StarCoder2 to improve generalizability.

[6] Language Models Training Data Extraction Challenge: https://github.com/google-research/lm-extraction-benchmark

We collect the pre-training samples for the attack from the Java subset of TheStack-v2 through the following procedure. The candidate samples for the attack are obtained by randomly selecting 300 token spans from anywhere in the file taking into account the context window used to train StarCoder2 (Lozhkov et al. 2024). All the files with less than 300 tokens were discarded. We considered only candidates with more than three duplicates inside the training corpus to be consistent among the RQs given the space limit. Additionally, to avoid overlaps we exclude the candidate samples also included in the fine-tuning set. We randomly select 1K samples from the filtered candidate set to perform our evaluation. Finally, we split the 300 tokens sequences into a suffix of 50 tokens and a prefix of 250 tokens. In our experimental setup, we will truncate the prefix to 100, 150, and 200 tokens to test extractability when feeding the inputs with different prefix lengths to the models. As in Al-Kaswan et al. (Al-Kaswan et al. 2024), our pre-train attack dataset construction procedure differs from the one used by Carlini et al. (Carlini et al. 2022b) in one aspect. Our dataset does not guarantee that for every $D_n = (s, p)$ there does not exist a $(s, p') \in D$ where $p' \neq p$. This is because the computational cost of identifying all unique samples $D_n = (s, p)$ is extremely large for a dataset of this size (Al-Kaswan et al. 2024).

The approach for collecting fine-tuning samples for the attack presents variations from the previously described procedure. Given the smaller scale of the fine-tuning set compared to the pre-training set, we implemented certain modifications that enabled further experimentation. We apply the candidate samples for the attack by applying a 300 tokens sliding window to each file of the fine-tuning set while considering the context window used in the fine-tuning setup. The size of the fine-tuning set allows us to know and control the number of times a 300-token sample is in the fine-tuning set; this enables us to control the duplication rate. We divided candidate samples into subsets based on their duplication rate: unique samples ($d = 1$), single duplicates ($d = 2$), double duplicates ($d = 3$), and samples with more than three duplicates ($d > 3$). This choice is guided by inspecting the duplication rate sample distribution on the fine-tuning set, as shown in Fig. 3.

For each duplication rate subset, we randomly select 1K samples from the filtered candidate set to perform our evaluation. Finally, we split the 300 token sequences into a suffix of 50 tokens and a prefix of 250 tokens with the same approach we used for the pre-training attack dataset. The smaller scale of the fine-tuning set compared to the pre-training set allowed us to guarantee that for every $D_n = (s, p)$, there does not exist a $(s, p') \in D$ where $p \neq p'$ as in the attack dataset construction procedure by Carlini et al. (Carlini et al. 2022b).

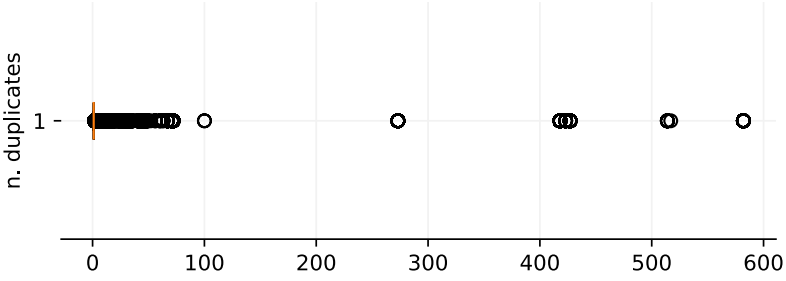


Fig. 3 Duplication rates distribution for 300-token samples (fine-tuning data)

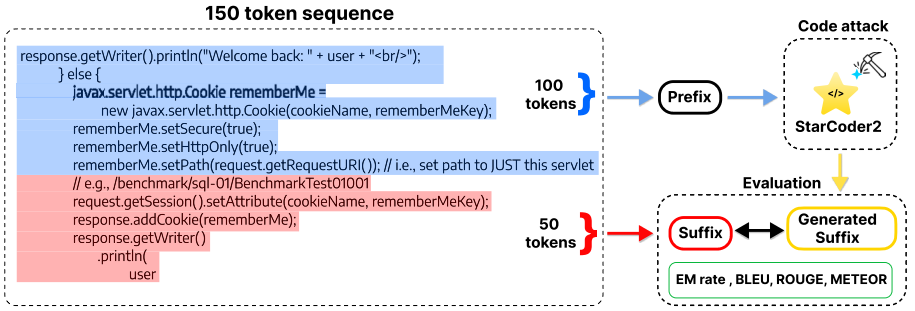


Fig. 4 The code data extraction attack

Table 1 Categories of data extraction datasets

Category	Purpose	Pre-training	Fine-tuning
Code	Code Logic	388	223
Dicts	Dictionaries or other data carriers	164	117
Docs	Documentation	173	291
License	License information	124	284
Testing	Test Code	50	2
Total		899	917

5.3 Performing the Attacks

We prompt the model under investigation with the prefix. We utilize the standard generation pipeline and the default generation configuration of StarCoder2, as defined in the model configuration. Figure 4 schematizes the data extraction attack. The models are prompted in a one-shot fashion with greedy decoding. The generated suffix is then compared with the true suffix, which serves as the ground truth. We evaluate the performance using two types of metrics. First, we measure the Exact Match (EM) score. Additionally, we also assess the match by fuzzier metrics such as BLEU (Post 2018), METEOR (Banerjee and Lavie 2005), and ROUGE-L (Lin 2004). To quantify the uncertainty of our measurements, we report 95% confidence intervals for all reported metrics. For the Exact Match (EM) rate, we compute Wilson score intervals, suitable for binomial proportions. For the continuous metrics (BLEU, METEOR, and ROUGE-L), we compute bootstrap percentile confidence intervals ($n_resamples = 10{,}000$, percentiles 2.5% and 97.5%). All confidence intervals are reported alongside point estimates in the results tables.

6 Analysis through Categorization

To better understand the type of extracted data we conduct an exploratory study by categorizing both the pre-trained and fine-tuned attack datasets. For consistency with the framework in the literature, we decided to keep and use the code categories defined by Al-Kaswan et al. (Al-Kaswan et al. 2024). The different categories are outlined in Table 1, while representative code snippets for each category are shown in Fig. 5. For fine-tuning, we selected

Code Logic

```
  class InHeadNoscript {
    boolean process(Token t, HtmlTreeBuilder tb) {
      if (t.isDoctype()) {
        tb.error(this);
      } else if (t.isStartTag() && t.asStartTag().name().equals("html")) {
        return tb.process(t, InBody);
      }
    }
  }
```

Dictionaries or other data carriers

```
  {
    "abc_tooltip_enter": "0×7f01000a",
    "abc_tooltip_exit": "0×7f01000b",
    "btn_checkbox_to_checked_box_inner_merged_animation": "0×7f01000c",
    "btn_checkbox_to_checked_box_outer_merged_animation": "0×7f01000d",
    "btn_checkbox_to_checked_icon_null_animation": "0×7f01000e"
  }
```

Documentation

```
  /**
   * FAILURE if there is a failure to allocate memory for buffer object.
   * CL_OUT_OF_RESOURCES if resources cannot be allocated on the device.
   * CL_OUT_OF_HOST_MEMORY if resources cannot be allocated on the host.
   *
   * @return cl_mem
   */
  @NativeType("cl_mem")
```

License Information

```
  Derivative Works; within the Source form or
  documentation, if provided along with the Derivative Works; or,
  within a display generated by the Derivative Works, if and
  wherever such third-party notices normally appear.
  The contents of the NOTICE file are for informational purposes only
  and do not modify the License.
```

Test Code

```
  @Test
  public void shouldExtractPrincipalCorrectly() {
      assertThat(events).extracting("principal")
          .containsExactlyInAnyOrder("test-user", "other-test-user");
  }
```

Fig. 5 Representative examples of the extracted code categories (Al-Kaswan et al. 2024)

one of the four 1k code attack datasets, specifically choosing the sample set with a duplication rate of 3. This choice represents a balance between duplication rate and category representativeness.

To simplify the classification process, samples with dual purposes (e.g., samples that start with license information and finish with code.) were assigned to their predominant category (Al-Kaswan et al. 2024).

Two authors independently conducted the classification and subsequently compared their results. Before labeling, both reviewers jointly examine 20 samples to familiarize themselves with the dataset. The labeling process shows strong agreement between annotators, with Cohen's Kappa scores of 0.87 for the pre-training code attack dataset and 0.90 for the fine-tuning code attack dataset. Analysis of the confusion matrix reveals that discrepancies primarily arise from differences between test categories and code categories, as well as from the decisions made when categorizing samples with dual purposes. We discard the samples that the reviewers do not agree on, and we are left with 899 and 917 samples for the pre-training and fine-tuning datasets, respectively.

7 Results

7.1 Extracting Pre-training Data from Pre-Trained Models

The results of the pre-training code attack on the pre-trained models are shown in Table 2.

We observe that increasing the length of the input prefix in attacks enhances the extractability of pre-training data from pre-trained models. We are able to extract 45.5% of the samples from the StarCoder2-15B model using a 100-token prefix. This extraction rate increases to 50.6% with a 150-token prefix, and further to 54.9% with a 250-token prefix. This pattern is also supported by analyzing the results of fuzzy metrics. The suffixes generated by the StarCoder2-15B model reached a BLEU score of 65.2% with a 100-token prefix, which improved to 72.8% with a 250-token prefix. As shown in Fig. 6, the increase in extractability is notably more pronounced when extending the prefix length from 100 to 150 tokens compared to the increase from 200 to 250 tokens. This suggests a nonlinear growth trend concerning prefix length.

We observe that model size impacts pre-training extractability in pre-trained models. Table 2 indicates that using a 100-token prefix, we are able to extract 38.9% of the pre-training code samples from the StarCoder2-3B model. This extraction rate increases to 45.5% when using the largest version of the model, StarCoder2-15B.

> **RQ1:** The base version of StarCoder2 memorizes pre-training data, with memorization rate scaling with model size. The extractability of this data increases with longer input prefixes, though not linearly.

7.2 Extracting Pre-training Data From Fine-tuned Models

In Fig. 7, we plot the EM rates for the pre-training code attack on base and fine-tuned models. We notice a considerable negative impact of fine-tuning on the extractability of pre-

Table 2 Pre-training code attack on the pre-trained models

		Memorization rate			
Model	Prefix length	EM	BLEU	METEOR	ROUGE-L
Starcoder2-3B	100 tokens	0.389 [.359, .420]	0.652 [.630, .675]	0.707 [.687, .727]	0.682 [.662, .703]
	150 tokens	0.432 [.402, .463]	0.689 [.667, .710]	0.736 [.717, .755]	0.712 [.692, .733]
	200 tokens	0.462 [.431, .493]	0.715 [.694, .736]	0.756 [.737, .775]	0.737 [.717, .757]
	250 tokens	0.483 [.452, .514]	0.729 [.707, .749]	0.769 [.750, .787]	0.752 [.732, .771]
Starcoder2-7B	100 tokens	0.395 [.365, .426]	0.664 [.642, .685]	0.720 [.700, .740]	0.694 [.674, .715]
	150 tokens	0.452 [.421, .483]	0.701 [.689, .731]	0.757 [.738, .775]	0.734 [.713, .754]
	200 tokens	0.485 [.454, .516]	0.731 [.710, .751]	0.772 [.753, .790]	0.754 [.734, .773]
	250 tokens	0.498 [.467, .529]	0.742 [.722, .763]	0.782 [.763, .800]	0.764 [.745, .783]
Starcoder2-15B	100 tokens	0.455 [.424, .486]	0.715 [.694, .736]	0.759 [.741, .778]	0.739 [.719, .758]
	150 tokens	0.506 [.475, .537]	0.757 [.737, .777]	0.792 [.774, .809]	0.775 [.756, .794]
	200 tokens	0.536 [.505, .567]	0.777 [.757, .796]	0.806 [.788, .824]	0.794 [.776, .812]
	250 tokens	0.549 [.518, .580]	0.789 [.770, .808]	0.819 [.802, .836]	0.805 [.787, .823]

training data. Primarily, for the StarCoder2-15B model, we observe that after three epochs of fine-tuning, 30% of the samples that were previously extractable during the pre-training stage could no longer be extracted. This trend is consistent across different model sizes to the same extent.

Table 3 presents the results of experiments involving the pre-training code attack on fine-tuned models across the fine-tuning epochs, with the input prefix length fixed at 100 tokens. The results show that fine-tuning epochs do not significantly decrease extractability. For the StarCoder2-15B base model, we initially extract 45.5% of the pre-training code attack dataset. This rate decreases to 14.5% after a single epoch of fine-tuning, with subsequent epochs maintaining a similar EM rate, showing only minor fluctuations within the standard error. Fig. 8 visualizes this behavior.

For the StarCoder2-3B and 7B models, the extractability of pre-training data decreases after the first epoch but increases slightly in subsequent epochs. Specifically, the EM rate

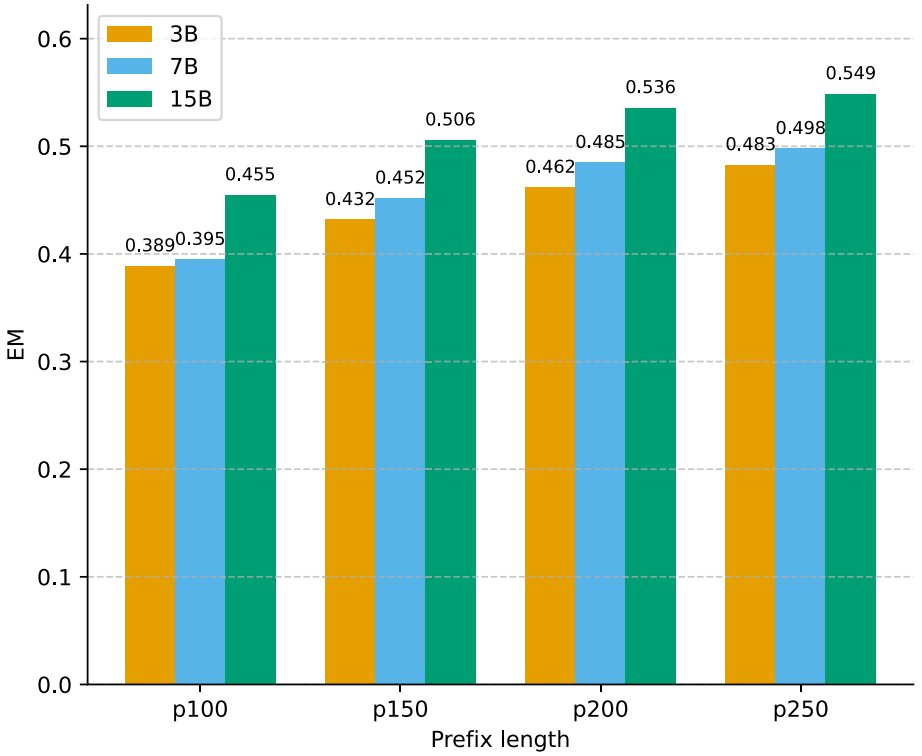


Fig. 6 EM rate when performing pre-training code attack on pre-trained models

for the StarCoder2-3B model drops to 8.7% after the first fine-tuning epoch, increases to 10.8% by the second epoch, and stabilizes at 10.5% after the third epoch. The StarCoder2-7B exhibits a more pronounced initial decline, with the EM rate decreasing to 1.5% after the first epoch, and then increasing to 11.0% and 11.6% after the second and third epochs, respectively.

Table 4 shows the results of the experiments involving the pre-training code attack on the fine-tuned models while varying the input prefix length of the attacks. In line with the findings of the first RQ, when attacking the pre-trained models, extending the input prefix allows us to extract more pre-training samples from the fine-tuned models. For instance, with a 100-token input attack on StarCoder2-15B, we extract 15.5% of the pre-training code attack samples. This extraction rate increases to 23.5% when extending the input prefix to 250 tokens. Comparing the pre-training code attack on pre-trained models (Table 2) with that on fine-tuned models (Table 4), we observe that the increase in extractability from extending the input prefix is less pronounced in fine-tuned models compared to pre-trained models, as also visualized in Fig. 9. For instance, extending the input prefix from 100 to 150 tokens allowed the extraction of 43 additional samples from the StarCoder2-3B base model. In contrast, the same prefix length extension on the fine-tuned version resulted in the extraction of only 19 additional samples.

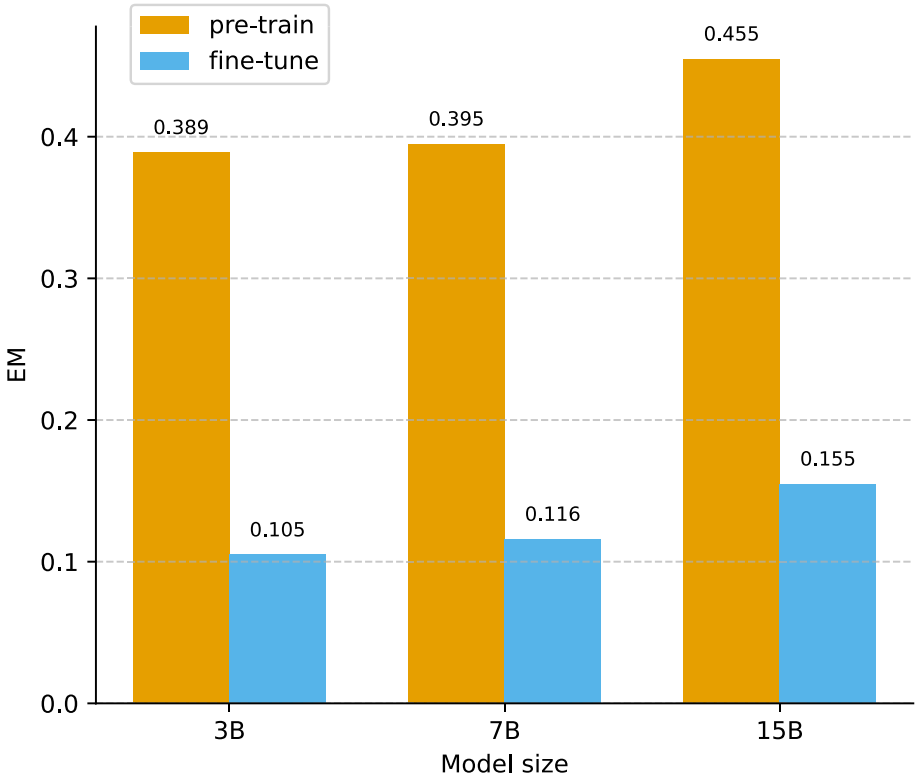


Fig. 7 EM rate when performing the pre-training code attack (with 100 token prefix) on base and fine-tuned models

RQ2: Fine-tuning reduces the extractability of pre-training data. This effect is consistent across model sizes and unaffected by the number of training epochs. Extending the input prefix length increases extractability but with a lower magnitude compared to pre-trained models.

7.3 Extracting Fine-tuning Data

In Table 5, we present the results of experiments involving the fine-tuning code attack across the fine-tuning epochs. Notably, performing the fine-tuning code attack on the pre-trained models revealed that some fine-tuning samples were also extractable from the pre-trained models. In this section, we demonstrate the increase in EM and BLEU (↑%) relative to the baseline at epoch zero. Due to space limits, METEOR and ROUGE-L scores are available in the replication package (Salerno et al. 2025). The baseline represents the residual memorization rate achieved by executing the fine-tuning code attack on the pre-trained models. Our experiments reveal that extractability increases with the number of fine-tuning epochs. Specifically, after one epoch of training StarCoder2-3B, we were able to extract 45.9% of

Table 3 Pre-training code attack on fine-tuned models across several fine-tuning epochs

		Memorization rate			
Model	Epochs	EM	BLEU	METEOR	ROUGE-L
Starcoder2-3B	0	0.389 [.359, .420]	0.652 [.630, .675]	0.706 [.687, .727]	0.682 [.662, .703]
	1	0.087 [.071, .106]	0.449 [.429, .469]	0.551 [.532, .569]	0.517 [.497, .536]
	2	0.108 [.090, .129]	0.452 [.431, .472]	0.552 [.533, .571]	0.518 [.499, .538]
	3	0.105 [.087, .126]	0.448 [.428, .468]	0.551 [.533, .570]	0.514 [.495, .533]
Starcoder2-7B	0	0.395 [.365, .426]	0.663 [.642, .685]	0.712 [.700, .740]	0.694 [.674, .715]
	1	0.015 [.009, .025]	0.381 [.364, .399]	0.524 [.507, .542]	0.483 [.465, .502]
	2	0.110 [.092, .131]	0.461 [.441, .481]	0.566 [.548, .585]	0.527 [.508, .547]
	3	0.116 [.098, .137]	0.470 [.450, .490]	0.572 [.554, .590]	0.534 [.514, .553]
Starcoder2-15B	0	0.455 [.424, .486]	0.714 [.694, .736]	0.760 [.741, .778]	0.738 [.719, .758]
	1	0.145 [.125, .168]	0.526 [.505, .546]	0.613 [.595, .632]	0.583 [.563, .603]
	2	0.143 [.123, .166]	0.531 [.510, .551]	0.619 [.600, .637]	0.586 [.566, .605]
	3	0.155 [.134, .179]	0.545 [.524, .565]	0.629 [.611, .648]	0.596 [.577, .615]

the fine-tuning samples. This extraction rate increased to 59.5% after the second epoch and further to 63.1% after the third epoch.

In Fig. 10, we present the increase of the EM rate over the baseline across epochs. Our experiments reveal that smaller models are more prone to extraction on fine-tuning data than bigger ones. Based on Fig. 10, we can see that as the number of fine-tuning epochs increases, smaller models exhibit greater vulnerability to data extractability, whereas larger models demonstrate reduced exposure. Specifically, when attacking the StarCoder2-3B model after three epochs of fine-tuning, we extracted 36.8% additional samples relative to the baseline. In contrast, attacking the StarCoder2-15B model under the same conditions yielded 30.9% additional extractable samples compared to the baseline.

Table 6 shows the results of the experiments consisting of attacking the fine-tuned models, controlling the duplication rate (i.e., the frequency of 300-token samples in the fine-tuning corpus). We find that the duplication rate has a notable impact on extractability.

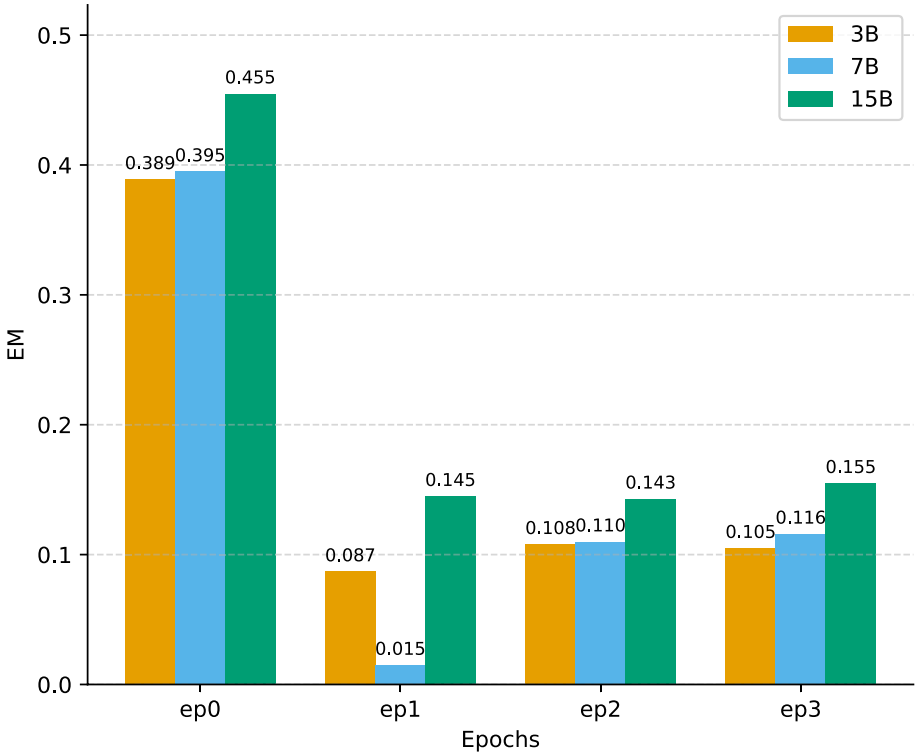


Fig. 8 EM when performing the pre-training code attack (with 100 token prefix) across fine-tuning epochs

Increasing duplication consistently raises the EM rate across model sizes, as shown in Fig. 11. For StarCoder2-3B, extractability is below 5% with unique samples. This percentage increases to 23.7% with a single duplicate in the fine-tuning corpus and further to 45.1% with two duplicates.

Table 7 describes the effect of extending the input prefix length on the success of fine-tuned code extraction attacks. Our results indicate that the extractability of fine-tuning data increases with the input prefix length. We can extract 63.1% of the samples from StarCoder2-3B inputting 100 prefix tokens. This percentage increased to 68.9% with a prefix of 150 tokens and further to 81.5% with a prefix of 250 tokens. Figure 12 illustrates this trend across all model sizes, showing a consistent rise in extractability with longer prefixes and only minor differences between models of different sizes.

To complement the extractability analysis, we evaluate the functional performance of the fine-tuned models using the pass@1 metric on Java code completion. We employ the HumanEval-X benchmark (Zheng et al. 2023), a multilingual extension of HumanEval (Chen et al. 2021) that extends the original Python-only problems with hand-written solutions in multiple languages, including Java. HumanEval-X enables a standardized evaluation of code generation capabilities across languages, making it suitable for assessing the effectiveness of our fine-tuning on Java. Figure 13 presents the pass@1 scores for the base and fine-tuned versions of StarCoder2. Across all model sizes, fine-tuning consistently improves pass@1. Specifically, StarCoder2-3B improves from 12.2% to 18.9%, StarCoder2-7B from 13.4% to

Table 4 Pre-training code attack on fine-tuned models varying the input prefix length

		Memorization rate			
Model	Prefix length	EM	BLEU	METEOR	ROUGE-L
Starcoder2-3B	100 tokens	0.105 [.087, .126]	0.447 [.428, .468]	0.551 [.532, .570]	0.514 [.495, .533]
	150 tokens	0.124 [.105, .146]	0.503 [.483, .524]	0.598 [.580, .616]	0.566 [.547, .586]
	200 tokens	0.151 [.130, .175]	0.526 [.505, .547]	0.613 [.594, .632]	0.583 [.564, .603]
	250 tokens	0.150 [.129, .173]	0.534 [.513, .555]	0.619 [.600, .637]	0.593 [.573, .612]
Starcoder2-7B	100 tokens	0.116 [.098, .137]	0.470 [.450, .489]	0.572 [.554, .591]	0.534 [.515, .553]
	150 tokens	0.150 [.129, .173]	0.524 [.504, .545]	0.616 [.597, .634]	0.582 [.562, .602]
	200 tokens	0.171 [.149, .196]	0.546 [.525, .567]	0.631 [.612, .650]	0.599 [.580, .619]
	250 tokens	0.178 [.156, .203]	0.560 [.539, .581]	0.642 [.624, .641]	0.613 [.593, .632]
Starcoder2-15B	100 tokens	0.155 [.134, .179]	0.545 [.525, .565]	0.629 [.611, .648]	0.596 [.577, .615]
	150 tokens	0.198 [.174, .224]	0.577 [.557, .598]	0.657 [.639, .675]	0.624 [.605, .644]
	200 tokens	0.220 [.195, .247]	0.599 [.578, .619]	0.675 [.657, .694]	0.646 [.626, .645]
	250 tokens	0.235 [.210, .262]	0.618 [.598, .639]	0.687 [.669, .706]	0.660 [.641, .679]

17.7%, and StarCoder2-15B from 23.8% to 27.4%. The largest model achieves the highest absolute pass@1 score, while the smallest model exhibits the largest relative improvement. The model that exhibits the largest relative improvement in pass@1 (StarCoder2-3B) is also the one showing the highest increase in fine-tuning data extractability after three epochs (Fig. 10).

> **RQ3:** Smaller models are more prone to extraction of fine-tuning data. Duplication rate significantly affects extractability. Increasing fine-tuning epochs leads to more samples memorized, and extending the input prefix length also facilitates fine-tuning extractability.

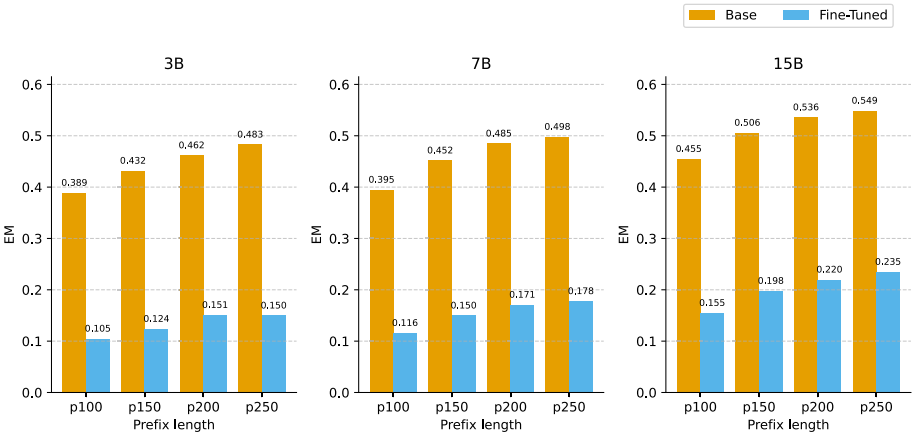


Fig. 9 EM rate when performing the pre-training code attack on base and fine-tuned models varying the input prefix length

7.4 Type of Extracted Data

Table 8 represents the type of code data extracted through our attacks. During the tagging process, we find numerous examples of names, emails, and GitHub links memorized by the model (Salerno et al. 2025).

Figure 14 shows the percentage per category of memorized samples performing the pre-training code attack on the base and fine-tuned models, respectively. The results indicate that *data carriers* and *licenses* are the most frequently memorized pre-training samples. Fine-tuning has an impact on the extractability of pre-training data. We find that this rate is not uniform between categories. Some categories are less extractable than others. Specifically, licenses become less extractable after fine-tuning compared to data carriers. For instance, the StarCoder2-3B base model memorizes 75.61% of the data carriers in the pre-training code attack dataset; this percentage decreases to 18.90% following fine-tuning. Similarly, the base model retains 68.55% of the licenses, but this figure drops to 6.45% after fine-tuning. Data carriers exhibit the highest memorization rate post-fine-tuning, being the category of code less affected by the "forgetting" phenomenon observed when performing the pre-training code attack on fine-tuned models.

The last column of Table 8 and Fig. 15 present, respectively, the number and percentage of memorized samples per category from the fine-tuning code attack. The results indicate that, similar to pre-training code attacks on pre-trained models, data carriers and licenses are the most memorized samples after fine-tuning. Specifically, the fine-tuned StarCoder2-15B model retained 54.70% of the data carriers and 68.66% of licenses from the fine-tuning code attack dataset.

> **RQ4:** Data carriers and license information are the most likely to be memorized, while the latter is the most likely to be forgotten after fine-tuning.

Table 5 Fine-tuning code attack on fine-tuned models across fine-tuning epochs

		Memorization rate			
Model	Epochs	EM	BLEU	EM ↑%	BLEU ↑%
Starcoder2-3B	0	0.263 [.237, .291]	0.538 [.514, .563]		
	1	0.459 [.428, .490]	0.739 [.716, .762]	0.196	0.168
	2	0.595 [.564, .625]	0.757 [.734, .780]	0.332	0.218
	3	0.631 [.601, .660]	0.764 [.741, .787]	0.368	0.226
Starcoder2-7B	0	0.249 [.223, .277]	0.546 [.522, .571]		
	1	0.408 [.378, .439]	0.739 [.716, .762]	0.159	0.158
	2	0.527 [.496, .558]	0.741 [.718, .764]	0.278	0.194
	3	0.578 [.547, .608]	0.749 [.726, .772]	0.329	0.202
Starcoder2-15B	0	0.300 [.272, .329]	0.576 [.551, .601]		
	1	0.549 [.518, .580]	0.756 [.733, .779]	0.249	0.181
	2	0.567 [.536, .597]	0.761 [.738, .784]	0.267	0.185
	3	0.609 [.578, .639]	0.774 [.752, .797]	0.309	0.198

7.5 Generalizability to Mellum-4B

To assess whether the memorization patterns observed in the StarCoder2 family generalize to a different architecture, we repeat the fine-tuning and pre-training code attacks on Mellum-4B.

Table 9 presents the results of the fine-tuning code attack on Mellum-4B across fine-tuning epochs. At epoch zero, the baseline extractability is 8.4%, indicating that a small portion of the fine-tuning samples can already be extracted from the pre-trained model. After one epoch, the EM rate increases to 9.5%, corresponding to an EM ↑% of 1.1 percentage points, though this increase is not statistically significant (CI $[-0.003, 0.025]$). Extractability becomes significant from the second epoch onward, reaching 11.9% (EM ↑% = 3.5) and 13.4% (EM ↑% = 5.0) after the third epoch. The fuzzy metrics confirm this trend: the BLEU

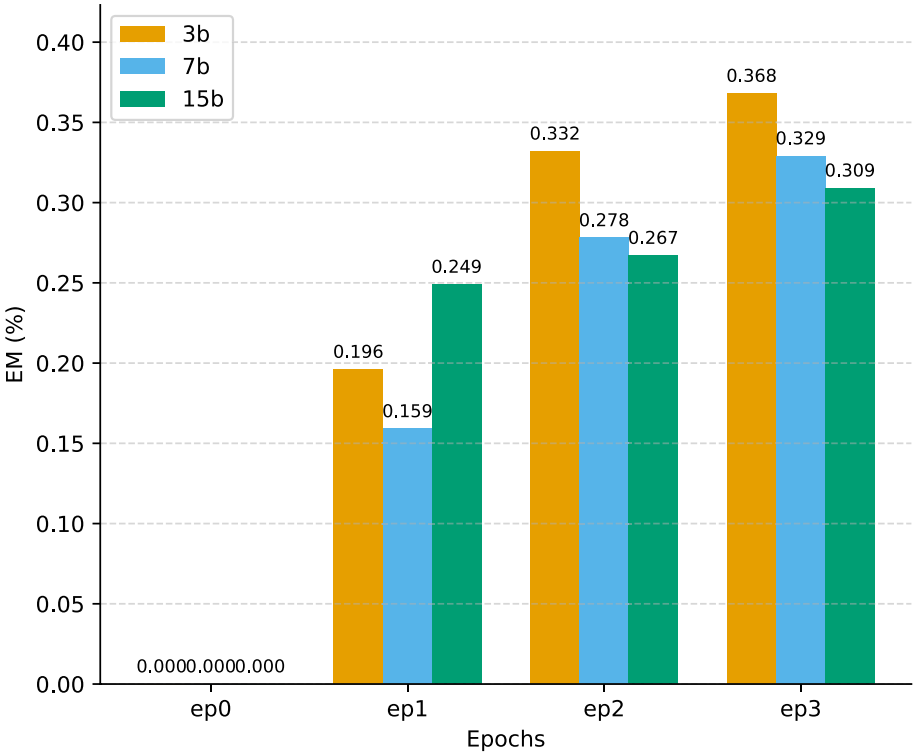


Fig. 10 EM ↑% rate when performing the fine-tuning code attack (with 100 token prefix) across fine-tuning epochs

score rises from 46.1% at epoch zero to 69.0% after three epochs, suggesting that even when exact matches are not achieved, the model generates sequences that closely resemble the fine-tuning data. Figure 16 illustrates this trend. Compared to StarCoder2-3B, which reaches an EM ↑% of 36.8 after three epochs, Mellum-4B exhibits substantially lower fine-tuning memorization under the same conditions.

Table 10 shows the results of the fine-tuning code attack on Mellum-4B with varying duplication rates. Consistent with the findings on StarCoder2, the duplication rate has a notable impact on extractability. With unique samples ($d = 1$), only 0.5% of the fine-tuning data can be extracted. This rate increases to 4.5% with a single duplicate ($d = 2$) and to 13.9% with two duplicates ($d = 3$). For samples with more than three duplicates ($d > 3$), extractability decreases slightly to 13.0%, though the confidence intervals overlap with $d = 3$ ([.111, .152] vs [.119, .162]), indicating that this difference is not statistically significant. Figure 17 illustrates this behavior. In StarCoder2-3B, the same trend is observed but at a much higher magnitude: extractability reaches 45.1% at $d = 3$ and 63.1% at $d > 3$.

Table 11 presents the effect of varying the input prefix length on the fine-tuning code attack. Extending the prefix from 100 to 200 tokens increases the EM rate from 13.4% to 19.9%. However, further extension to 250 tokens does not yield additional gains, with the EM rate at 20.4% and overlapping confidence intervals ([.175, .225] vs [.180, .230]). This plateau contrasts with StarCoder2-3B, where extractability continues to increase monotoni-

Table 6 Fine-tuning code attack on fine-tuned models with varying duplication rates

		Memorization rate			
Model	Duplication	EM	BLEU	EM↑%	BLEU↑%
Starcoder2-3B	=1	0.049 [.037, .064]	0.423 [.404, .441]	0.025	0.074
	=2	0.237 [.212, .264]	0.615 [.594, .636]	0.172	0.174
	=3	0.451 [.420, .482]	0.754 [.734, .774]	0.329	0.333
	>3	0.631 [.601, .660]	0.764 [.741, .788]	0.368	0.266
Starcoder2-7B	=1	0.061 [.048, .078]	0.433 [.414, .452]	0.042	0.082
	=2	0.238 [.213, .265]	0.616 [.595, .637]	0.181	0.174
	=3	0.450 [.419, .481]	0.776 [.758, .795]	0.348	0.337
	>3	0.578 [.547, .608]	0.749 [.726, .773]	0.329	0.202
Starcoder2-15B	=1	0.075 [.060, .093]	0.475 [.457, .494]	0.052	0.110
	=2	0.352 [.323, .382]	0.694 [.674, .715]	0.277	0.244
	=3	0.544 [.513, .575]	0.832 [.815, .849]	0.417	0.385
	>3	0.609 [.578, .639]	0.774 [.752, .798]	0.309	0.198

cally from 63.1% at 100 tokens to 81.5% at 250 tokens. The fuzzy metrics show a more consistent increase: the BLEU score rises from 69.0% at 100 tokens to 76.8% at 250 tokens. Figure 18 illustrates this trend.

Figure 19 presents the distribution of extractable fine-tuning samples per category for Mellum-4B. License information is the most frequently extracted category at 31.3%, followed by code at 10.8% and documentation at 4.8%. Notably, data carriers show a 0% extraction rate, in contrast with StarCoder2-3B where data carriers are among the most memorized categories (46.2%). Testing samples are also not extracted. While license information remains the most vulnerable category across both architectures, the absence of data carrier memorization in Mellum-4B represents a notable divergence from the patterns observed in StarCoder2.

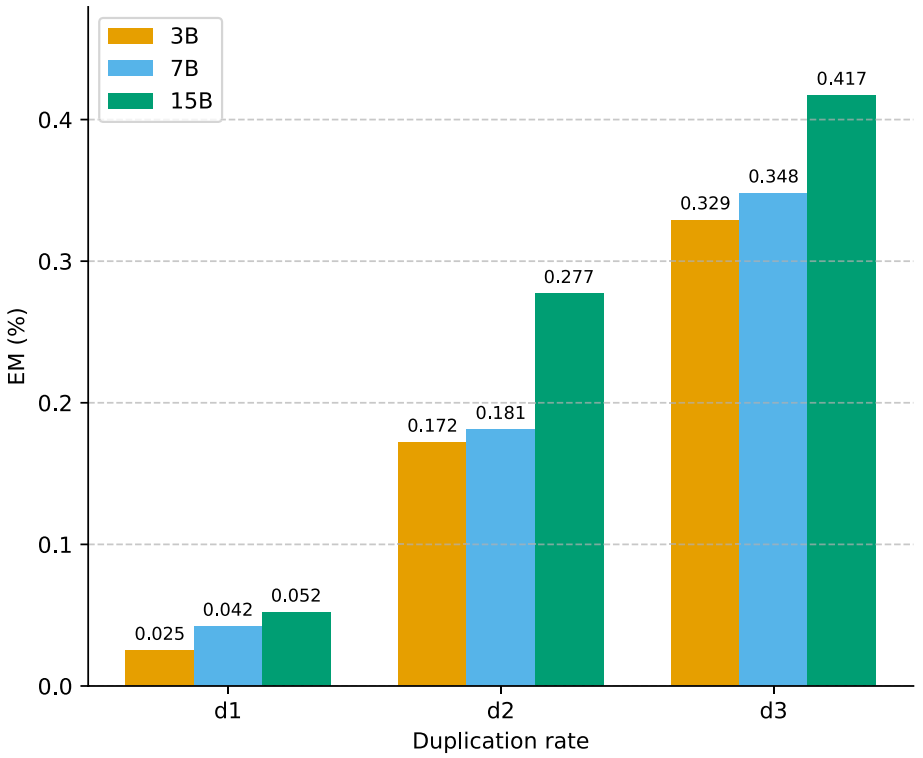


Fig. 11 EM ↑% rate when performing the fine-tuning code attack (with 100 token prefix) varying duplication rates

As a analysis, we perform the pre-training code attack on Mellum-4B. We note that the pre-training data overlap between Mellum and the attack dataset constructed from The Stack v2 cannot be fully verified; consequently, these results should be interpreted with caution.

Table 12 presents the pre-training code attack across fine-tuning epochs. The pre-trained Mellum-4B model yields an EM rate of 1.0%, which drops to 0.1% after the first epoch and remains below 0.3% throughout fine-tuning. While the EM rates are too low to draw strong conclusions, the fuzzy metrics reveal a clear forgetting pattern: the BLEU score drops from 41.5% at epoch zero to 19.7% after three epochs, and METEOR decreases from 50.0% to 31.7%. This behavior is directionally consistent with the forgetting observed in StarCoder2, where fine-tuning reduces the extractability of pre-training data. Figure 20 illustrates the EM trend across epochs.

Table 13 shows the pre-training code attack on the fine-tuned model with varying prefix lengths. The EM rate remains stable at 0.2%–0.3% regardless of prefix length, indicating that extending the prefix does not recover extractability at the exact match level. However, the fuzzy metrics show a modest increase: BLEU rises from 19.7% at 100 tokens to 24.1% at 250 tokens, and ROUGE-L from 32.0% to 37.9%. Figure 21 compares the pre-training and fine-tuning attack results across prefix lengths.

Table 7 Fine-tuning code attack on fine-tuned models with varying prefix lengths

		Memorization rate			
Model	Prefix length	EM	BLEU	EM↑%	BLEU↑%
Starcoder2-3B	100 tokens	0.631 [.601, .660]	0.764 [.741, .788]	0.367	0.226
	150 tokens	0.689 [.660, .717]	0.785 [.762, .809]	0.393	0.231
	200 tokens	0.747 [.719, .773]	0.816 [.795, .839]	0.443	0.260
	250 tokens	0.815 [.790, .838]	0.865 [.845, .885]	0.502	0.297
Starcoder2-7B	100 tokens	0.578 [.547, .608]	0.750 [.726, .774]	0.330	0.202
	150 tokens	0.646 [.616, .675]	0.773 [.750, .796]	0.386	0.220
	200 tokens	0.712 [.683, .739]	0.802 [.780, .824]	0.436	0.243
	250 tokens	0.800 [.774, .824]	0.850 [.829, .870]	0.492	0.245
Starcoder2-15B	100 tokens	0.610 [.578, .639]	0.775 [.752, .798]	0.310	0.198
	150 tokens	0.730 [.701, .756]	0.805 [.783, .827]	0.372	0.216
	200 tokens	0.817 [.792, .840]	0.830 [.807, .851]	0.441	0.232
	250 tokens	0.888 [.867, .906]	0.868 [.848, .889]	0.474	0.244

RQ5: The qualitative memorization patterns observed in StarCoder2 generalize to the Mellum-4B architecture: extractability of fine-tuning data increases with training epochs, duplication rate, and prefix length, while fine-tuning reduces the extractability of pre-training data. However, Mellum-4B exhibits substantially lower memorization rates across all conditions. License information remains the most vulnerable category.

7.6 Model Compression

To assess the impact of model compression, we repeated both the pre-training and fine-tuning code attacks using the StarCoder2-15B model. By default, StarCoder2-15B is configured at full-precision (fp32). In this section, we introduce compression as an additional

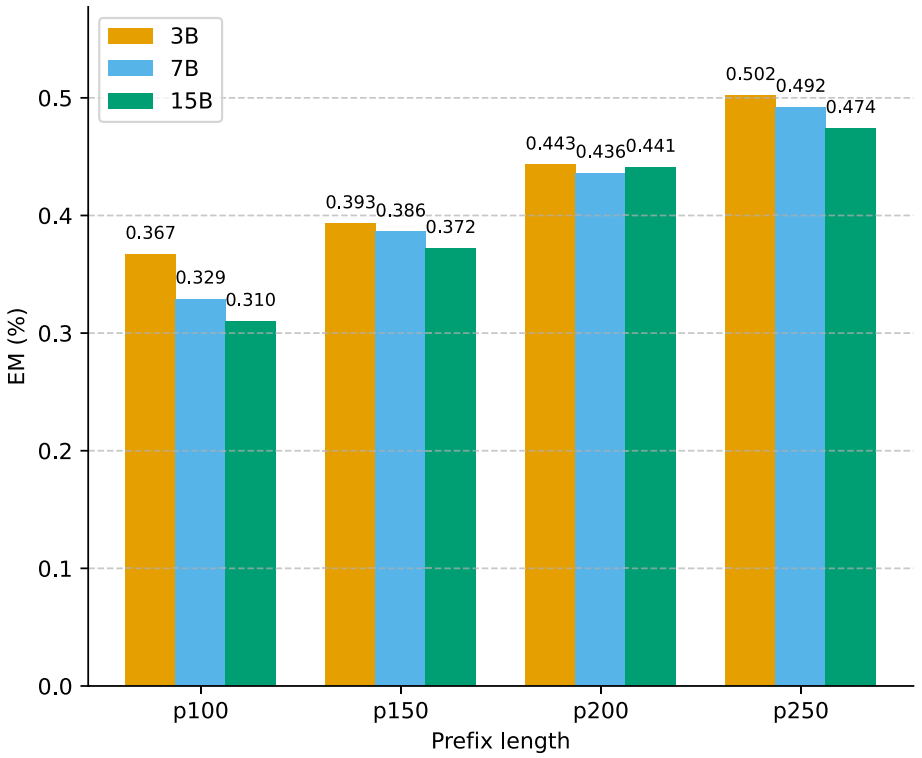


Fig. 12 EM ↑% when performing the fine-tuning code on fine-tuned models with varying prefix lengths

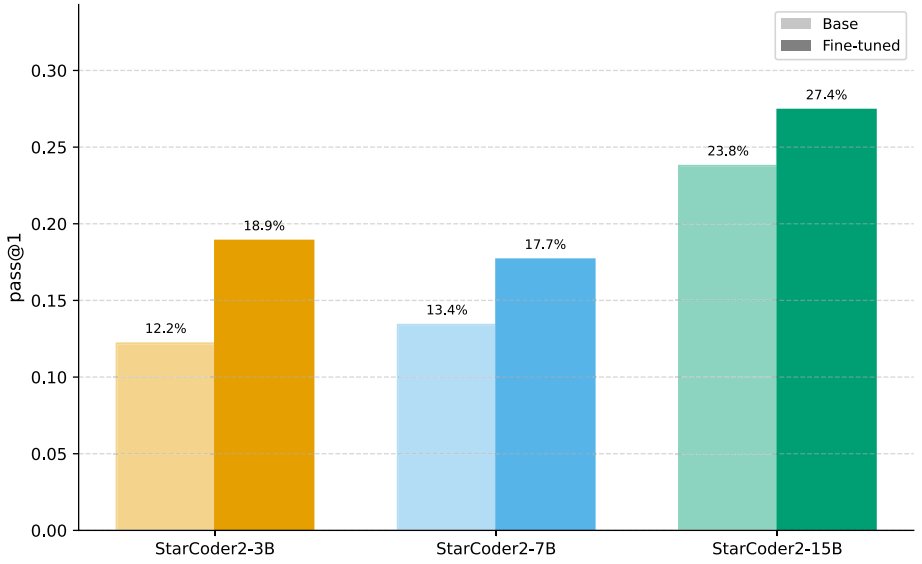


Fig. 13 pass@1 on base and fine-tuned models on the Java line completion task

Table 8 Code attack results per data category and model

Code Attack type		Pre-training (RQ1-2)		Fine-tuning (RQ3)
Model	Category	Base	Fine-tuned	Fine-tuned
Starcoder2-3B	Code	115	53	56
	Dicts	124	31	54
	Docs	32	4	104
	License	85	8	185
	Test	3	2	1
Starcoder2-7B	Code	117	64	53
	Dicts	112	26	52
	Docs	42	5	123
	License	88	11	181
	Test	3	3	1
Starcoder2-15B	Code	136	62	88
	Dicts	133	47	64
	Docs	52	16	137
	License	94	15	195
	Test	4	4	1

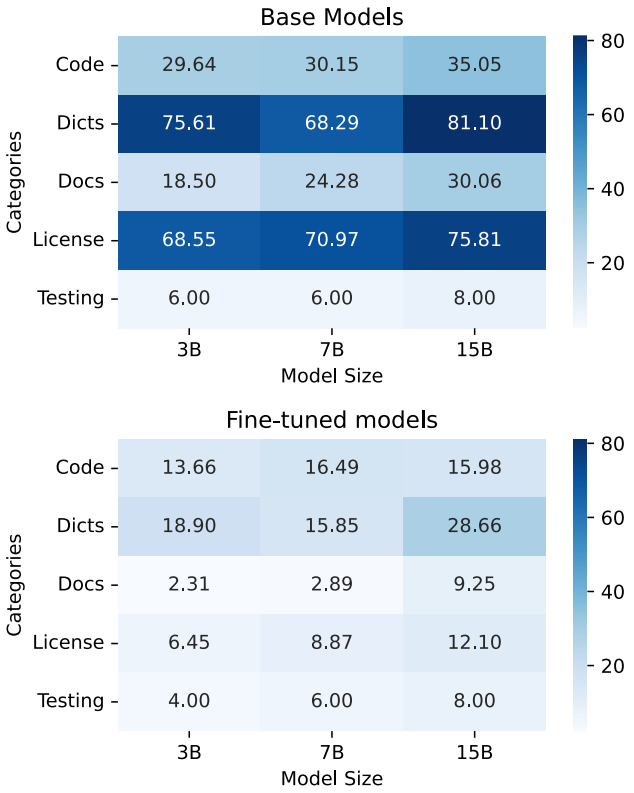


Fig. 14 Percentage of extractable pre-training samples per category on base and fine-tuned models

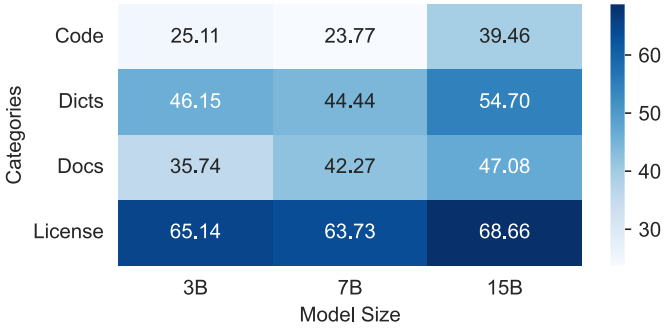


Fig. 15 Percentage of extractable fine-tuning samples per category

Table 9 Fine-tuning code attack on fine-tuned Mellum-4B across fine-tuning epochs

		Memorization rate			
Model	Epochs	EM	BLEU	EM ↑%	BLEU ↑%
Mellum-4B	0	0.084 [.068, .103]	0.461 [.438, .486]		
	1	0.095 [.078, .115]	0.518 [.496, .541]	0.011	0.057
	2	0.119 [.100, .141]	0.641 [.619, .664]	0.035	0.180
	3	0.134 [.114, .157]	0.690 [.668, .712]	0.050	0.229

experimental dimension, applying it to the largest model size and evaluating alternative configurations in bfloat16, int8 quantization, and int4 quantization.

Table 14 shows the results of the pre-training code attack on the pre-trained StarCoder2-15B model under different precision settings. Across all precisions, the memorization rate increases with longer input prefixes, with the highest values observed at 250 tokens. The bfloat16 and int8 configurations yield results that are nearly identical to the fp32 baseline for all metrics. The int4 configuration consistently produces lower scores, with reductions across all metrics. For example, at 100 tokens, the EM drops from 45.5% in the fp32 baseline to 41%, and at 250 tokens from 54.9% to 50.2%.

Tables 15 and 16 present the results of the pre-training code attack on the fine-tuned StarCoder2-15B model under different precision settings. After the first epoch of fine-tuning, the memorization rate drops sharply across all precisions, with EM decreasing from 45.5% at epoch zero to around 15% at epoch three. The bfloat16 and int8 configurations remain closely aligned with the fp32 baseline throughout fine-tuning, while the int4 setting results in slightly lower scores at all epochs. When varying the input prefix length, all configurations show higher memorization rates with longer prefixes. The number of successfully extracted pre-training samples is considerably lower than in the pre-trained model. For example, EM for the int8 model rises from 15.5% at 100 tokens to 23.6% at 250 tokens, whereas the corresponding pre-trained values were 45.7% and 54.6%. The bfloat16 and

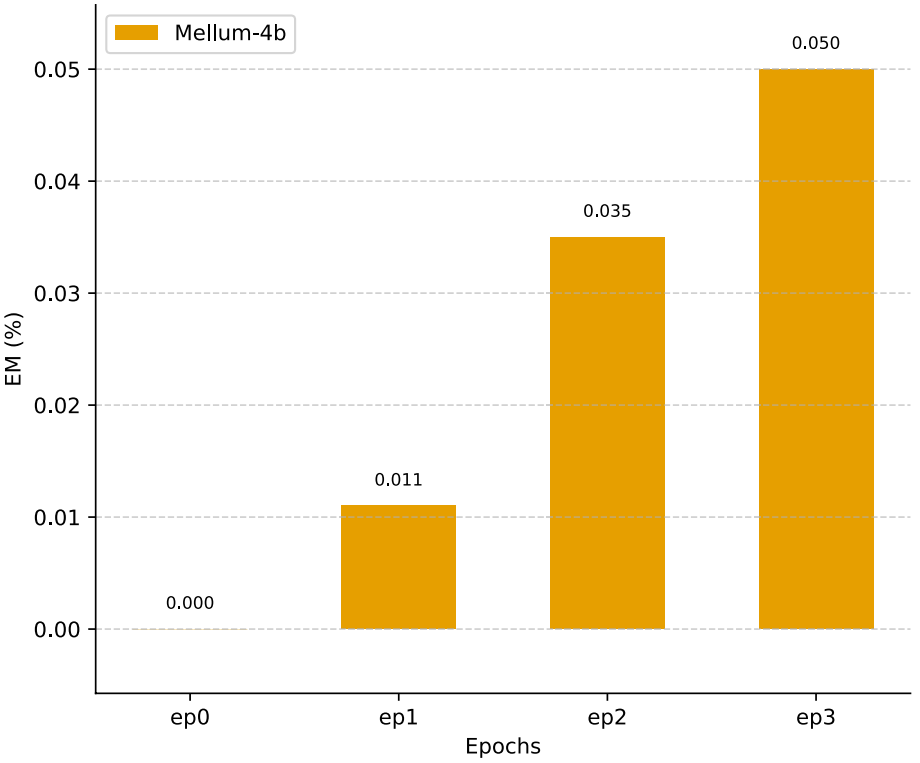


Fig. 16 EM ↑% rate when performing the fine-tuning code attack (with 100 token prefix) on Mellum-4B across fine-tuning epochs

Table 10 Fine-tuning code attack on fine-tuned Mellum-4B with varying duplication rates

		Memorization rate			
Model	Duplication	EM	BLEU	EM↑%	BLEU↑%
Mellum-4B	=1	0.005 [.002, .012]	0.369 [.352, .389]	0.005	0.112
	=2	0.045 [.034, .060]	0.521 [.500, .541]	0.026	0.188
	=3	0.139 [.119, .162]	0.637 [.614, .660]	0.064	0.312
	>3	0.130 [.111, .152]	0.692 [.670, .714]	0.046	0.234

int8 configurations closely mirror the fp32 trend. The int4 model follows the same trend but produces slightly lower scores at all prefix lengths. These results are described in Fig. 22, which highlights the consistent increase in extractability with longer prefixes and the overall reduction in memorization after fine-tuning across all precision settings, and int4's tendency toward lower scores across prefix lengths.

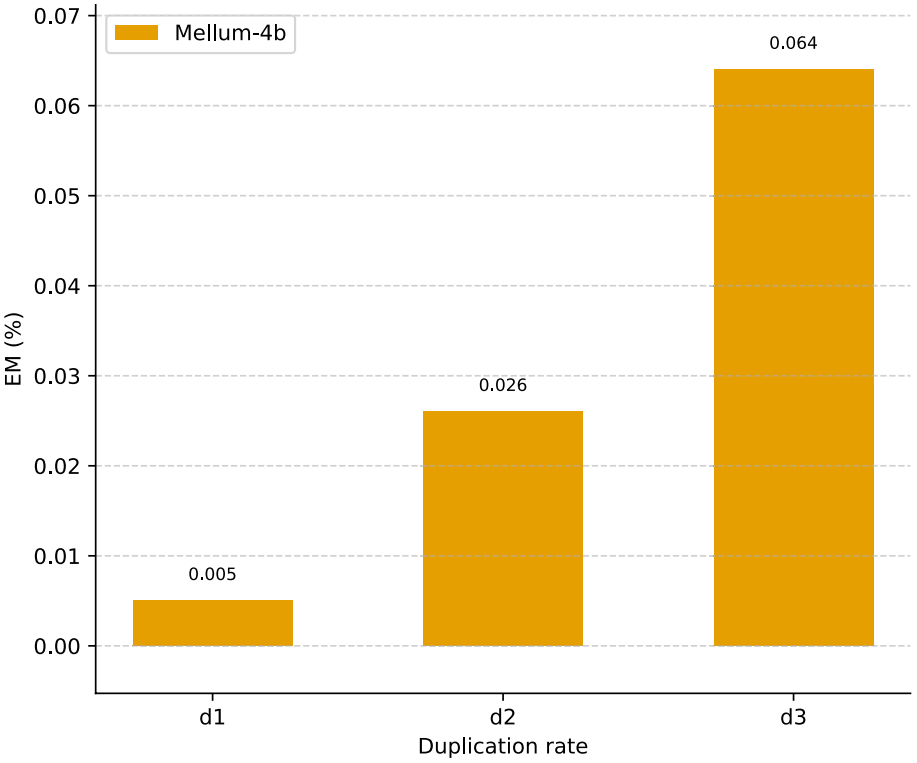


Fig. 17 EM ↑% rate when performing the fine-tuning code attack (with 100 token prefix) on Mellum-4B with varying duplication rates

Table 11 Fine-tuning code attack on fine-tuned Mellum-4B with varying prefix lengths

		Memorization rate			
Model	Prefix length	EM	BLEU	EM↑%	BLEU↑%
Mellum-4B	100 tokens	0.134 [.114, .157]	0.690 [.668, .712]	0.050	0.228
	150 tokens	0.171 [.149, .196]	0.735 [.713, .755]	0.081	0.264
	200 tokens	0.199 [.175, .225]	0.749 [.728, .770]	0.101	0.273
	250 tokens	0.204 [.180, .230]	0.768 [.748, .789]	0.095	0.287

Tables 17, 18, and 19 present the results of the fine-tuning code attack on the fine-tuned StarCoder2-15B model under different precision settings. Across training epochs, the bfloat16 and int8 configurations closely reflect the fp32 baseline. For instance, in the int8 model, EM rises from 29.7% at the first epoch to 61.0% by the third, with EM ↑% increasing

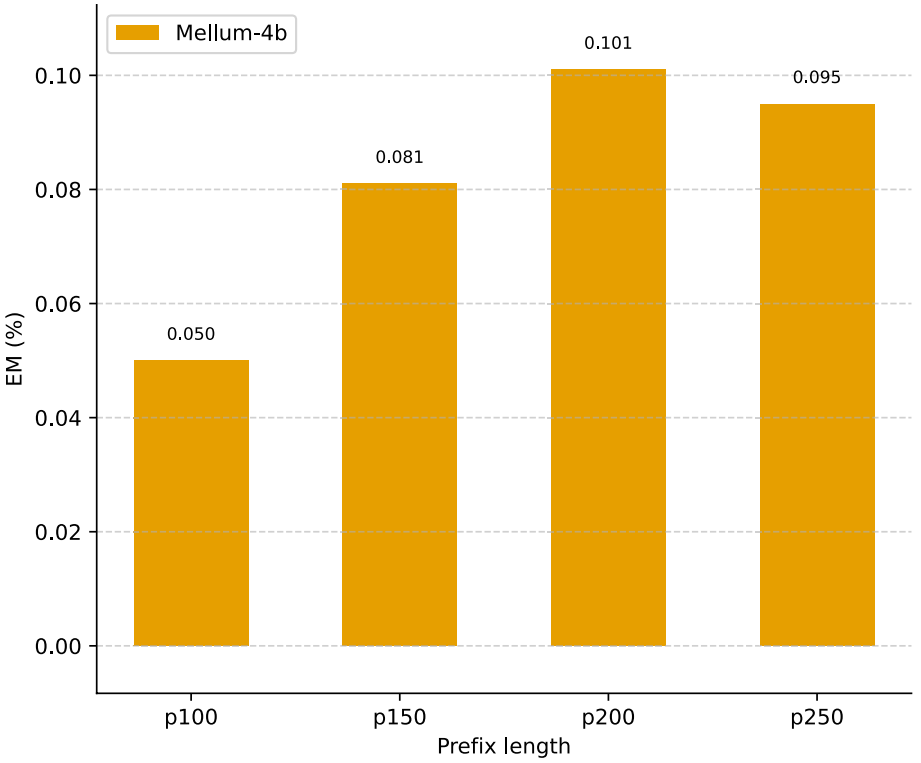


Fig. 18 EM ↑% rate when performing the fine-tuning code attack on Mellum-4B with varying prefix lengths

from 24.3% to 31.3%. By contrast, the int4 model shows a different trend. While it begins with a lower residual memorization (21.4% EM at epoch zero), its EM ↑% values (30.8 at epoch one and 36.9% at epoch three) show that it tends to memorize more compared to the baseline as fine-tuning progresses. This shows that while int4 compression suppresses pre-existing memorization, it amplifies relative memorization gains during fine-tuning. Under varying duplication rates, the bfloat16 and int8 models follow equivalent behavior to the fp32 baseline. For instance, at three duplications, the baseline reaches an EM ↑% of 41.7%, while bfloat16 and int8 record nearly identical values of 41.8% and 41.5%, respectively. The int4 model departs slightly from this pattern. While its EM ↑% remains close to the other formats at low duplication levels, it is marginally lower than the baseline for one, two, and three duplications. At higher duplication levels, however, the trend reverses. For >3 duplications the int4 model reaches 36.9% compared to 30.9% of the fp32 baseline. This indicates that the tendency toward stronger relative gains, already observed in the epoch experiments, holds for samples duplicated more than three times during fine-tuning, but not for less frequently seen samples. For varying prefix lengths, the bfloat16 and int8 configurations closely reflect the fp32 baseline. All three show steadily increasing EM and EM ↑% as the prefix grows longer. For example, at 100 tokens the fp32 model attains 61.0% EM with an EM ↑% of 31.0%, and at 250 tokens it reaches 88.8% EM with an EM ↑% of 47.4%. The int4 model departs from this pattern. At shorter prefixes, it records slightly lower residual

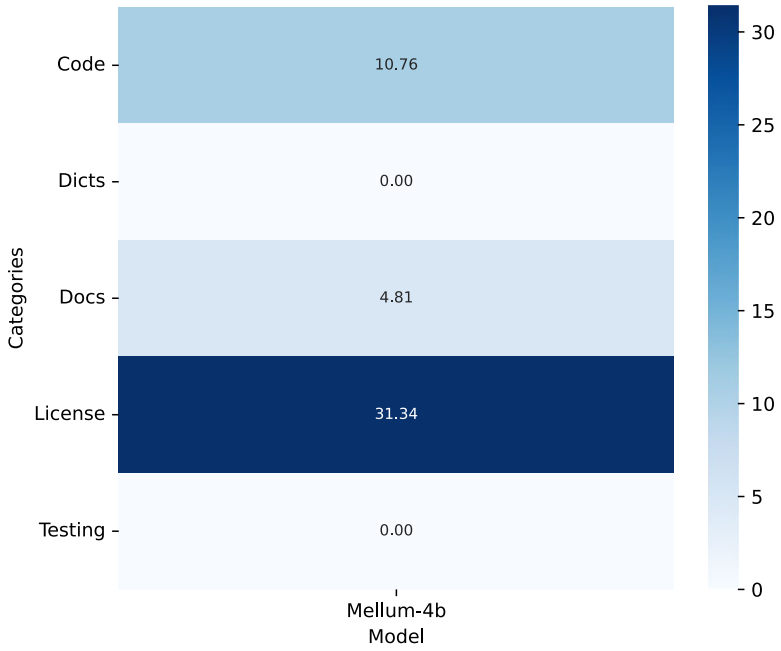


Fig. 19 Percentage of extractable fine-tuning samples per category for the fine-tuned Mellum-4B

Table 12 Pre-training code attack on Mellum-4B across fine-tuning epochs

		Memorization rate			
Model	Epochs	EM	BLEU	METEOR	ROUGE-L
Mellum-4B	0	0.010 [.005, .018]	0.415 [.394, .436]	0.500 [.482, .518]	0.541 [.522, .560]
	1	0.001 [.000, .006]	0.181 [.168, .194]	0.305 [.290, .319]	0.304 [.290, .320]
	2	0.002 [.001, .007]	0.192 [.179, .205]	0.311 [.297, .326]	0.315 [.299, .330]
	3	0.003 [.001, .009]	0.197 [.183, .210]	0.317 [.302, .332]	0.320 [.305, .335]

memorization (EM 58.3% at 100 tokens) but substantially higher EM ↑% (36.9% at 100 tokens). As the prefix length increases, this relative edge grows. At 250 tokens, int4 reaches 86.3% EM while its EM ↑% rises to 57.3%, compared to the fp32 baseline EM ↑% of 47.4%. Figure 23 illustrates this contrast: bf16 and int8 closely follow the fp32 trend, while int4 consistently memorizes more as the prefix length extends.

Table 20, Figs. 24, and' 25 present the distribution of extracted data categories for the StarCoder2-15B model under different precision settings. Across code categories, the overall memorization patterns remain the same under different precision settings, with only minor numerical fluctuations. In the pre-training code attack on base models, the largest share of

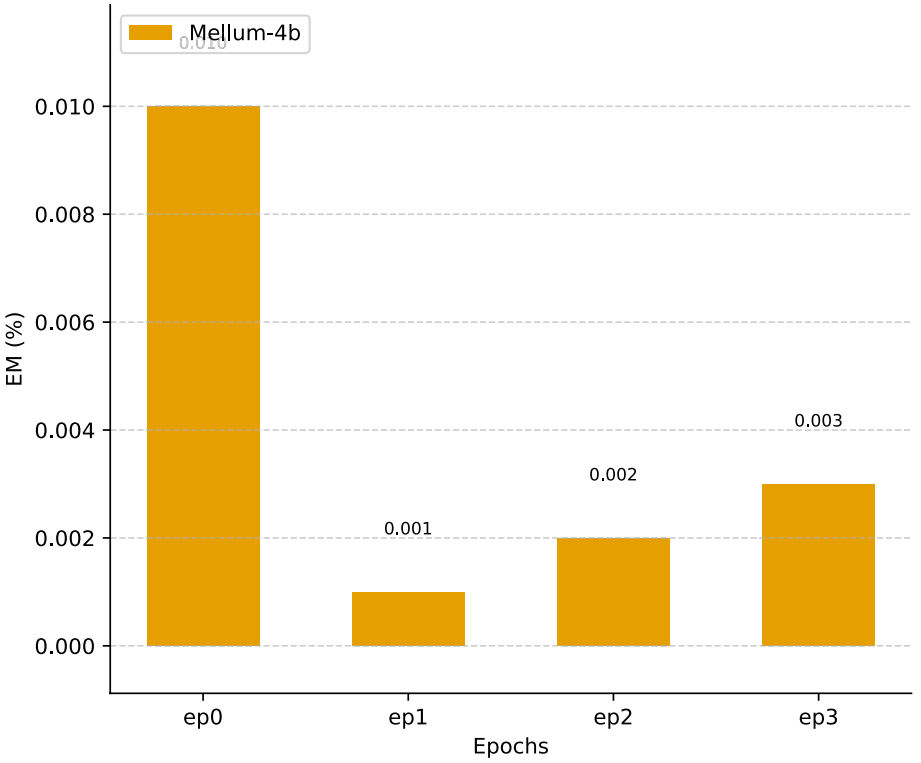


Fig. 20 EM rate when performing the pre-training code attack (with 100 token prefix) on Mellum-4B across fine-tuning epochs

Table 13 Pre-training code attack on fine-tuned Mellum-4B varying the input prefix length

		Memorization rate			
Model	Prefix length	EM	BLEU	METEOR	ROUGE-L
Mellum-4B	100 tokens	0.003 [.001, .009]	0.197 [.183, .210]	0.317 [.302, .332]	0.320 [.305, .335]
	150 tokens	0.002 [.001, .007]	0.216 [.202, .230]	0.339 [.325, .354]	0.349 [.334, .365]
	200 tokens	0.003 [.001, .009]	0.230 [.215, .244]	0.349 [.335, .365]	0.365 [.349, .381]
	250 tokens	0.003 [.001, .009]	0.241 [.226, .256]	0.360 [.344, .375]	0.379 [.364, .395]

memorized samples comes from Dicts and License categories across all precision settings. For example, the bfloat16 model recovers 133 dictionaries and 93 licenses, while the int8 model yields slightly fewer, with 115 dictionaries and 82 licenses. The int4 variant shows a similar behavior, with 133 dictionaries and 96 licenses. When performing the pre-training

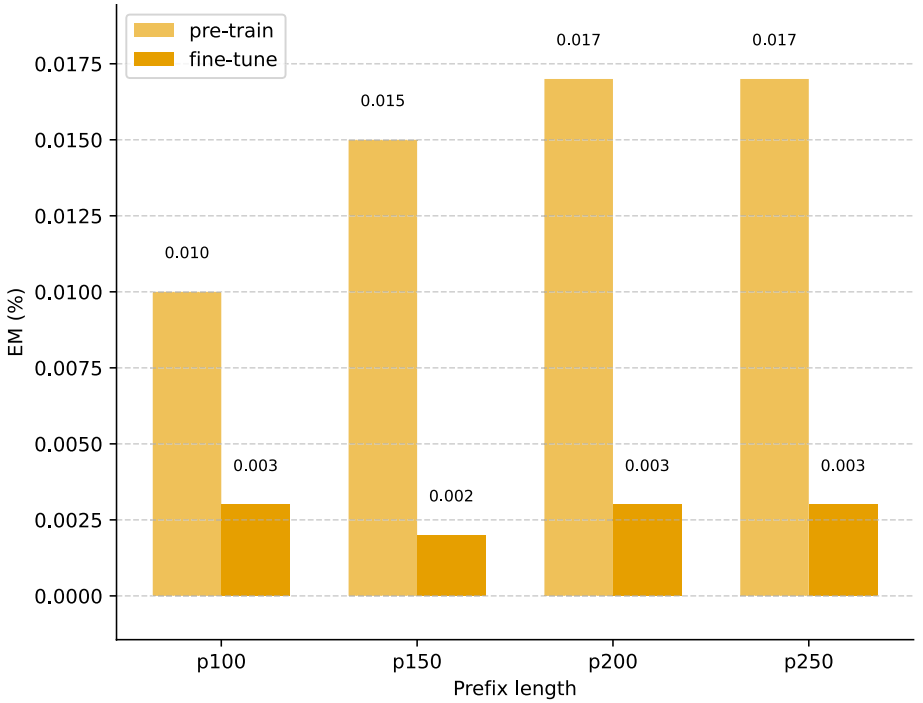


Fig. 21 EM rate when performing the pre-training code attack on base and fine-tuned Mellum-4B varying the input prefix length

code attack on fine-tuned models, the same forgetting behavior is observed under compression. Licenses become much less extractable, dropping from 75.8% to 12.1% in the baseline and to similar levels across all compressed variants. Data carriers remain comparatively higher, with around 62 code samples retained in the fp32 model and nearly identical values in bfloat16 and int8, while from int4 are extracted slightly fewer samples (57). In the fine-tuning code attack, both licenses and data carriers are the most extracted across all settings. In the fp32 model, 88 code samples and 195 licenses were recovered, corresponding to 39.5% and 68.7% of the respective categories. These proportions remain nearly unchanged in bfloat16 and int8, while int4 shows small variations, with slightly fewer licenses (192) but comparable relative percentages.

RQ5: Model compression does not alter the overall memorization trends. In the pre-training code attack, both pre-trained and fine-tuned models exhibit the same patterns across all precision settings. In the fine-tuning code attack, however, the int4 configuration shows stronger relative memorization than the fp32 baseline, while bfloat16 and int8 closely replicate it. Finally, the distribution of memorized code categories remains stable, with data carriers and licenses consistently the most extractable across all settings.

Table 14 Pre-training code attack on pre-trained StarCoder2-15B under different precision settings, varying the input prefix length

		Memorization rate			
Model	Prefix length	EM	BLEU	METEOR	ROUGE-L
fp32 (full-precision)	100 tokens	0.455 [.424, .486]	0.714 [.694, .736]	0.760 [.741, .778]	0.738 [.719, .758]
	150 tokens	0.506 [.475, .537]	0.757 [.737, .777]	0.770 [.774, .809]	0.774 [.756, .794]
	200 tokens	0.536 [.505, .567]	0.777 [.757, .796]	0.806 [.788, .824]	0.794 [.776, .812]
	250 tokens	0.549 [.518, .580]	0.790 [.770, .808]	0.819 [.802, .836]	0.805 [.787, .823]
bfloat16	100 tokens	0.453 [.422, .484]	0.714 [.692, .734]	0.759 [.740, .778]	0.739 [.719, .758]
	150 tokens	0.508 [.477, .539]	0.759 [.739, .779]	0.793 [.775, .811]	0.776 [.757, .795]
	200 tokens	0.535 [.504, .566]	0.776 [.756, .795]	0.806 [.789, .824]	0.793 [.775, .811]
	250 tokens	0.551 [.520, .582]	0.789 [.770, .808]	0.819 [.802, .836]	0.805 [.787, .823]
int8 quantization	100 tokens	0.457 [.426, .488]	0.712 [.691, .733]	0.757 [.739, .776]	0.737 [.717, .757]
	150 tokens	0.500 [.469, .531]	0.751 [.731, .771]	0.787 [.769, .805]	0.771 [.751, .790]
	200 tokens	0.537 [.506, .568]	0.777 [.757, .796]	0.809 [.791, .826]	0.795 [.777, .813]
	250 tokens	0.546 [.515, .577]	0.787 [.768, .806]	0.816 [.799, .833]	0.802 [.784, .820]
int4 quantization	100 tokens	0.410 [.380, .441]	0.694 [.672, .714]	0.742 [.723, .760]	0.719 [.699, .738]
	150 tokens	0.451 [.420, .482]	0.733 [.712, .752]	0.775 [.757, .792]	0.755 [.736, .774]
	200 tokens	0.469 [.438, .500]	0.748 [.727, .768]	0.785 [.767, .803]	0.769 [.750, .787]
	250 tokens	0.502 [.471, .533]	0.767 [.747, .786]	0.802 [.784, .819]	0.788 [.770, .806]

Table 15 Pre-training code attack on fine-tuned StarCoder2-15B under different precision settings, across several fine-tuning epochs

		Memorization rate			
Model	Epochs	EM	BLEU	METEOR	ROUGE-L
fp32 (full-precision)	0	0.455 [.424, .486]	0.714 [.694, .736]	0.760 [.741, .778]	0.738 [.719, .758]
	1	0.145 [.125, .168]	0.525 [.505, .546]	0.613 [.595, .632]	0.583 [.563, .603]
	2	0.143 [.123, .166]	0.530 [.510, .551]	0.619 [.600, .637]	0.585 [.566, .605]
	3	0.155 [.134, .179]	0.544 [.524, .565]	0.629 [.611, .648]	0.595 [.577, .615]
bfloat16	0	0.453 [.422, .484]	0.714 [.692, .734]	0.759 [.740, .778]	0.739 [.719, .758]
	1	0.142 [.122, .165]	0.525 [.505, .545]	0.613 [.595, .632]	0.583 [.563, .603]
	2	0.139 [.119, .162]	0.531 [.511, .551]	0.619 [.600, .637]	0.586 [.567, .606]
	3	0.155 [.134, .179]	0.544 [.524, .565]	0.629 [.611, .647]	0.596 [.576, .615]
int8 quantization	0	0.457 [.426, .488]	0.712 [.691, .733]	0.757 [.739, .776]	0.737 [.717, .757]
	1	0.136 [.116, .159]	0.521 [.501, .542]	0.612 [.594, .631]	0.579 [.560, .599]
	2	0.132 [.112, .154]	0.526 [.506, .546]	0.614 [.595, .632]	0.583 [.564, .602]
	3	0.155 [.134, .179]	0.539 [.518, .559]	0.624 [.606, .643]	0.590 [.571, .609]
int4 quantization	0	0.410 [.380, .441]	0.694 [.672, .714]	0.742 [.723, .760]	0.719 [.699, .738]
	1	0.132 [.112, .154]	0.506 [.485, .526]	0.599 [.581, .618]	0.565 [.546, .585]
	2	0.135 [.115, .158]	0.513 [.492, .533]	0.604 [.585, .622]	0.570 [.551, .590]
	3	0.144 [.124, .167]	0.519 [.499, .539]	0.606 [.588, .625]	0.572 [.553, .592]

Table 16 Pre-training code attack on fine-tuned StarCoder2-15B under different precision settings, varying the input prefix length

		Memorization rate			
Model	Prefix length	EM	BLEU	METEOR	ROUGE-L
fp32 (full-precision)	100 tokens	0.155 [.134, .179]	0.544 [.525, .565]	0.629 [.611, .648]	0.595 [.577, .615]
	150 tokens	0.198 [.174, .224]	0.577 [.557, .598]	0.657 [.639, .675]	0.624 [.605, .644]
	200 tokens	0.220 [.195, .247]	0.600 [.578, .619]	0.675 [.657, .694]	0.645 [.626, .665]
	250 tokens	0.235 [.210, .262]	0.618 [.598, .639]	0.687 [.669, .706]	0.660 [.641, .679]
bfloat16	100 tokens	0.155 [.134, .179]	0.544 [.525, .564]	0.629 [.611, .648]	0.596 [.577, .615]
	150 tokens	0.198 [.174, .224]	0.578 [.557, .598]	0.658 [.640, .676]	0.625 [.606, .645]
	200 tokens	0.221 [.196, .248]	0.600 [.579, .620]	0.676 [.657, .694]	0.646 [.626, .665]
	250 tokens	0.236 [.211, .263]	0.619 [.598, .640]	0.688 [.670, .706]	0.661 [.642, .680]
int8 quantization	100 tokens	0.155 [.134, .179]	0.539 [.519, .559]	0.624 [.606, .643]	0.590 [.570, .609]
	150 tokens	0.191 [.168, .217]	0.575 [.555, .596]	0.655 [.637, .674]	0.624 [.604, .643]
	200 tokens	0.211 [.187, .237]	0.599 [.578, .619]	0.677 [.658, .695]	0.645 [.625, .664]
	250 tokens	0.225 [.200, .252]	0.614 [.593, .634]	0.684 [.666, .702]	0.659 [.639, .678]
int4 quantization	100 tokens	0.144 [.124, .167]	0.519 [.499, .539]	0.606 [.588, .625]	0.572 [.553, .591]
	150 tokens	0.174 [.152, .199]	0.560 [.540, .580]	0.642 [.623, .660]	0.609 [.589, .629]
	200 tokens	0.203 [.179, .229]	0.592 [.571, .612]	0.671 [.652, .689]	0.641 [.621, .660]
	250 tokens	0.215 [.191, .242]	0.605 [.584, .625]	0.678 [.660, .696]	0.651 [.632, .670]

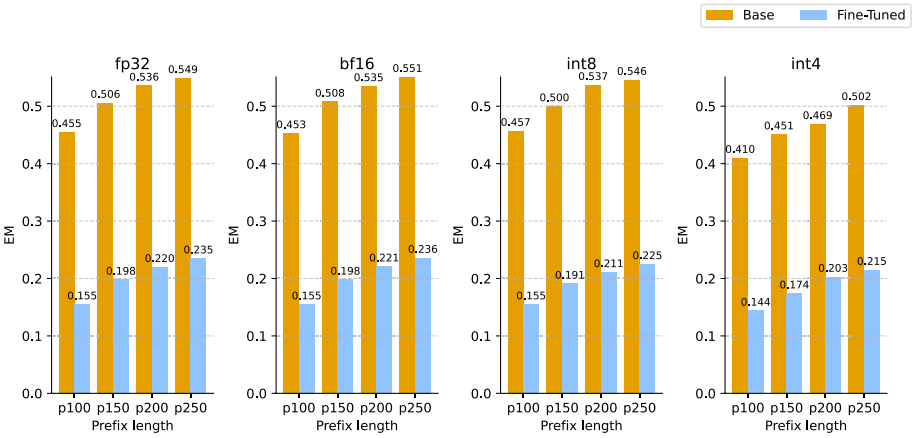


Fig. 22 EM rate when performing the pre-training code attack on base and fine-tuned models under different precision settings, varying the input prefix length

7.7 PEFT

To evaluate the impact of PEFT we tune StarCoder-7B using LoRA on the same fine-tuning data as the other experiments. After 3 epochs we observe that the eval loss is 0.66, higher than full fine-tuning which was able to achieve a loss of 0.58 after 3 epochs.

Figure 26 shows the results of applying the pre-training attack on StarCoder-7B and the associated PEFT model. As mentioned earlier, the StarCoder-7B model suffers from a quite pronounced drop in pre-training data extractability after the first epoch, and then recovers in the second and third epoch. We can observe that the forgetting behaviour for LoRA is less pronounced than in full fine-tuning. The rate of forgetting is also more constant across different epochs, with no sudden drop in the first epoch, but with more consistent decrease across epochs.

In Fig. 27 we show the results of the fine-tuning attack. We can see that the LoRA tuned model does not tend to memorize data at the same rate as fine-tuned models. The rate after three epochs does not exceed the base rate of fine-tuning memorization. The increase between epochs is also minimal.

In Figs. 28 and 29 we show the impact of prefix length on the memorisation rate after three epochs. We observe that the LoRA model is more sensitive to the prefix length when recalling pretraining data. The EM rate of 0.281 at a prefix length of 100 increases to 0.328 (EM ↑%) when the prefix is increased to 250 tokens.

On the other hand, we find that the LoRA-tuned model exhibits significantly reduced sensitivity to prefix length increases during the fine-tuning attack. The EM rate only increases modestly from 0.235 to 0.265 (EM ↑%) when the prefix is extended from 100 to 250 tokens, representing a 11% improvement. In stark contrast, the full fine-tuned model demonstrates a substantial increase from 0.578 to 0.800 (EM ↑%), reflecting a 28% improvement.

LoRA exhibits distinct memorization patterns compared to full fine-tuning. For pre-training data, LoRA demonstrates more gradual and consistent forgetting across epochs without the catastrophic drop observed in full fine-tuning, while maintaining higher sensitivity to prefix length variations. For fine-tuning data, LoRA shows significantly reduced memoriza-

Table 17 Fine-tuning code attack on fine-tuned StarCoder2-15B under different precision settings, across fine-tuning epochs

		Memorization rate			
Model	Epochs	EM	BLEU	EM ↑%	BLEU ↑%
fp32 (full-precision)	0	0.300 [.272, .329]	0.576 [.551, .601]		
	1	0.549 [.518, .580]	0.756 [.733, .779]	0.249	0.181
	2	0.567 [.536, .597]	0.761 [.738, .784]	0.267	0.185
	3	0.609 [.578, .639]	0.774 [.752, .797]	0.309	0.198
bfloat16	0	0.302 [.274, .331]	0.576 [.551, .601]		
	1	0.550 [.519, .581]	0.757 [.734, .781]	0.248	0.181
	2	0.571 [.540, .601]	0.761 [.738, .784]	0.269	0.185
	3	0.610 [.579, .640]	0.773 [.750, .796]	0.308	0.197
int8 quantization	0	0.297 [.269, .326]	0.571 [.546, .596]		
	1	0.540 [.509, .571]	0.756 [.732, .779]	0.243	0.184
	2	0.563 [.532, .593]	0.759 [.735, .782]	0.266	0.187
	3	0.610 [.579, .640]	0.774 [.751, .797]	0.313	0.203
int4 quantization	0	0.214 [.189, .241]	0.507 [.483, .531]		
	1	0.522 [.491, .553]	0.747 [.723, .771]	0.308	0.240
	2	0.550 [.519, .581]	0.755 [.732, .779]	0.336	0.248
	3	0.583 [.552, .613]	0.768 [.745, .791]	0.369	0.261

Table 18 Fine-tuning code attack on fine-tuned StarCoder2-15B under different precision settings, with varying duplication rates

		Memorization rate			
Model	Duplication	EM	BLEU	EM↑%	BLEU↑%
fp32 (full-precision)	=1	0.075 [.060, .093]	0.475 [.457, .494]	0.052	0.110
	=2	0.352 [.323, .382]	0.694 [.674, .715]	0.277	0.244
	=3	0.544 [.513, .575]	0.832 [.815, .849]	0.417	0.385
	>3	0.609 [.578, .639]	0.775 [.752, .798]	0.310	0.198
bfloat16	=1	0.078 [.063, .096]	0.476 [.457, .495]	0.055	0.106
	=2	0.352 [.323, .382]	0.694 [.673, .714]	0.277	0.246
	=3	0.545 [.514, .576]	0.835 [.818, .852]	0.418	0.386
	>3	0.610 [.579, .640]	0.773 [.750, .796]	0.308	0.197
int8 quantization	=1	0.077 [.062, .095]	0.478 [.459, .497]	0.053	0.112
	=2	0.350 [.321, .380]	0.692 [.671, .712]	0.276	0.242
	=3	0.541 [.510, .572]	0.831 [.814, .847]	0.415	0.387
	>3	0.610 [.579, .640]	0.774 [.751, .797]	0.313	0.202
int4 quantization	=1	0.079 [.064, .097]	0.471 [.452, .489]	0.056	0.108
	=2	0.331 [.303, .361]	0.677 [.656, .698]	0.258	0.228
	=3	0.512 [.481, .543]	0.814 [.795, .831]	0.393	0.374
	>3	0.583 [.552, .613]	0.768 [.745, .791]	0.369	0.261

Table 19 Fine-tuning code attack on fine-tuned StarCoder2-15B under different precision settings, with varying prefix lengths

		Memorization rate			
Model	Prefix length	EM	BLEU	EM↑%	BLEU↑%
fp32 (full-precision)	100 tokens	0.610 [.578, .639]	0.775 [.752, .798]	0.310	0.198
	150 tokens	0.730 [.701, .756]	0.805 [.783, .827]	0.372	0.216
	200 tokens	0.817 [.792, .840]	0.830 [.807, .851]	0.441	0.232
	250 tokens	0.888 [.867, .906]	0.868 [.848, .889]	0.474	0.244
bfloat16	100 tokens	0.610 [.579, .640]	0.773 [.750, .796]	0.308	0.197
	150 tokens	0.729 [.701, .756]	0.806 [.784, .828]	0.372	0.214
	200 tokens	0.818 [.793, .841]	0.830 [.808, .852]	0.446	0.231
	250 tokens	0.886 [.865, .904]	0.868 [.848, .889]	0.473	0.243
int8 quantization	100 tokens	0.610 [.579, .640]	0.774 [.751, .797]	0.313	0.202
	150 tokens	0.730 [.702, .757]	0.808 [.785, .829]	0.375	0.221
	200 tokens	0.815 [.790, .838]	0.830 [.808, .851]	0.447	0.237
	250 tokens	0.887 [.866, .905]	0.869 [.850, .890]	0.469	0.241
int4 quantization	100 tokens	0.583 [.552, .613]	0.768 [.745, .791]	0.369	0.261
	150 tokens	0.699 [.670, .727]	0.799 [.776, .821]	0.451	0.282
	200 tokens	0.770 [.743, .795]	0.827 [.805, .849]	0.491	0.301
	250 tokens	0.863 [.840, .883]	0.865 [.845, .886]	0.573	0.313

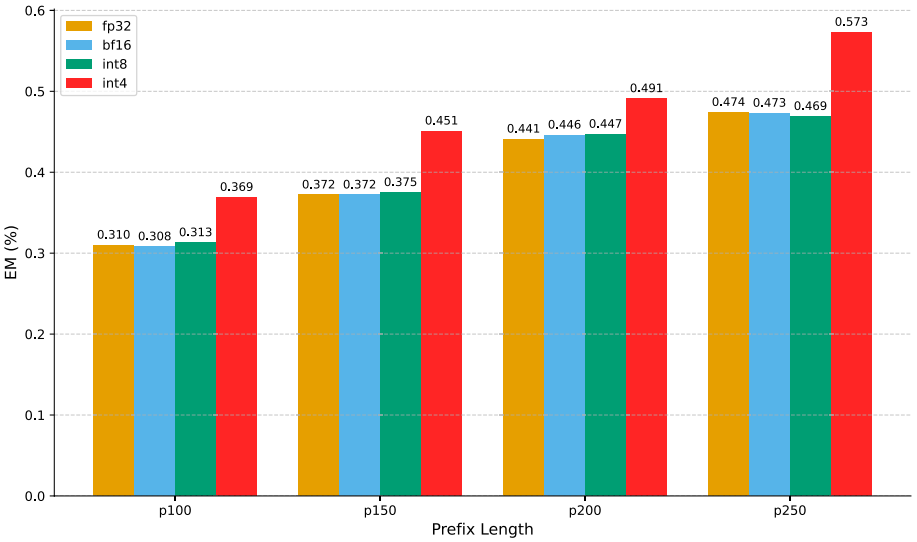


Fig. 23 EM ↑% rate when performing the fine-tuning code attack on fine-tuned models under different precision settings, varying the input prefix length

Table 20 Code attack results on StarCoder2-15B per data category and different precision settings

Code Attack type		Pre-training		Fine-tuning
Model	Category	Base	Fine-tuned	Fine-tuned
fp32 (full-precision)	Code	136	62	88
	Dicts	133	47	64
	Docs	52	16	137
	License	94	15	195
	Test	4	4	1
bfloat16	Code	134	61	87
	Dicts	133	47	65
	Docs	52	17	137
	License	93	16	196
	Test	4	3	1
int8 quantization	Code	128	62	89
	Dicts	115	46	63
	Docs	46	19	136
	License	82	15	196
	Test	4	3	1
int4 quantization	Code	136	57	77
	Dicts	133	41	67
	Docs	53	17	117
	License	96	15	192
	Test	4	4	1

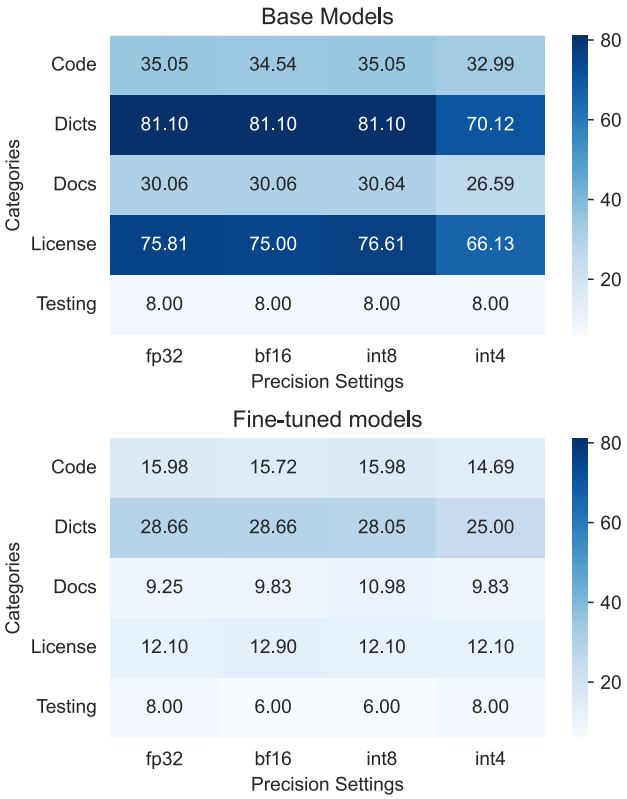


Fig. 24 Percentage of extractable pre-training samples per category for the base and fine-tuned StarCoder2-15B model across different precision settings

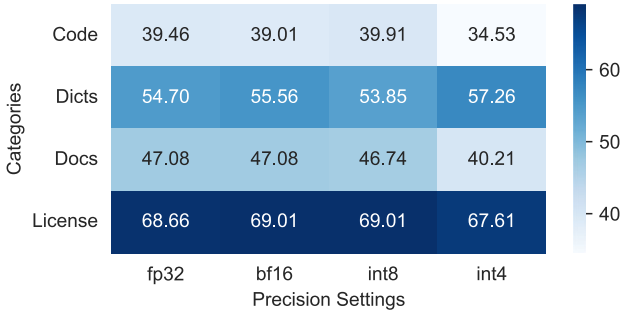


Fig. 25 Percentage of extractable fine-tuning samples per category for the fine-tuned StarCoder2-15B model across different precision settings

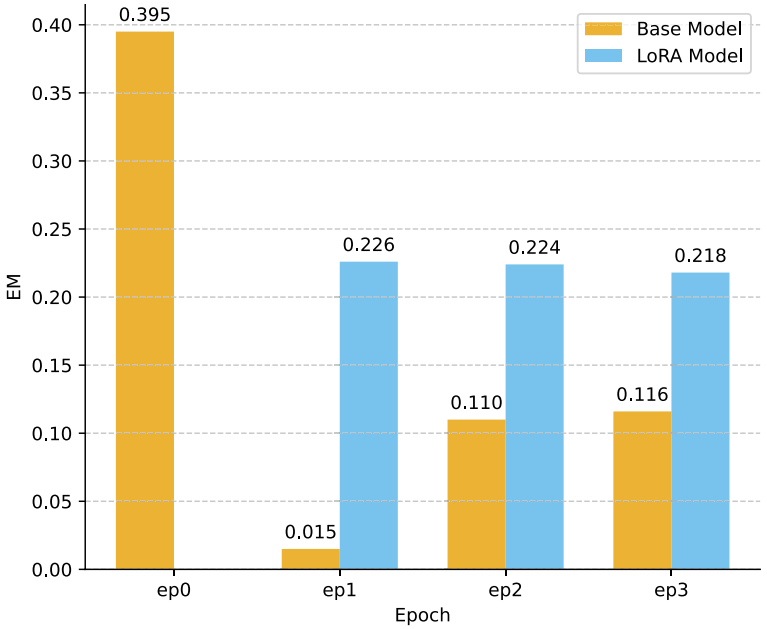


Fig. 26 EM rate when performing the pre-training code attack (with 100 token prefix) on base and PEFT models

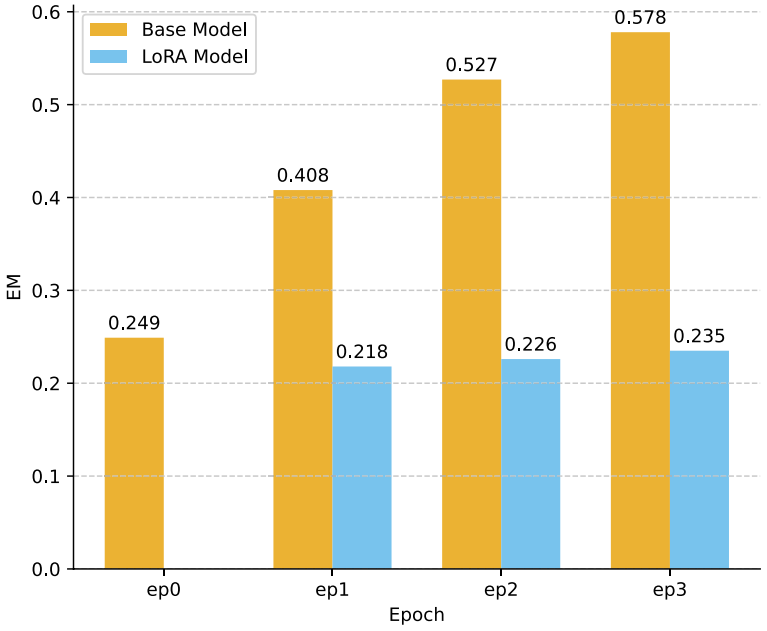


Fig. 27 EM rate when performing the fine-tuning code attack (with 100 token prefix) on base and PEFT models

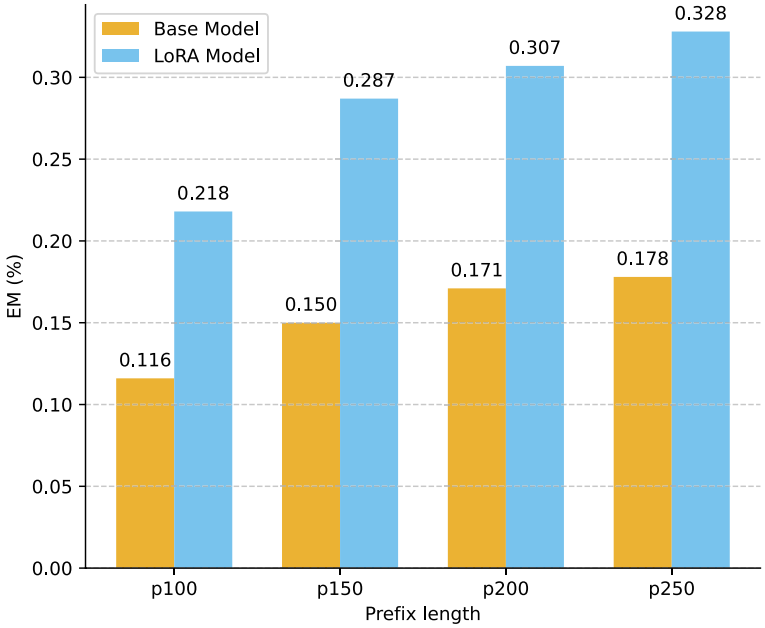


Fig. 28 EM ↑% when performing the pre-training code attack on base and PEFT models

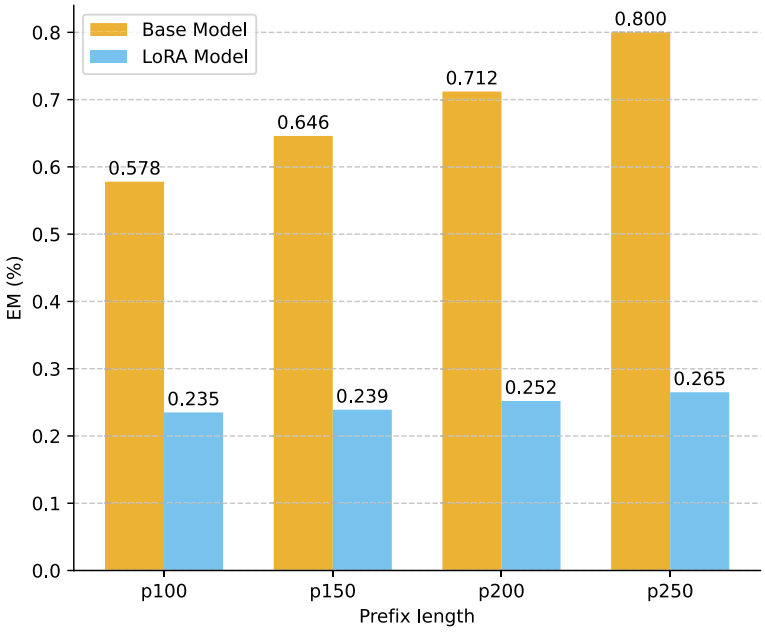


Fig. 29 EM ↑% when performing the fine-tuning code attack on base and PEFT models

tion capacity, with extraction rates remaining below the baseline and minimal sensitivity to prefix length changes.

In Figs. 31 and 30 we compare the rate of extraction between the full fine-tuned model and the LoRA model across different categories. In the pre-training data we can observe that the LoRA shows higher extractability across all data categories, execpt testing.

In the fine-tuning data (Fig. 30), only the licence information, which is the most extractable class at 64%, retains some extractability, decreasing to 30%. While the rest of the classes vary between 0% and 9% EM ↑%.

> **RQ6:** LoRA exhibits more gradual forgetting of pre-training data without the catastrophic drops seen in full fine-tuning, while showing reduced capacity to memorize fine-tuning data compared to full fine-tuning. LoRA models demonstrate higher sensitivity to prefix length for pre-training data extraction but significantly lower sensitivity for fine-tuning data extraction.

8 Discussion

8.1 Pre-Training Data Memorization - Pre-Trained Models

Our findings indicate that the StarCoder2 base model family tends to memorize its pre-training data. Consistent with prior research (Carlini et al. 2022b; Al-Kaswan et al. 2024;



Fig. 30 Percentage of extractable pre-training samples per category for the full fine-tuned and LoRA fine-tuned StarCoder2-7B

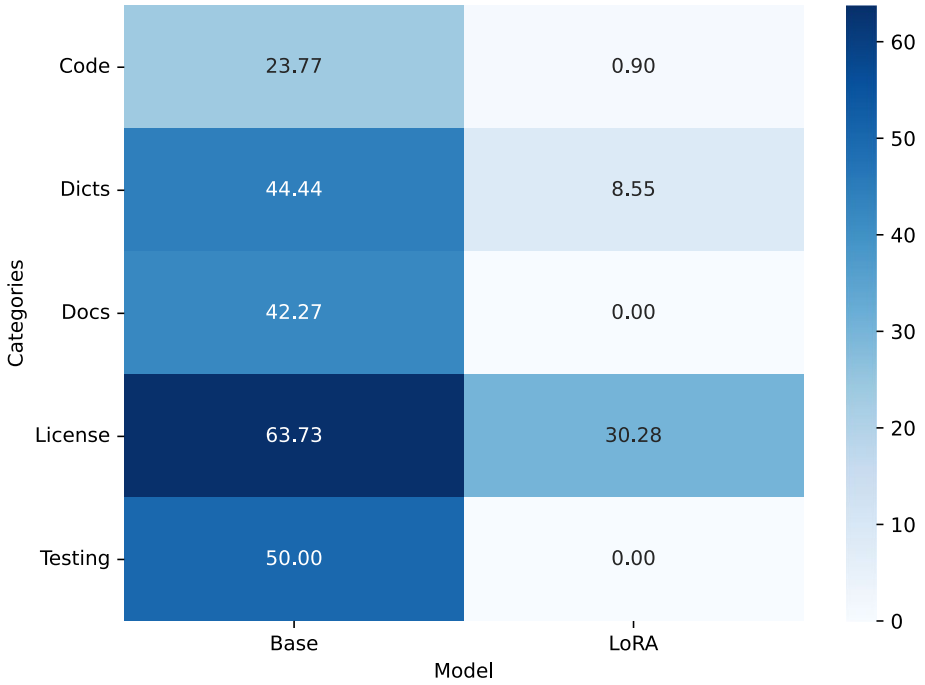


Fig. 31 Percentage of extractable fine-tuning samples per category for the full fine-tuned and LoRA fine-tuned StarCoder2-7B

Yang et al. 2024), we observed that the extent of memorization correlates with model size. Using a 100-token attack on the StarCoder2-15B model, we extracted 45.5% of the dataset. This rate dropped to 38.9% with the three-billion-parameter version. Our results substantiate the log-linear relation between size and memorization, which has been recently observed also for LLMs trained on code (Al-Kaswan et al. 2024).

Our experiments reveal that extending the input prefix length enhances the memorization rate of the pre-trained models when attacking with samples from the pre-training data. These findings align with previous studies, which demonstrated that longer input prefixes facilitate the extraction of more information from the model (Carlini et al. 2022b; Yang et al. 2024). Our results reinforce the connection identified by Carlini et al. between extractability and the precise identification of key information necessary for accessing specific details from a language model (Carlini et al. 2021).

Al-Kaswan et al. demonstrated that data carriers are memorized at a higher rate compared to regular code or documentation (Al-Kaswan et al. 2024). Our findings are consistent with this observation, as our categorization revealed that the types of code most frequently memorized by the pre-trained models were data carriers and licenses. We extracted up to 80% and 75% of the total data carriers and licenses from the attack dataset, but only 30% for code and documentation.

8.2 Pre-training Data Memorization - Fine-tuned Models

Performing the pre-training code attack on the fine-tuned models revealed that fine-tuning reduces the extractability of pre-training data. We observed that after three epochs of fine-tuning, around 30% of the samples that were previously extracted during the pre-training stage could no longer be extracted. To further investigate this phenomenon, we replicated the experiments conducted on the pre-trained models to examine how factors influencing pre-training extractability in pre-trained models affect fine-tuned models.

Our experiments revealed that the decrease in extractability is consistent across different model sizes to the same extent. Additional fine-tuning epochs do not significantly decrease extractability; despite some fluctuations, the EM rate does not consistently drop across epochs. For the 3B and 7B parameter models, extractability after the first epoch is slightly lower compared to later epochs. For example, the EM rate for the StarCoder2-3B model is 8.7% after the first fine-tuning epoch, increasing to 10% by the second epoch and remaining stable through the third epoch. This pattern likely reflects the model's initial adjustment to new, unseen data during the first epoch of fine-tuning (Liu et al. 2022). The model requires a few epochs to adapt to this new data distribution and perform the code completion task effectively. The initial drop at the first epoch is more pronounced in the 7B model. Lozhkov et al. state that it is not clear why StarCoder2-7B is not performing as well as the 3B and 15B models for their size (Lozhkov et al. 2024). We considered this as a plausible explanation for the observed discrepancies. Note that after the initial drop, extractability stabilizes in subsequent epochs.

Inspecting the type of pre-training code that is no longer extractable by fine-tuned models, showed that some categories are less extractable than others. Specifically, licenses become less extractable after fine-tuning compared to data carriers. StarCoder2-3B base model memorized 68.55% of the licenses, but this figure dropped to 6.45% after fine-tuning. Notably, data carriers exhibit the highest memorization rate post-fine-tuning, suggesting that this category of code is less affected by the "forgetting" phenomenon observed in the fine-tuned models.

8.3 Fine-Tuning Data Memorization

Performing the fine-tuning code attack revealed that, even only one training epoch is sufficient to make the model memorize 55% of the attack dataset. This number goes up to 61% after the third epoch. Overall, extractability increases at each fine-tuning epoch.

Attacking different model sizes, allowed us to observe that smaller models memorize more fine-tuning data. As shown in Figure 10, this behavior happened only from the second fine-tuning epoch. Attacking the fine-tuned three billion model we were able to extract 36.8% of additional samples. This number decreased to 30.9% attacking the fine-tuned fifteen billion model.

Kandpal et al. ; Carlini et al. revealed that the duplication rate severely impacts extractability (Kandpal et al. 2022; Carlini et al. 2022b). This result also persists in fine-tuning memorization. For StarCoder2-3B, extractability is below 5% with unique samples (i.e., $d = 1$). This percentage rises to 23.7% with a single duplicate in the fine-tuning corpus and further to 45.1% with two duplicates (i.e., $d = 3$).

Performing the fine-tuning code attack on the pre-trained models revealed that some fine-tuning samples were also extractable from the pre-trained models. Several factors may contribute to this observation.

Notably, it indicates that even before fine-tuning, the pre-trained models had already memorized portions of the fine-tuning dataset, despite the extensive filtering process applied to the fine-tuning data (Katzy et al. 2025). The repetitive and highly structured nature of Java syntax, which adheres to specific patterns that enhance readability, maintainability, and consistency, could be a significant factor. Additionally, our approach to constructing the attack dataset did not account for code complexity, leading to some suffixes being easily inferred by the model.

To further investigate, we inspected the types of code extractable from the pre-trained models. We found that the majority of the fine-tuning samples extractable from the pre-trained models consisted of licenses. Specifically, performing the fine-tuning code attack on the StarCoder2-3B base model, we extracted 120 samples out of 917, of which 107 were licenses.

Our pass@1 evaluation on HumanEval-X confirms that fine-tuning effectively improves the models' code completion capabilities across all sizes. The improvement in functional performance co-occurs with the increase in fine-tuning data extractability, indicating that memorization is not merely a symptom of overfitting but accompanies genuine task adaptation. In particular, StarCoder2-3B exhibits both the largest relative improvement in pass@1 and the highest increase in fine-tuning data extractability after three epochs. This suggests that smaller models, while benefiting most from fine-tuning in terms of task performance, are also the most exposed to data leakage.

Closed-source LLMs may also be susceptible to similar memorization and extraction risks. However, assessing this empirically is challenging because the training data, preprocessing pipelines, and deduplication procedures of proprietary models are typically not publicly disclosed. While it is possible that such systems employ additional filtering or data curation mechanisms, there is currently no public evidence confirming that these practices systematically eliminate memorization or leakage risks. As a result, the extent to which the patterns observed in open-source models transfer to closed-source systems remains uncertain due to the limited transparency surrounding their training processes.

Prior studies on code memorization have focused predominantly on Python (Al-Kaswan et al. 2024; Yang et al. 2024; Pappu et al. 2024), with those authors explicitly recommending that more syntactically structured languages be investigated as future work. Our Java findings are broadly consistent with prior Python results at the category level: data carriers and license information remain the most memorized categories across both languages, suggesting that memorization of these types is driven by duplication and structural regularity rather than language-specific syntax. This is unsurprising given that Java's greater verbosity and boilerplate produce highly repetitive token sequences that are inherently easier to memorize.

8.4 The Impact of Model Compression Techniques on Memorization

Our experiments show that model compression has only a limited influence on memorization patterns in StarCoder2-15B. Across both the pre-training and fine-tuning code attacks, bfloat16 and int8 quantization behave almost identically to the fp32 baseline. These settings

preserve the same trend, suggesting that reducing precision to these formats does not alter memorization levels.

The int4 configuration, by contrast, departs from this stability. In the pre-training code attack, int4 consistently reduces raw memorization rates compared to the fp32 baseline, across all prefix lengths. After fine-tuning, however, the effect reverses in relative terms. While int4 starts from lower absolute levels, it exhibits stronger proportional increases during training, with higher EM ↑% values than the baseline across epochs, duplication rates, and prefix lengths. This suggests that int4 suppresses pre-existing memorization but amplifies the gains that arise as new information is acknowledged during fine-tuning.

Despite these quantitative differences, the qualitative distribution of memorized categories remains unaffected by compression. In both attack settings, licenses and data carriers consistently dominate the recovered samples, with only minor numerical fluctuations across formats. This invariance highlights that int4 quantization influences the intensity of memorization but not its underlying structure.

The amplified fine-tuning leakage under int4 quantization can be understood through the lens of model capacity. Precision reduction reduces the model's capacity to retain pre-training knowledge, lowering baseline memorization. However, during fine-tuning, the reduced weight space forces the model to encode frequently seen sequences more rigidly than a full-precision model would. This explains why int4 suppresses pre-training memorization but disproportionately amplifies relative fine-tuning leakage, particularly for highly duplicated samples.

These findings suggest that precision reduction through bfloat16 and int8 is effectively transparent with respect to memorization, while int4 introduces a distinct dynamic. Weaker retention of baseline knowledge, but a stronger tendency to memorize additional fine-tuning data. From a practical standpoint, this implies that aggressive quantization may offer partial mitigation against residual memorization, but at the cost of amplifying the extractability of fine-tuning data.

8.5 The Impact of PEFT Techniques on Memorization

Our investigation into PEFT and in particular LoRA seems to suggest that the rate of memorization and forgetting is not as pronounced as with full fine-tuning.

LoRA models forget less of their training data and do not suffer from the catastrophic forgetting behaviour shown by StarCoder2-7b. LoRA models also do not memorize as much of the training data as full fine-tuned models. Quite the opposite, they tend to forget a bit of the fine-tuning data, as their recall of fine-tuning data is lower than the base rate of full fine-tuning.

Our analysis reveals that LoRA-tuned models exhibit markedly different sensitivity to prefix length variations compared to fully fine-tuned models, with important implications for both memorization patterns and deployment security. For pre-training data extraction, LoRA models demonstrate heightened sensitivity to prefix length. This suggests that LoRA's parameter-efficient approach, which only updates low-rank matrices while keeping the majority of pre-trained weights frozen, preserves much of the original model's attention patterns and contextual dependencies. Conversely, for fine-tuning data extraction, LoRA models show remarkably reduced sensitivity to prefix length.

When considering the type of memorized data, LoRA models almost exclusively memorize licence information. While the rest of the data types show little to no memorization behavior.

The smoother forgetting and reduced fine-tuning memorization observed in LoRA follow from its architectural constraints. Because the pre-trained weights are kept entirely frozen and only low-rank perturbations are introduced, the attention patterns learned during pre-training are largely preserved, producing gradual rather than catastrophic forgetting across epochs. The same constraint limits the model's capacity to consolidate new fine-tuning sequences into its representations, explaining why fine-tuning extraction rates remain below the full fine-tuning baseline throughout training. This characteristic of LoRA may inadvertently provide a degree of protection against fine-tuning data extraction attacks, particularly those that rely on extended contextual cues.

This suggests that parameter-efficient fine-tuning techniques like LoRA may provide inherent protection against fine-tuning data extraction attacks while preserving more stable access to pre-training knowledge.

8.6 Cross-Architecture Generalizability

Our experiments on Mellum-4B confirm that the qualitative memorization patterns observed in StarCoder2 are not architecture-specific. Across both model families, fine-tuning data extractability increases with training epochs, duplication rate, and input prefix length, while fine-tuning reduces the extractability of pre-training data. These trends hold despite fundamental architectural differences between the two models: StarCoder2 uses Grouped Query Attention with a Fill-in-the-Middle training objective, whereas Mellum adopts a LLaMA-style decoder-only design optimized for autoregressive code completion.

However, the magnitude of memorization differs substantially. After three epochs of fine-tuning under the same conditions, StarCoder2-3B reaches an EM ↑% of 36.8, while Mellum-4B achieves only 5.0. This gap is consistent across all experimental dimensions: duplication rate, prefix length, and code categories. Several factors may contribute to this difference.

First, the two models differ in their pre-training data curation. Mellum is trained exclusively on permissively licensed code, which may result in a less duplicated and more carefully curated training corpus. Since duplication is a well-established driver of memorization (Kandpal et al. 2022; Carlini et al. 2022b), a cleaner pre-training corpus could reduce the model's overall tendency to memorize. Second, architectural differences may play a role. The LLaMA-style architecture used by Mellum differs from StarCoder2 in attention mechanisms and training objectives, which could affect how strongly the model encodes specific sequences during fine-tuning.

Regarding code categories, the low number of extracted samples from Mellum-4B limits the strength of category-level conclusions. Nevertheless, license information is the most extracted category in both architectures, which is consistent with the highly repetitive and standardized nature of license headers across repositories. Other categories, such as data carriers, show divergent behavior between the two models, but whether this reflects architectural differences or is an artifact of the small sample size remains an open question.

While our analysis identifies these architectural and data-level differences as plausible explanations, disentangling their individual contributions would require controlled experiments that vary architecture and training data independently.

8.7 Implications

The implications of memorization and extractability are strongly dependent on the deployment context, threat model, and sensitivity of the fine-tuning data. In particular, the practical risk of leakage may be lower in tightly controlled enterprise settings where models are used internally and access is restricted. At the same time, even in controlled enterprise deployments, memorization may remain relevant because it is related to overfitting to fine-tuning data, and the effects of such overfitting may become more visible if the model is later applied to different tasks, users, or data distributions (Yeom et al. 2018; Hartmann et al. 2023).

Risks for Fine-tuning Models: Our findings suggest that fine-tuning on code data can introduce privacy and leakage risks, particularly when the fine-tuning set contains proprietary or sensitive information. However, the significance of these risks depends on the intended use of the model. In enterprise settings, stronger adaptation to internal codebases may be beneficial for utility, while the practical leakage risk may be reduced when the model is deployed only under restricted internal access. Smaller models are more prone to memorizing fine-tuning data compared to larger models (Al-Kaswan et al. 2024). Although they are cheaper and faster to fine-tune, they carry a higher risk of data leakage. Therefore, model size should be considered as one factor in the privacy–utility trade-off when fine-tuning code models on personal or proprietary data

Furthermore, our results show that parameter-efficient methods such as LoRA reduce the degree of fine-tuning data memorization compared to full fine-tuning. At the same time, LoRA exhibits a smoother decay of pre-training memorized knowledge, allowing for better retention of previously memorized information compared to full fine-tuning. This suggests that LoRA may offer a fair balance between reducing fine-tuning leakage and preserving general pre-training knowledge.

Security Vulnerability Propagation: LLMs may generate code snippets containing security flaws present in their training data, potentially introducing vulnerabilities into new software projects (Al-Kaswan et al. 2026; Jesse et al. 2023). This could lead to the unintentional spread of security issues across codebases (Al-Kaswan and Izadi 2023; Al-Kaswan et al. 2026; Yang et al. 2024).

Data Privacy Leakage Concern: If models output verbatim snippets from proprietary or sensitive data used in fine-tuning, it could lead to unintended disclosure of confidential information. This is particularly problematic for industries dealing with personal, medical, or financial data. This also raises ethical questions about consent, especially if the data involves personal information or creative works used without explicit permission for such reproduction (Henderson et al. 2023).

We also observe that precision reduction at deployment has privacy implications. While bfloat16 and int8 quantization preserve memorization behavior similar to the full-precision setting, int4 quantization amplifies the leakage of fine-tuning data. This finding suggests that aggressive compression may inadvertently heighten privacy risks when proprietary data is involved in fine-tuning.

Legal and IP Issues: Many existing LLMs for code utilize copyleft and other non-permissive licensed code (Al-Kaswan and Izadi 2023). Using public code to train these models can fall under fair use, which allows copyrighted works to be used in new and transformative ways in many jurisdictions (Henderson et al. 2023). However, if the model's output closely resembles the copyrighted input, fair use may no longer apply. In such cases, the output must adhere to the original license terms, which may include share-alike and attribution requirements (Henderson et al. 2023). Outputting memorized samples from proprietary datasets could potentially infringe on intellectual property rights. This raises significant legal concerns, especially if the model is used commercially or made publicly accessible. Ensuring compliance with IP laws is crucial to avoid litigation and uphold ethical standards in model deployment (Choksi and Goedicke 2023).

Overfitting and Reduced Generalization: If the model memorizes specific examples from the fine-tuning data instead of learning general patterns, it may perform poorly on new, unseen data. This overfitting can limit the model's effectiveness in real-world applications (Hartmann et al. 2023; Carlini et al. 2021; Al-Kaswan et al. 2024; Yeom et al. 2018).

Our results also suggest that LoRA adaptation, while less prone to memorization, may also generalize better by avoiding overfitting to specific fine-tuning samples. In contrast, compression methods such as int4 amplify relative memorization of fine-tuning data, which may further reduce generalization capacity.

8.8 Recommendations

To mitigate these risks, organizations that fine-tune foundation models on specialized or proprietary data need to implement robust safeguards. From a technical perspective, implementing differential privacy techniques like Whispered Tuning (Singh et al. 2024) can help ensure that the model does not memorize specific training data by adding noise during the training process. Additionally, using data filtering and deduplication methods can remove sensitive information and prevent the inclusion of duplicate data entries, thereby reducing the risk of memorization (Kandpal et al. 2022; Lee et al. 2022; Carlini et al. 2022b). When fine-tuning models, it is important to limit the number of epochs, employ regularization techniques, and prefer larger models to prevent overfitting and memorization.

When selecting deployment strategies, organizations should also account for how compression and parameter-efficient adaptation affect memorization. Our results show that bfloat16 and int8 quantization preserve memorization patterns similar to full precision. By contrast, int4 quantization amplifies fine-tuning data leakage, suggesting that aggressive compression should be avoided when sensitive data is involved. Parameter-efficient approaches such as LoRA offer a promising alternative, as they both reduce the memorization of fine-tuning data and better retain general pre-training knowledge compared to full fine-tuning.

Organizational practices are also vital in ensuring the secure and ethical deployment of LLMs. Active monitoring of model outputs through automated checks is essential to detect and address any instances where the model might reproduce memorized data. Developing methods to "forget" specific pieces of information post-training, such as unlearning techniques, can further safeguard against data leakage (Al-Kaswan et al. 2024; Yang et al. 2024).

Memorization can put creators and users of LLMs for code at legal risk, including license and fair use infringements (Al-Kaswan and Izadi 2023; Henderson et al. 2023). To mitigate

these risks, it is crucial to train models using code licensed under permissive licenses (such as BSD-3 or MIT) or to provide provenance information that traces the code back to its source (Al-Kaswan and Izadi 2023; Al-Kaswan et al. 2024; Yang et al. 2024; Henderson et al. 2023).

8.9 Ethical Considerations

Although this work explores techniques that could potentially be used to extract sensitive information from models, it is conducted with ethical considerations in mind. Our objective is to highlight the issue of memorization in LLMs for code, inform users and creators about this problem, and provide tools for measuring it. We emphasize that our work does not intentionally expose private information and we strongly urge users of our work to avoid such practices. We believe the benefits of sharing our data outweigh the risks, and we decide to make them available.

8.10 Limitations and Threats to the Validity

8.10.1 Internal Validity

In our evaluation, we did not account for the spatial distribution of samples within the files. While the samples maintain a fixed token length, their locations within the files are selected arbitrarily to avoid bias. The methodology was chosen to ensure samples could be extracted from any location within the file to reflect a realistic and varied scenario (Al-Kaswan et al. 2024; Carlini et al. 2022b).

8.10.2 External Validity

While we study three model sizes, our evaluation focuses on the StarCoder2 model family. Other model architectures may exhibit different levels of memorization. Additionally, our pre-training code dataset considers samples that are duplicated more than three times in the training corpus. Due to computational constraints, we were not able to control the duplication rate as we did on the fine-tuning code attack dataset. More studies on pre-training code attacks on low-duplication data could extend our findings.

Most of the previous studies have focused on Python. This leaves a gap for other programming languages. We focused on Java for our attack datasets due to its well-defined syntax and structure, which may impact memorization. Given Java's popularity, our results may not apply to less common languages. In future evaluations, we plan to include more programming languages to compare how different languages are memorized by the model.

Our experiments use a single fine-tuning dataset: a 20k-file Java subset from The Heap as it is filtered against The StackV2 (Lozhkov et al. 2024). Its specific properties, such as language, size, duplication patterns, licenses, and filtering, may affect memorization behavior. As a result, our findings may not generalize to other fine-tuning datasets, tasks, or programming languages. Future work should replicate the study on additional datasets with different characteristics to assess generalizability.

8.10.3 Construct Validity

We primarily use the EM metric for reporting extractability rates in code LMs as we focus on exact reproductions to address privacy and security concerns associated with memorization. However, this metric likely underestimates the true extent of memorized samples, as some might be slightly altered by the model (Hartmann et al. 2023). To mitigate this threat, we also compute and report other fuzzy metrics such as BLEU, METEOR, and ROUGE-L in the paper and in our replication package (Salerno et al. 2025). Our findings indicate that these metrics are highly correlated with the EM rate.

9 Conclusion

In this study, we conducted an extensive investigation into data extraction from pre-trained and fine-tuned code language models. We formally defined a data extraction security game based on the concepts of k-extractability and membership inference attacks. This framework was employed to develop a custom benchmark for assessing the vulnerability of both pre-training and fine-tuning samples to extraction attacks. Our findings indicate that the StarCoder2 model memorizes its pre-training data. We further observed that fine-tuning reduces the extractability of pre-training data. Contrarily, we found that fine-tuning increases the vulnerability of smaller models to fine-tuning data extraction attacks compared to larger models. Additionally, our results show that data carriers and license information are the most likely to be memorized, with the latter being the most likely to be forgotten after fine-tuning

We further examined memorization, considering different model precision settings. Our results show that bfloat16 and int8 quantization largely preserve memorization behavior comparable to the full-precision fp32 baseline, making them relatively safe deployment defaults. In contrast, int4 quantization reduces the extractability of pre-training data but amplifies leakage of fine-tuning data, highlighting that aggressive compression may inadvertently increase privacy risks when proprietary datasets are used.

PEFT exhibits more gradual forgetting of pre-training data without the catastrophic drops seen in full fine-tuning, while showing reduced capacity to memorize fine-tuning data compared to full fine-tuning. These characteristics suggest that parameter-efficient fine-tuning techniques like LoRA may provide inherent protection against fine-tuning data leakage while maintaining more stable access to pre-training data. These insights highlight that memorization risks in code language models extend beyond the training stage to deployment strategies such as fine-tuning, compression, and parameter-efficient adaptation. Addressing these risks is essential for safeguarding sensitive data and highlight the need for further research and development of strategies to mitigate these vulnerabilities.

Author Contributions Fabio Salerno, Ali Al-kaswan: Conceptualization, methodology, implementation, writing, original draft, and editing. Maliheh Izadi: Conceptualization, methodology, writing, original draft, and review.

Funding The authors gratefully acknowledge the financial support provided by the Amazon Research Award granted to Maliheh Izadi for research on Memorisation in LLMs for code.

Data Availability Our code and data are available in our replication package (Salerno et al. 2025).

Declarations

Ethical Approval Not applicable.

Informed consent Not applicable.

Conflict of interest The authors declare that they have no conflict of interest.

Clinical Trial Number Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Ahmed T, Devanbu P (2022) Few-shot training llms for project-specific code-summarization. Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. pp 1–5

Al-Kaswan A, Izadi M (2023) The (ab)use of open source code to train large language models. Proceedings of the 2nd International Workshop on Natural Language-based Software Engineering (NLBSE)

Al-Kaswan A, Ahmed T, Izadi M, Sawant AA, Devanbu P, van Deursen A (2023a) Extending source code pre-trained language models to summarise decompiled binaries. In: Proceedings of the 30th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)

Al-Kaswan A, Izadi M, van Deursen A (2023b) Targeted attack on gpt-neo for the satml language model data extraction challenge. ArXiv abs/2302.07735

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

Al-Kaswan A, Spiess C, Devanbu P, van Deursen A, Izadi M (2026) Model see, model do? exposure-aware evaluation of bug-vs-fix preference in code llms. In: 2026 IEEE/ACM 23nd International Conference on Mining Software Repositories (MSR), ACM

Allal LB, Li R, Kocetkov D, Mou C, Akiki C, Ferrandis CM, Muennighoff N, Mishra M, Gu A, Dey M (2023) Santacoder: don't reach for the stars! In: Deep Learning for Code (DL4C) Workshop

Banerjee S, Lavie A (2005) 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, Association for Computational Linguistics, Ann Arbor, Michigan, pp 65–72, https://www.aclweb.org/anthology/W05-0909

Biderman S, Prashanth U, Sutawika L, Schoelkopf H, Anthony Q, Purohit S, Raff E (2023) Emergent and predictable memorization in large language models. Adv Neural Inf Process Syst 36:28072–28090

Bogomolov E, Zhuravlev S, Spirin E, Bryksin T (2022) Assessing project-level fine-tuning of ml4se models. arXiv:2206.03333 arXiv preprint

Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A et al (2020) Language models are few-shot learners. Adv Neural Inf Process Syst 33:1877–1901

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

Carlini N, Chien S, Nasr M, Song S, Terzis A, Tramer F (2022a) Membership inference attacks from first principles. In: 2022 IEEE Symposium on Security and Privacy (SP), IEEE, pp 1897–1914

Carlini N, Ippolito D, Jagielski M, Lee K, Tramer F, Zhang C (2022b) Quantifying memorization across neural language models. In: The Eleventh International Conference on Learning Representations

Carlini N, Jagielski M, Zhang C, Papernot N, Terzis A, Tramer F (2022) The privacy onion effect: Memorization is relative. Adv Neural Inf Process Syst 35:13263–13276

Chen M, Tworek J, Jun H, Yuan Q, Pinto H, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G (2021) Evaluating large language models trained on code.arXiv:2107.03374 arXiv preprint

Choksi MZ, Goedicke D (2023) Whose text is it anyway? exploring bigcode, intellectual property, and ethics. https://arxiv.org/abs/2304.02839, 2304.02839

van Dam T, Izadi M, van Deursen A (2023) Enriching source code with contextual data for code completion models: An empirical study. In: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), IEEE, pp 170–182

Dettmers T, Pagnoni A, Holtzman A, Zettlemoyer L (2023) Qlora: Efficient finetuning of quantized llms. Adv Neural Inf Process Syst 36:10088–10115

Devlin J, Chang MW, Lee K, Toutanova K (2019) 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, Volume 1 (Long and Short Papers), pp 4171–4186

Delft High Performance Computing Centre (DHPC) (2024) DelftBlue Supercomputer (Phase 2). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2

Egashira K, Vero M, Staab R, He J, Vechev M (2024) Exploiting llm quantization. Adv Neural Inf Process Syst 37:41709–41732

Fabio Salerno, Ali Al-Kaswan, Maliheh Izadi (2025) How do deployment strategies influence extractability in code language models? https://doi.org/10.5281/zenodo.19453254, https://doi.org/10.5281/zenodo.19453254

Han Z, Gao C, Liu J, Zhang J, Zhang SQ (2024) Parameter-efficient fine-tuning for large models: A comprehensive survey. CoRR

Hartmann V, Suri A, Bindschaedler V, Evans D, Tople S, West R (2023) Sok: Memorization in general-purpose large language models. https://arxiv.org/abs/2310.18362, 2310.18362

Henderson P, Li X, Jurafsky D, Hashimoto T, Lemley MA, Liang P (2023) Foundation models and fair use. J Mach Learn Res 24(400):1–79

Hou X, Zhao Y, Liu Y, Yang Z, Wang K, Li L, Luo X, Lo D, Grundy J, Wang H (2024) Large language models for software engineering: A systematic literature review. ACM Transactions on Software Engineering and Methodology 33(8):1–79

Hu EJ, Shen Y, Wallis P, Allen-Zhu Z, Li Y, Wang S, Wang L, Chen W et al (2022) Lora: Low-rank adaptation of large language models. ICLR 1(2):3

Hu H, Salcic Z, Sun L, Dobbie G, Yu PS, Zhang X (2022) Membership inference attacks on machine learning: A survey. ACM Computing Surveys (CSUR) 54(11s):1–37

Huang J, Shao H, Chang KCC (2022) Are large pre-trained language models leaking your personal information? In: Findings of the Association for Computational Linguistics: EMNLP 2022, Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, pp 2038–2047, https://aclanthology.org/2022.findings-emnlp.148

Huang Y, Li Y, Wu W, Zhang J, Lyu MR (2024) Your code secret belongs to me: Neural code completion tools can memorize hard-coded credentials. Proceedings of the ACM on Software Engineering 1(FSE):2515–2537

Ippolito D, Tramer F, Nasr M, Zhang C, Jagielski M, Lee K, Choquette Choo C, Carlini N (2023) Preventing generation of verbatim memorization in language models gives a false sense of privacy. In: Proceedings of the 16th International Natural Language Generation Conference, Association for Computational Linguistics, Prague, Czechia, https://doi.org/10.18653/v1/2023.inlg-main.3, https://aclanthology.org/2023.inlg-main.3

Ishihara S (2023) Training data extraction from pre-trained language models: A survey. In: The Third Workshop on Trustworthy Natural Language Processing, p 260

Izadi M, Katzy J, Van Dam T, Otten M, Popescu RM Van Deursen A (2024) Language models for code completion: A practical evaluation. Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp 1–13

Jesse K, Ahmed T, Devanbu PT, Morgan E (2023) Large language models and simple, stupid bugs. 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR). IEEE, pp 563–575

Kandpal N, Wallace E, Raffel C (2022) Deduplicating training data mitigates privacy risks in language models. International Conference on Machine Learning. pp 10697–10707 (**PMLR**)

Katzy J, Popescu R, Van Deursen A, Izadi M (2024) An exploratory investigation into code license infringements in large language model training datasets. Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering. pp 74–85

Katzy J, Popescu RM, van Deursen A, Izadi M (2025) The heap: A contamination-free multilingual code dataset for evaluating large language models. In: 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge), IEEE, pp 151–155

Lee K, Ippolito D, Nystrom A, Zhang C, Eck D, Callison-Burch C, Carlini N (2022) Deduplicating training data makes language models better. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp 8424–8445

Lin CY (2004) ROUGE: A package for automatic evaluation of summaries. In: Text Summarization Branches Out, Association for Computational Linguistics, Barcelona, Spain, pp 74–81, https://www.aclweb.org/anthology/W04-1013

Liu H, Tam D, Muqeeth M, Mohta J, Huang T, Bansal M, Raffel CA (2022) Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. In: Koyejo S, Mohamed S, Agarwal A, Belgrave D, Cho K, Oh A (eds) Advances in Neural Information Processing Systems, Curran Associates, Inc., vol 35, pp 1950–1965, https://proceedings.neurips.cc/paper_files/paper/2022/file/0cde695b83bd186c1fd456302888454c-Paper-Conference.pdf

Liu J, Xia CS, Wang Y, Zhang L (2023) Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In: Thirty-seventh Conference on Neural Information Processing Systems, https://openreview.net/forum?id=1qvx610Cu7

Lozhkov A, Li R, Allal LB, Cassano F, Lamy-Poirier J, Tazi N, Tang A, Pykhtar D, Liu J, Wei Y, Liu T, Tian M, Kocetkov D, Zucker A, Belkada Y, Wang Z, Liu Q, Abulkhanov D, Paul I, Li Z, Li WD, Risdal M, Li J, Zhu J, Zhuo TY, Zheltonozhskii E, Dade NOO, Yu W, Krauß L, Jain N, Su Y, He X, Dey M, Abati E, Chai Y, Muennighoff N, Tang X, Oblokulov M, Akiki C, Marone M, Mou C, Mishra M, Gu A, Hui B, Dao T, Zebaze A, Dehaene O, Patry N, Xu C, McAuley J, Hu H, Scholak T, Paquet S, Robinson J, Anderson CJ, Chapados N, Patwary M, Tajbakhsh N, Jernite Y, Ferrandis CM, Zhang L, Hughes S, Wolf T, Guha A, von Werra L, de Vries H (2024) Starcoder 2 and the stack v2: The next generation. https://arxiv.org/abs/2402.19173, 2402.19173

Lu S, Guo D, Ren S, Huang J, Svyatkovskiy A, Blanco A, Clement C, Drain D, Jiang D, Tang D (2021) Codexglue: A machine learning benchmark dataset for code understanding and generation. Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track.(Round 1)

Mireshghallah F, Goyal K, Uniyal A, Berg-Kirkpatrick T, Shokri R (2022a) Quantifying privacy risks of masked language models using membership inference attacks. In: Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, pp 8332–8347, https://aclanthology.org/2022.emnlp-main.570

Mireshghallah F, Uniyal A, Wang T, Evans D, Berg-Kirkpatrick T (2022b) Memorization in nlp fine-tuning methods. In: First Workshop on Pre-training: Perspectives, Pitfalls, and Paths Forward at ICML 2022

Muennighoff N, Liu Q, Zebaze A, Zheng Q, Hui B, Zhuo TY, Singh S, Tang X, Werra V, Longpre L, S (2023) Octopack: Instruction tuning code large language models. NeurIPS 2023 workshop on instruction tuning and instruction following

Oh MG, Park LH, Kim J, Park J, Kwon T (2023) Membership inference attacks with token-level deduplication on korean language models. IEEE Access 11:10207–10217

Pappu A, Porter B, Shumailov I, Hayes J (2024) Measuring memorization in rlhf for code completion. arXiv:2406.11715 arXiv preprint

Pavlichenko N, Nazarov I, Dolgov I, Garanina E, Ustalov D, Bondyrev I, Lysaniuk K, Vu E, Chekmenev K, Shtok J (2025) Mellum: Production-grade in-ide contextual code completion with multi-file project understanding.arXiv:2510.05788 arXiv preprint

Post M (2018) A call for clarity in reporting BLEU scores. In: Proceedings of the Third Conference on Machine Translation: Research Papers, Association for Computational Linguistics, Belgium, Brussels, pp 186–191, https://www.aclweb.org/anthology/W18-6319

Shazeer N, Stern M (2018) Adafactor: Adaptive learning rates with sublinear memory cost. International Conference on Machine Learning. pp 4596–4604 (**PMLR**)

Shokri R, Stronati M, Song C, Shmatikov V (2017) Membership inference attacks against machine learning models. 2017 IEEE symposium on security and privacy (SP). IEEE, pp 3–18

Singh T, Aditya H, Madisetti VK, Bahga A (2024) Whispered tuning: Data privacy preservation in fine-tuning llms through differential privacy. J Softw Eng Appl 17(1):1–22

Sun AY, Zemour E, Saxena A, Vaidyanathan U, Lin E, Lau C, Mugunthan V (2023) Does fine-tuning gpt-3 with the openai api leak personally-identifiable information? arXiv preprint.arXiv:2307.16382

Wang Y, Wang W, Joty S, Hoi SC (2021) CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, pp 8696–8708, https://doi.org/10.18653/v1/2021.emnlp-main.685, https://aclanthology.org/2021.emnlp-main.685

Wei X, Gonugondla SK, Wang S, Ahmad W, Ray B, Qian H, Li X, Kumar V, Wang Z, Tian Y, et al. (2023) Towards greener yet powerful code generation via quantization: An empirical study. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 224–236

Weyssow M, Zhou X, Kim K, Lo D, Sahraoui H (2025) Exploring parameter-efficient fine-tuning techniques for code generation with large language models. ACM Transactions on Software Engineering and Methodology 34(7):1–25

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

Yeom S, Giacomelli I, Fredrikson M, Jha S (2018) Privacy risk in machine learning: Analyzing the connection to overfitting. IEEE 31st computer security foundations symposium (CSF). IEEE, pp 268–282

Zeng S, Li Y, Ren J, Liu Y, Xu H, He P, Xing Y, Wang S, Tang J, Yin D (2024) Exploring memorization in fine-tuned language models. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp 3917–3948

Zhang S, Dong L, Li X, Zhang S, Sun X, Wang S, Li J, Hu R, Zhang T, Wu F, et al. (2023) Instruction tuning for large language models: A survey. arXiv preprint arXiv:2308.10792

Zheng Q, Xia X, Zou X, Dong Y, Wang S, Xue Y, Shen L, Wang Z, Wang A, Li Y, et al. (2023) Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In: Proceedings of the 29th ACM SIGKDD conference on knowledge discovery and data mining, pp 5673–5684

Zhuo TY, Zebaze A, Suppattarachai N, von Werra L, de Vries H, Liu Q, Muennighoff N (2024) Astraios: Parameter-efficient instruction tuning code large language models. CoRR

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.