

Understanding Privacy Risks in Code Models Through Training Dynamics: A Causal Approach

Hua Yang* Alejandro Velasco† Sen Fang* Bowen Xu* Denys Poshyvanyk†

December 9, 2025

Abstract

Large Language Models for Code (*LLM4Code*) have significantly enhanced developer productivity in real-world software engineering. However, their reliance on open-source repositories (e.g., GitHub) introduces severe privacy risks, as these repositories contain abundant Personally Identifiable Information (*PII*), such as *Key*, *Username*, *Email*, *Password*, *Name*, and *IP Address*. Sensitive *PII* can be memorized during training and later resurfaced during inference, leading to critical privacy breaches. Prior studies have shown that commercial code completion models are indeed capable of reproducing sensitive *PII*, further exacerbating concerns about the trustworthiness and compliance of *LLM4Code* in sensitive industries. Nonetheless, existing work has largely treated *PII* as a homogeneous category, overlooking the heterogeneous risks posed by different *PII* types.

To fill this gap, we investigate whether different types of *PII* vary in their likelihood of being learned and leaked by *LLM4Code*. More specifically, we examine whether the training dynamics of *LLM4Code* on each type influence its leakage risk at inference time, and whether this relationship is causal. Our methodology consists of four stages: (i) constructing a high-quality dataset with a diverse set of *PII* types sourced from real-world software repositories; (ii) fine-tuning representative *LLM4Code* of different scales and architectures to capture cross-family patterns; (iii) computing training dynamics on real *PII* data and (iv) formulating a structural causal model to estimate the causal effect of learning difficulty on leakage, validated through multiple refutation strategies.

Our experimental results demonstrate that leakage risks vary significantly across *PII* types and are closely tied to their training dynamics. Easy-to-learn instances (e.g., *IP Address*) exhibit high leakage risks, whereas hard-to-learn instances (e.g., *Key* and *Password*) leak less frequently. Ambiguous types display more complex dual effects. This study provides the first causal evidence that leakage risks differ by *PII* type, offering actionable insights for designing type-aware and learnability-aware defense mechanisms in *LLM4Code*.

1 Introduction

The future of software engineering is evolving towards a paradigm where tasks such as code completion, code summarization, and code generation are heavily reliant on *Large Language Models for Code (LLM4Code)*. *LLM4Code* have demonstrated strong capabilities in real-world development, significantly improving productivity. However, their success is based on massive training corpora drawn from open-source repositories such as GitHub [GitHub Inc.([n. d.]), Lozhkov et al.(2024), Roziere et al.(2023)].

Although these open-source repositories provide abundant training material, they also introduce serious privacy risks due to the strong memorization abilities of *LLM4Code*. GitHub contains large amounts of *Personally Identifiable Information (PII)*, including *API keys*, *usernames*, *email addresses*, *passwords*, *personal names*, and *IP addresses* [Li et al.([n. d.]), Huang et al.(2024), Krause et al.(2023)]. When incorporated into training, this sensitive information can persist in a model’s parameters and later resurface at inference time

*North Carolina State University

†William & Mary

[Carlini et al.(2022), Yang et al.(2024)]. Recently, this problem has also surfaced in industry contexts. For example, Microsoft’s AI research team accidentally exposed an additional 38 TB of private data when publishing an open source training dataset on GitHub. The leak included backups of two employees’ workstations containing confidential information, private keys, passwords, and more than 30,000 internal Teams messages [Wiz(2023)].

Such incidents highlight the growing tension between the utility of *LLM4Code* and the regulatory landscape. Frameworks such as the General Data Protection Regulation (GDPR) and the Health Insurance Portability and Accountability Act (HIPAA) impose strict requirements on handling *PII*, yet prior studies [Niu et al.(2023), Huang et al.(2024)] confirm that commercial code completion models can reproduce sensitive information. This raises serious concerns about the trustworthiness and scalability of *LLM4Code* in critical industries where privacy guarantees are not negligible.

Despite these findings, there are still important research gaps. Most prior work treats *PII* as a single undifferentiated type, overlooking the fact that different types of *PII* pose very different risks. The impact of a leak is highly dependent on what is leaked [Krause et al.(2023)]. High-risk data, such as API keys or passwords, can directly compromise systems, as in the Toyota incident [McDaniel(2022)], where a hard coded credential to access a data server was publicly pushed to GitHub and remained exposed for five years, granting attackers access to the data of more than 290,000 customers. In contrast, medium- or low-risk information, such as developer names or email addresses, may not cause immediate damage, but can still enable identity inference, privacy violations, or social engineering attacks. Addressing these issues requires going beyond categorizing leaks to explain how different types of *PII* are learned and propagated by *LLM4Code*.

We address this gap by asking whether different types of *PII* vary in their probability of being learned and subsequently leaked by *LLM4Code*. More specifically, we examine whether the ease with which a model acquires a given *PII* type during training influences its leakage risk during inference, and whether this relationship is causal. We pursue these issues in three connected dimensions: *the relative learning difficulty of different PII types, how this difficulty translates into leakage under attack, and whether training dynamics directly shape leakage behavior*.

Our methodology proceeds in four stages. First, during dataset construction, we scan and extract code snippets containing different types of *PII* from real-world LLM training corpora. We design an automated collection and refinement workflow that integrates regex-based filters, LLM-assisted judgment, and heuristic rules, and we also conduct human verification to ensure accuracy and reliability. Second, in the model setup, we fine-tune representative *LLM4Code* of varying scales and architectures to capture patterns that generalize across families of models. Third, we compute the training dynamics [Swayamdipta et al.(2020)] of *LLM4Code* on real *PII* data to measure the relative learning difficulty of each type. Finally, we formulate a structural causal model (*SCM*) to estimate the effect of learning difficulty on leakage and validate robustness through multiple refutation strategies.

The results reveal that the probability of *PII* being leaked is highly dependent on both its type and its training dynamics. Easy-to-learn instances, such as *IP Address* instances, exhibit high leakage risks, whereas hard-to-learn cases, such as *Key* and *Password* instances leak less often. Ambiguous cases show more complex patterns, reducing leakage in some contexts but amplifying it in others. These findings highlight that the risk of privacy in *LLM4Code* is not uniform but is shaped by type-specific learning behavior. In summary:

- We construct a multi-type *PII* dataset from real-world code, combining automated detection, LLM refinement, and human validation to ensure high reliability¹.
- We analyze training dynamics across *PII* types, characterizing their relative difficulty in terms of confidence and variability.
- We perform systematic *PII* attack experiments to link learning dynamics with leakage behavior across model families and sizes.
- We formulate a structural causal model to quantify the causal effect of learning difficulty on leakage and validate the robustness of the results.

¹https://anonymous.4open.science/r/pii_final-42A1

- We provide the first causal evidence that the types of PII differ in their leakage risks, offering actionable information on defenses that are type- and learnability-aware in *LLM4Code*.

2 Background

2.1 PII in Code Repositories

Open-source code repositories often contain a large amount of *PII*, some of which may have severe consequences if leaked. Developers widely rely on version control systems such as GitHub, to manage their source code. Previous studies show that many developers struggle to properly manage *PII* and may unintentionally commit it to repositories [Krause et al.(2023)]. For example, in 2019, Meli et al. identified more than 100,000 instances of *PII* in public GitHub repositories [Meli et al.(2019)]. In 2022, Rahman et al. found that developers often committed *PII* even when warned by secret detection tools; their case study revealed that developers classified nearly 50% of warnings as false positives, and therefore bypassed alerts and uploaded secrets regardless [Rahman et al.(2022)]. In 2023, TruffleHog reported the discovery of 721 live API keys and passwords in a subset of GitHub’s public Pull Requests and Issue comments, demonstrating that developers were directly pasting sensitive values such as passwords and keys into public text fields [Truffle Security(2023)]. In 2024, more than 39 million API keys, credentials and other sensitive secrets were exposed on GitHub, raising considerable concern among both the developer community and global enterprises [GBHackers on Security(2024)]. Therefore, the presence of *PII* in open source repositories remains an urgent and unsolved problem, highlighting the need for more related research.

2.2 Training Dynamics

Different training samples often exhibit varying levels of learning difficulty for neural networks [Shen and Sanghavi(2019), Ethayarajh et al.(2022), Swayamdipta et al.(2020)]. Training dynamics provides an effective tool for characterizing and diagnosing individual samples within training datasets [Swayamdipta et al.(2020)]. In general, training dynamics refers to the behavior of a model in specific instances as training progresses. For example, Toneva et al. identified training examples that are frequently misclassified in later epochs despite being correctly classified earlier [Toneva et al.(2018)]. Ethayarajh et al. proposed Pointwise V-Information as a metric to quantify the difficulty of individual instances [Ethayarajh et al.(2022)].

Building on this line of work, Swayamdipta et al. introduced *Data Maps*, which leverage training dynamics to categorize training data into three distinct regions: an *ambiguous region*, where the model is uncertain about the sample; an *easy-to-learn region*, which contains the majority of samples that the model can learn reliably; and a *hard-to-learn region*, consisting of samples that the model struggles to learn [Swayamdipta et al.(2020)]. Inspired by this methodology, our study employs training dynamics to analyze different types of *PII* data in terms of their *confidence* and *variability*.

The **confidence** of a target PII token is defined as the mean probability that the model assigns to its ground-truth label throughout training epochs E :

$$\hat{\mu}_i = \frac{1}{E} \sum_{e=1}^E p_{\theta^{(e)}}(y_i^* | x_i),$$

where $p_{\theta^{(e)}}$ denotes the probability predicted by the model with parameters $\theta^{(e)}$ at the end of the e^{th} epoch, y_i^* is the true label of the token i , and x_i is its context. Intuitively, a high-confidence instance is considered easier for the learner since the model consistently assigns a high probability to the correct label.

The **variability** of a target PII token reflects the stability of its confidence over the course of training. It is measured by the standard deviation of the confidence scores of the model in epochs:

$$\hat{\sigma}_i = \sqrt{\frac{1}{E} \sum_{e=1}^E (p_{\theta^{(e)}}(y_i^* | x_i) - \hat{\mu}_i)^2}.$$

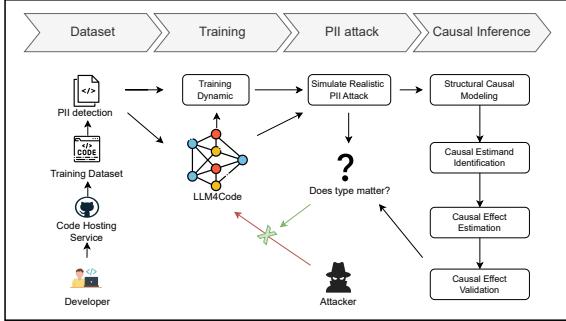


Figure 1: Overview of our study workflow. The process starts with *PII* dataset construction, followed by collection of training dynamics during fine-tuning, execution of the *PII* attack, and finally causal analysis of the link between training dynamics and attack success rate.

This enables us to determine which types of *PII* are easier or harder for *LLM4Code* to learn, providing insights that can inform both the memorization of *PII* and the design of future defense mechanisms. Lower variability indicates that the model learns the token in a stable manner across epochs, whereas higher variability suggests fluctuations and potential inconsistency in learning.

2.3 Causal Interpretability

To go beyond correlation and rigorously investigate the relationship between learning dynamics and *PII* leakage, we adopt a perspective of structural causal modeling (*SCM*). Causal interpretability, as introduced in *do_code* [Nader Palacio et al.(2024)], provides a post hoc framework grounded in Pearl’s ladder of causation [Pearl and Mackenzie(2018)] for distinguishing genuine causal effects from spurious correlations. In the context of privacy for *LLMs4Code*, we adapt this framework to analyze how the dynamics of training (*i.e.*, whether a *PII* instance is easy, ambiguous, or difficult to learn) causally influence the probability of leakage during inference.

Formally, we frame the analysis as estimating the *causal effect* of different levels of learning difficulty on the outcome of a *PII Attack*. This allows us to test whether leakage tendencies reflect intrinsic memorization behaviors or superficial correlations. Following *do_code*, our procedure consists of four steps: ① **model** the problem using causal assumptions encoded in a directed acyclic graph representing the relationship between learning dynamics, confounders, and leakage, ② **identify** the causal estimand of interest (the effect of learning difficulty on leakage), ③ **estimate** causal effects using statistical and machine learning methods, and ④ **validate** these estimates through sensitivity analyses to ensure robustness. This causal framework enables us to quantify whether the observed leakage risks are actually caused by the way *PII* is learned during training, rather than being artifacts of data set bias or model variance.

3 Study Design

This section outlines the design of our study. Our experimental workflow is illustrated in Fig. 1. The procedure begins with the construction of the dataset, where we match, refine, and manually verify the code snippets containing different types of *PII*. Next, we capture the training dynamics of *PII* during the fine-tuning of *LLM4Code* using the constructed dataset. We then perform a *PII* attack on the fine-tuned *LLM4Code* and analyze how the training dynamics of each *PII* type relate to the attack success rate. Finally, we adopt causal interpretability by formulating a structural causal model that quantifies the effect of learning

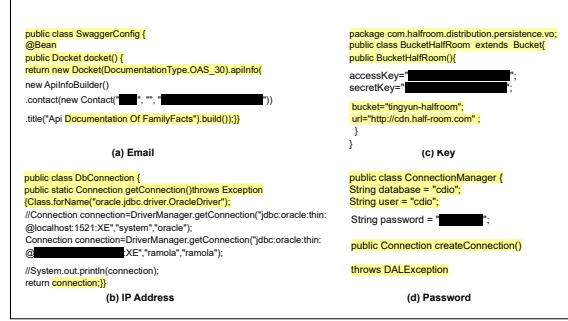


Figure 2: Examples where real *PII* values are leaked when the *LLM4Code* is queried with surrounding context. Leaked values are redacted with black boxes and query inputs are highlighted in yellow.

difficulty on *PII* leakages.

3.1 Threat Model

A well-defined threat model that specifies the attacker’s knowledge, assumptions, and capabilities is essential for any rigorous security analysis. To understand the relationship between the type of *PII* and *PII* attacks, we conduct a realistic *PII* attack under the following threat model.

We consider an adversary whose goal is to extract *PII* from a code completion model by constructing inputs that partially reproduce the surrounding context of *PII* values from open-source repositories, such as GitHub. This assumption is realistic, since training large-scale code generation models without open-source code is virtually impractical. We can always find contextual information accompanying sensitive *PII* from open source repositories [Niu et al.(2023), Huang et al.(2024)].

We assume that the attacker has only input–output access to the model. Specifically, the adversary can query the model and observe its generated completions, including the next predicted token and the log probabilities of the top-ranked tokens. However, the attacker has no access to the internal structure, parameters, or training data of the model. This setting reflects realistic deployment scenarios both commercial and open source *LLMs4Code*. For example, commercial models such as OpenAI’s GPT-4 [OpenAI and et al(2024)] expose only limited information via APIs, while open-source models such as StarCoder [Li et al.([n. d.])] and CodeLlama [Roziere et al.(2023)] are often evaluated under similar query constraints to simulate practical audit conditions. We provide actual examples in Fig. 2.

3.2 Research Questions

Building on the threat model and the study design, we frame our research around three central questions. We progressively move from characterizing the learning behavior of different types of *PII*, to examining how such behavior translates into practical leakage risks, and finally to establishing whether a causal relationship exists between the two.

RQ₁ [Learning Difficulty Across PII Types] Do different types of *PII* exhibit varying degrees of learning difficulty during model training?

Motivation: Previous work has shown that *LLM4Code* memorize sensitive information from their training corpora [Yang et al.(2024), Carlini et al.(2021), Huang et al.(2024), Niu et al.(2023)]. However, most studies treat *PII* as a single, undifferentiated category without considering the heterogeneity among different types. In reality, different *PII* types (*e.g.*, *Key*, *Name*, and *Password*) have different formats, distributions, and contextual usage in source code. These differences may influence the ease with which a *LLM4Code* learns to memorize them. Understanding whether certain *PII* types are inherently easier or harder to learn during training provides critical insights into the underlying risks of memorization.

RQ₂ [Learning Difficulty–Leakage Risk Relationship] Does the learning difficulty of different *PII* types affect their likelihood of being leaked during attack?

Motivation: One might expect that tokens that are easier to learn are more likely to be consistently memorized by *LLM4Code* and consequently more vulnerable to attacks, whereas tokens that are harder to learn are less likely to appear in the model’s output. However, it remains an open question whether this phenomenon holds across all *PII* types. Although different types exhibit varying levels of learning difficulty, it is still unclear whether such differences translate into practical leakage risks. To address this question, we investigate the relationship between the training dynamics of each type and their corresponding attack success rates, thereby linking the memorization process during training to observable leakage behavior inference.

RQ₃ [Causal Effect of Learning Dynamics on Leakage Risk] What is the causal relationship between the learning dynamics of different *PII* types and their likelihood of being leaked during inference?

Motivation: Although **RQ₂** examines the relationship between learning difficulty and leakage tendency empirically, they do not establish whether there is a causal link between the two. To go beyond the observed

correlation, we rigorously investigate whether the way in which a *PII* instance is learned directly influences its leakage risk by causal inference. Specifically, we categorize the training dynamics into three levels of learning difficulty: *easy*, *ambiguous*, and *hard*, based on each model’s confidence in predicting the true class and the variability of this confidence across epochs. We then compute the causal effect of each difficulty level on the attack success rate for every *PII* type, providing a unified view of how the dynamics of memorization shape the privacy risks.

4 Dataset Construction

Although research on software privacy breaches develops datasets containing *PII*, such datasets (*e.g.*, [Niu et al.(2023), Basak et al.(2023)]) are rarely made publicly available due to regulations such as GDPR and HIPAA and to avoid introducing practical security and privacy concerns. Moreover, the contextual distribution of different types of *PII* is often difficult to simulate. Hence, the synthetic datasets used in the previous work [He et al.(2025)] are not suitable for our purpose.

To investigate how different types of real-world *PII* can be memorized and leaked by *LLM4Code*, we constructed a dataset containing six types of *PII* from real-world code corpora widely used for training models. The dataset consists of 9,000 code files drawn from the Java subset of The Stack v2, where each file contains at least one sensitive *PII*. As a result, each type contains at least 1,500 instances of *PII*.

We experimented with multiple detection tools, including TruffleHog [Truffle Security(2025)], Detect-secrets [Yelp(2025)], and Presidio [Microsoft(2025)], for *PII* matching. However, we find that their performance was unsatisfactory due to various specific limitations (elaborated in Section 4.2). Our work thus proposes a methodology for the construction of a type-based *PII* dataset, laying the foundation for future research in this direction.

The construction process is illustrated in Fig. 3. We designed an automated data collection and refinement pipeline that integrates both rule-based (*i.e.*, regular expressions and heuristics) and LLM-based detection methods. To ensure dataset reliability, two researchers in software engineering conducted a sampling-based annotation test. At a 95% confidence level with a 5% margin of error, we estimate that 84.2% to 94.5% of the filtered dataset consists of highly plausible sensitive *PII* instances. In the rest of this section, we present the detailed process of how the dataset was constructed, including data collection, *PII* retrieval, LLM-based refinement, and expert evaluation.

4.1 Data Collection

We selected a widely used dataset built based on real-world software code, *i.e.*, the `Stack v2` [Lozhkov et al.(2024)], as our source, ensuring that the identified *PII* instances had been used for *LLM4Code* training and were highly likely to be memorized. Specifically, we used the `bigcode/the-stack-v2-dedup` version hosted on Hugging Face, which performs near-deduplication while retaining as many programming languages as possible. This dataset contains 6,457.14 GB of code in 658 different programming languages. To ensure that the fine-tuned models effectively capture the context of *PII*, we focus on one of the popular programming languages: Java. We downloaded 1.2M Java files out of the total of 154.28M Java files in `the-stack-v2-dedup`. From this collection, we constructed a subset of 9,000 highly suspicious *PII* instances, demonstrating the widespread presence of *PII* in the dataset.

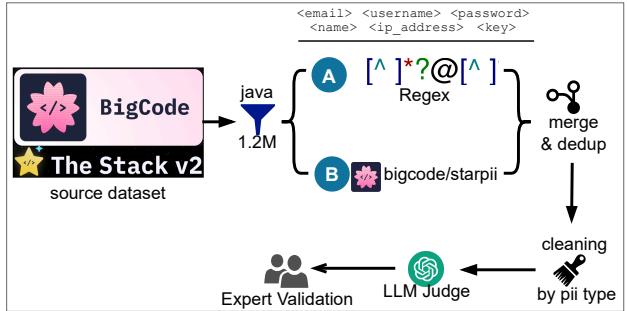


Figure 3: Dataset Curation Process

Table 1: Statistics of detection results for regex match and StarPII

	Email	Key	IP Address	Name	Username	Password
Regex-based PII Detection	53265	601	10554	0	0	0
StarPII	6137	31855	9608	157751	287461	50564

Table 2: Statistics of LLM refinement results

	Email	Key	IP Address	Name	Username	Password
Filtered by Pre-check	471	4438	13388	208	2104	27880
Evaluated by LLM	2953	2356	6029	2356	3425	21544
Exceeding Threshold	2000	2000	1772	2000	2000	1958
Exceeding Threshold Rate	67.73%	84.89%	29.39%	84.89%	58.39%	9.09%

4.2 PII Retrieval

After experimenting with multiple *PII* detection tools, we found that many of them produced excessive false positives, significantly increasing the cost of subsequent processing. Specifically, we employed three *PII* detection tools: TruffleHog [Truffle Security(2025)], Detect-secrets [Yelp(2025)], and Presidio [Microsoft(2025)]. Detect-secrets only reports the line number where *PII* may appear, without providing the actual *PII*, making it unsuitable for our task. In our evaluation on 1,000 source files, TruffleHog and Presidio identified 5 and 38,230 *PII* matches, respectively. Closer inspection revealed that TruffleHog only flagged five instances of Java Database Connectivity (JDBC) strings, none of which contained actual passwords or production-level public addresses and thus did not constitute real *PII*. Presidio, on the other hand, detected 15 different *PII* types. However, except for email addresses, the precision of its classifications was below 10% (based on manual inspection of 50 random samples per type). This limitation comes from the fact that Presidio is designed primarily for processing natural language text, making it not suitable for detecting *PII* in the source code.

Through iterative trials and filtering, we employ three complementary methods:

- **Regex-based detection method.** It is worth noting that although we initially used regexes for five *PII* types, for accuracy considerations we only retained three types in the final dataset: *Email*, *Key* and *IP Address*. The complete set of regexes used for the detection of *PII* is provided in our replication package. Specifically, the regexes for *Email* and *IP Address* were adapted from [Allal et al.(2023)], while the regex for *Key* was derived from the list of 18 representative secret types in GitHub Secret Scanning [Huang et al.(2024)]. The *Key* type includes widely used credentials issued by major service providers, such as `aws_access_key_id`. Leakage of such credentials and their corresponding secret key is particularly critical: An attacker could impersonate an AWS identity to perform arbitrary API calls, including downloading S3 data, deleting resources, injecting malicious code, or deploying illicit cryptocurrency mining programs.
- **LLM-based detection method.** StarPII is a dedicated *PII* detection model developed in [Li et al.([n. d.])]. It supports the detection of six types of *PII*: *Email*, *Key*, *IP Address*, *Name*, *Username*, and *Password*. A detailed breakdown of the retrieval results is presented in Table 1. It should be noted that although regex-based *PII* detection identified only 601 instances of *Key*, the extracted samples exhibit relatively high quality. We merged the results obtained from regex-based detection and StarPII; if two *PII* instances from the same code file had identical values, we treated them as the same data point.
- **Heuristic rules to refine the results.** Following the suggestions of the StarPII developers [Hugging Face(2023)], we further applied several heuristic rules to refine the results. For example, we ignored secrets with fewer than 4 characters or more than 300 characters and used the `ipaddress` Python package to filter out invalid or private IP addresses. For *Name*, we retained only full names. Despite these refinements, many *PII* instances—particularly those in the *Password* and *Key* types—remained false positives. Therefore, in the subsequent step, we leveraged LLMs to further refine and validate the extracted *PII* data.

4.3 LLM Refinement

Because manually annotating all detected *PII* values would be prohibitively costly in terms of both time and resources, we leveraged LLMs' capability to assess further whether the detected data corresponded to genuine sensitive information. To reduce the cost of LLM usage and accelerate the refinement process, we first designed a set of pre-check rules for each *PII* type. These rules filtered out low-quality noisy candidates, which are often caused by formatting errors or placeholder values. The complete list of pre-check rules is presented in Table 3 and is included in the replication package. Candidates who did not pass these quick checks were discarded, and only the remaining instances were refined using LLM. For example, for *Key*, we constructed a list of common test keys and discarded any instances that matched this list. Additionally, we filtered out keys shorter than nine characters or those with Shannon entropy below three.

Through iterative trials and analysis of LLM annotation capabilities, we instructed LLM to evaluate each candidate *PII* instance in three dimensions:

- **Format:** *What is the valid format for this type of PII, and what are the common invalid cases?*
- **Context:** *In what contexts is this type of PII typically sensitive, and does the candidate occur within a test file?*
- **Realness:** *Is the candidate a placeholder, a dummy value, or only a partial PII value?*

We incorporated type-specific heuristics to mitigate false positives. For example, for *Username*, if the string was a nickname or contained only a first name or last name, it was treated as non-sensitive. The complete prompts used in this step are provided in our replication package. We employed `gpt-5-mini` to refine the collected *PII* data using one-shot prompting guided by these criteria. Moreover, after experimenting with context window sizes of 100, 200, and 500 characters, we settled on providing the LLM with 500 preceding and 500 succeeding characters surrounding each target *PII* value. The LLM then assigned a sensitivity score between 0 and 100, with higher values indicating a greater likelihood of sensitivity. To ensure balance between *PII* types, we enforced a minimum of 1,500 instances per type with sensitivity scores that exceed the threshold. Given that *Password* and *IP Address* instances exhibited higher false positive rates during subsequent manual validation, we set their thresholds at 95, while using 90 for the other types. We report more details about this process in Table 2. In particular, only 9% of the *Password* instances exceeded the threshold, indicating that they are highly prone to false positives.

4.4 Expert Evaluation

To validate the reliability of the LLM-refined dataset, we recruited two researchers in software engineering, each with at least four years of Java experience, to manually annotate a subset of the data. We first sampled 100 instances of each type to estimate the expected proportion of valid *PII*, which was approximately 0.9. To determine the required sample size for manual validation, we applied the standard sample size estimation formula with finite-population correction. For each *PII* type, with a population size of 1,500, a 95% confidence level, and a 5% margin of error, the minimum required sample size was 127. Consequently, we randomly sampled 150 instances per type, which were independently reviewed by the two experts. Specifically, we asked participants the following question: "*Is the highlighted PII value in the code file considered sensitive information?*" Participants responded using a three-point Likert scale [Joshi et al.(2015)], where 1, 2, and 3 corresponded to *Disagree*, *Uncertain*, and *Agree*, respectively. The results showed that for each *PII* type, at least 89.33% of the instances were labeled as 3, indicating strong agreement that the highlighted values represented sensitive information.

Based on this expert evaluation, we obtained the final dataset consisting of 9,000 code files containing sensitive *PII*, evenly distributed over six types with 1,500 instances per type. We then partitioned the dataset by random sampling into training, validation, and test sets that contain 7,200, 900, and 900 samples, respectively.

Table 3: Complete list of pre-check rules for each *PII* type

	<i>Pre-Check Rules</i>
<i>Email</i>	Exactly one "@" symbol must exist; both local and domain parts must be non-empty. No leading, trailing, or consecutive ("..") dots. No whitespace, slashes, or parentheses. Domain Validation: The domain must not start or end with a dot. The top-level domain is verified against the official IANA list.
<i>IP Address</i>	IP Address must be syntactically valid and informative, i.e., publicly routable. Private, loopback, link-local, unspecified, multicast, reserved, and broadcast addresses are rejected.
<i>Key</i>	No suspicious characters (slashes, whitespace, braces). Minimum length of 16.
<i>Name</i>	At least 8 unique characters. Shannon entropy more than 3.0, indicating sufficient randomness. Must conform to patterns of full English names (more than 2 parts). Permitted separators: space, period, hyphen, or comma. Strings that fail to match multi-word English name formats are discarded.
<i>Password</i>	Minimum length of 8. All characters must belong to the allowed secure charset. Exclusion of known weak or common passwords (e.g., "123456", "admin", "password1").
<i>Username</i>	Only letters, digits, underscores, and hyphens permitted. Minimum of 6 characters. Disallow known false usernames (e.g., "admin", "root", "guest", "testuser").

5 Experimental Settings

To systematically examine how different types of *PII* influenced leakage rates, we conducted a comprehensive empirical study. Specifically, we analyzed training dynamics during fine-tuning on our constructed *PII* dataset and simulated realistic attack scenarios to evaluate leakage risks. Finally, we employed Structural Causal Modeling (*SCM*) to control for potential confounders and revealed the causal relationship between training dynamics and attack success rates.

5.1 Models

We focus on fine-tuning *LLM4Code*, as it has been extensively applied in both research and industry [He et al.(2025)]. For the downstream task, we select *code completion*, a core feature of nearly all modern integrated development environments and used by prior work in privacy leak research [Huang et al.(2024)]. Code completion suggests relevant code snippets based on the current programming context. Analyzing token-level completions allows us to examine how effectively *PII*-related tokens are learned and memorized by *LLM4Code*. We also select models that support the *Fill-in-the-Middle (FIM)* objective, as the suffix provides additional context that may increase the risk of leakage [Lukas et al.(2023)].

To ensure the generalizability of our findings, we fine-tune models with different architectures and sizes on our constructed dataset and analyze their training dynamics. Specifically, we include 3 widely used open-source models capable of completing code snippets:

- **Stable Code.** Stable Code [Pinnaparaju et al.(2024)] is a decoder-only *LLM4Code* trained for code completion that supports FIM. In our study, we use the `bigcode/the-stack-v2-dedup` version from Hugging Face.
- **CodeLlama.** CodeLlama [Roziere et al.(2023)] is derived from Llama 2 and designed for code generation and FIM tasks. It supports multiple languages including Python, C++, and Java. We use the models `codellama/CodeLlama-7b-hf` and `codellama/CodeLlama-13b-hf`.
- **Qwen2.5-Coder.** Qwen2.5-Coder [Hui et al.(2024)] is based on the Qwen2.5 Transformer architecture and supports long context lengths as well as FIM objectives. It is trained on more than 5.5 trillion tokens from various domains, including code, natural language, mathematics, and synthetic corpora. We use `Qwen/Qwen2.5-Coder-3B`, `Qwen/Qwen2.5-Coder-7B`, and `Qwen/Qwen2.5-Coder-14B`.

In general, our selected models span three architectures and three model sizes, each size represented by at least two architectures.

5.2 Evaluation Metrics

Metrics related to training dynamics. **Confidence** measures how strongly a model predicts the correct token on average across training epochs, while **variability** captures the stability of these predictions (elaborated in Sec. 2.2). High confidence and low variability indicate that the model consistently learns a token, whereas low confidence or unstable predictions suggest difficulty. To translate these raw signals into interpretable categories, we classify each *PII* instance into one of three levels of learning difficulty: *easy-to-learn*, *hard-to-learn*, or *ambiguous*.

Building on the methodology proposed by Swayamdipta et al. [Swayamdipta et al.(2020)] for dataset cartography, we adopt a quantile-based method to determine thresholds from the empirical distributions of confidence and variability across all samples. Instances with confidence above the 75th percentile and variability below the 25th percentile are labeled as **easy-to-learn**, since the model predicts them correctly and consistently. Instances with confidence below the 25th percentile and variability below the 25th percentile are labeled as **hard-to-learn**, reflecting tokens that are rarely predicted correctly despite stable training behavior. Instances with variability above the 75th percentile are labeled as **ambiguous**, as their predictions fluctuate between epochs without convergence. Samples that fall between these thresholds are assigned to the nearest category on the basis of which axis is more decisive.

Metrics related to vulnerability to *PII Attacks*. Following prior work [Huang et al.(2024), Niu et al.(2023)], we evaluate the leakage of fine-tuned models using a black-box query strategy. Specifically, we construct completion inputs consisting of partial prefixes and suffixes of code snippets surrounding a target *PII* instance. If the completion of the model contains the target *PII*, we count it as a successful attack. Formally, the ***PII* attack success** for a target *PII* i is defined as follows. Let the ground-truth *PII* token (or token sequence) be $\mathbf{s}_i = (s_{i,1}, \dots, s_{i,m_i})$, and let the query input constructed from its prefix and suffix context. When the model generates a completion $\hat{\mathbf{y}}_i = (\hat{y}_{i,1}, \dots, \hat{y}_{i,L_t})$, we define the single-attempt attack success indicator as

$$A_i^{(t)} = \mathbb{1}[\mathbf{s}_i \sqsubseteq \hat{\mathbf{y}}_i^{(t)}],$$

where \sqsubseteq denotes that \mathbf{s}_i appears as a contiguous subsequence of the completion. A higher attack success rate indicates that the fine-tuned model is more likely to reproduce memorized sensitive *PII* tokens when prompted with surrounding context, thereby reflecting greater leakage risk.

5.3 Structural Causal Modeling

To test whether learning dynamics causally influence *PII* leakage, we formulate a *SCM* as shown in Fig. 4. The treatment variable (T) represents the learning difficulty of a *PII* instance, with easy cases as control (T_0) and hard (T_1) or ambiguous (T_2) cases as treatments. The outcome (Y) is the success rate of a *PII Attack*. Confounders (Z) include structural features of the code (e.g., `<nloc>`, `<token_counts>`, `<#ast_levels>`, `<#ast_nodes>`, `<#identifiers>`, `<#ast_errors>`, `<complexity>`) that may affect both learning difficulty and leakage.

We compute the Average Treatment Effect (*ATE*), expressed as $P(Y | do(T))$ [Pearl and Mackenzie(2018)], to quantify the expected difference in leakage between treatment and control. Causal estimands are identified using the adjustment formula and estimated with *DoWhy* [Sharma and Kiciman(2020)]. To validate robustness, we apply multiple refutation strategies: (R1) introducing a random common cause, (R2) applying placebo treatments, (R3) adding unobserved confounders correlated with treatment and outcome, and (R4) performing data subset validation. Robustness checks confirm that the estimated effects reflect genuine causal relationships rather than artifacts of sampling or bias.

Interpreting *ATEs*: A negative *ATE* in the easy-hard comparison means that hard instances reduce leakage relative to easy ones, while a positive value would imply the opposite. Similarly, a negative *ATE* in the easy-ambiguous comparison indicates that ambiguous cases have lower leakage compared to easy ones, while a positive value signals that ambiguous cases leak more. In this way, the sign of *ATE* directly reveals whether different learning dynamics amplify or mitigate the risk of leakage *PII*.

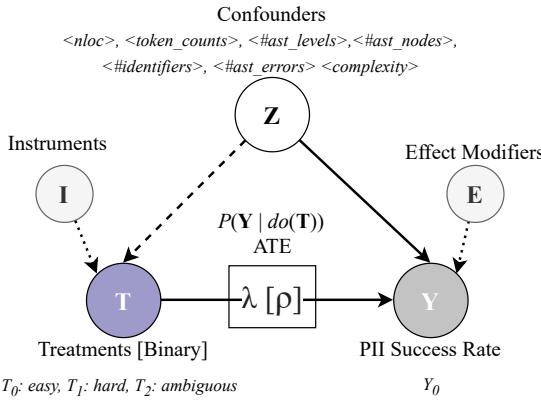


Figure 4: Structural causal model for the relationship between learning dynamics and *PII* leakage.

5.4 Implementation Details

We fine-tune all selected *LLM4Code* using LoRA [Hu et al.(2022)], a technique that efficiently updates the model weights through low-rank adaptations. LoRA, widely adopted in both academia and industry, enables efficient fine-tuning of large models by incorporating task-specific information while preserving pretrained knowledge. Our LoRA configuration uses rank $r = 16$ and scaling factor $\alpha = 32$, following recommendations from previous work [He et al.(2025)].

For FIM, we set `fim_rate` to 0.5, meaning that 50% of training samples are formatted with the FIM objective. We employ the fine-tuning scripts provided by Hugging Face [Hugging Face Transformers contributors([n. d.]) to ensure reproducibility. All models are fine-tuned for ten epochs to capture the training dynamics. The learning rates are set according to the model size: 2×10^{-4} for 3B models, 1×10^{-4} for 7-15B models. All fine-tuned models are evaluated in the code completion task with a context window of 1,024 tokens.

Because *LLM4Code* are typically trained on open-source corpora, prior work has simulated attacks by incorporating the surrounding context of sensitive information collected from platforms such as GitHub into prompts [Huang et al.(2024), Niu et al.(2023)]. When performing the *PII* attack, we construct each code completion input by randomly sampling prefixes and suffixes surrounding a target *PII* instance. Specifically, we remove the target *PII* token itself along with a total of 50 neighboring tokens, randomly distributed between its preceding and subsequent context.

6 Empirical Results

We conducted experiments to address RQs on the impact of the type of *PII* on its leakage in *LLM4Code*. For **RQ₁**, we evaluated the training dynamics of each *PII* type during fine-tuning. For **RQ₂**, we explore the relationship between training dynamics and *PII* attack success risks. To answer **RQ₃**, we use the causal interpretability framework to estimate the true effect of learning dynamics on *PII* leakage, comparing easy, hard, and ambiguous cases across different model families.

6.1 RQ₁: Learning Difficulty Across PII Types

To answer RQ1, we construct a dataset for each *PII* type (as described in Section 4) by extracting instances of that type from the original training corpus of *LLM4Code*. For each *PII* type, we collect 1500 code files that contain highly sensitive *PII*. We then fine-tune the multiple *LLM4Code* of varying architectures and sizes in the *PII* dataset. We report the training loss in Fig. 5, which shows that the fine-tuning process was effective.

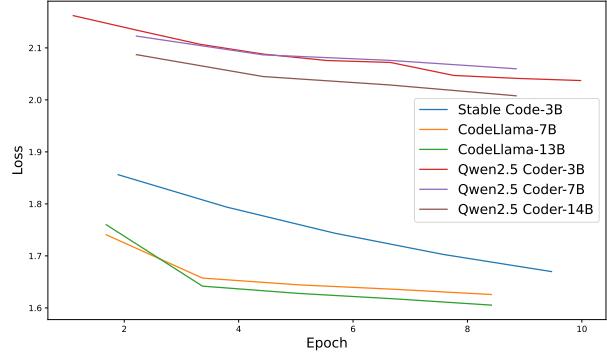


Figure 5: Training loss curves of different *LLM4Code* models across 10 epochs. The abscissa length of the curves varies because of the different batch sizes of the models.

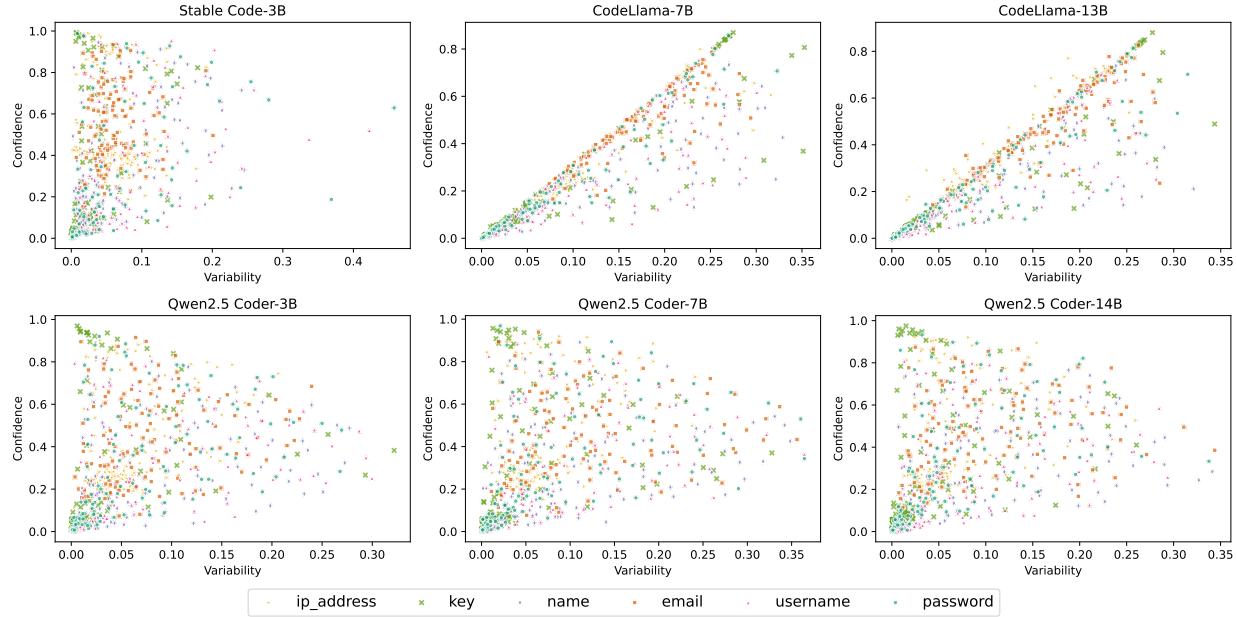


Figure 6: Dataset maps of the constructed training set for each fine-tuned *LLM4Code*. Axes represent variability (x) and confidence (y), with samples down-sampled to preserve distribution. Unlike other models, *CodeLlama-7b* and *CodeLlama-13b* show many high-confidence yet high-variability samples, suggesting that fine-tuning amplifies their memorization.

Qualitative analysis via Dataset Map visualization. To evaluate training dynamics, we measure *confidence* and *variability* (as defined in Section 5.2) for each *PII* instance in training epochs. We compute the training dynamics only for the tokens corresponding to *PII*. Drawing on dataset cartography [Swayamdipta et al.(2020)], we first provide a qualitative analysis by visualizing the training dataset for each model to understand the relative learning difficulty of different *PII* types. In Fig. 6, we illustrate the training dynamics (in the Dataset Map form [Karamcheti et al.(2021)]) of individual *PII* instances over 10 fine-tuning epochs. The y-axis represents *confidence*, while the x-axis represents *variability*. Thus, the top-left corner of the data map (low variability, high confidence) corresponds to *easy-to-learn* examples, the bottom-left corner (low variability, low confidence) to *hard-to-learn* examples, and the right region (high variability) to *ambiguous* examples [Swayamdipta et al.(2020)].

As illustrated in Fig. 6, our analysis reveals distinct differences in the training dynamics of various types of *PII*. Specifically, several *Key* and *IP Address* instances fall within the easy-to-learn region, characterized by confidence values approaching 0.9 and variability close to 0. In contrast, a considerable number of *Email* and *Username* instances occupy the ambiguous region, where the model predictions fluctuate between epochs. Furthermore, the majority of *Key*, along with most *Password* and *Username* instances, are categorized as hard-to-learn, consistently exhibiting low confidence values.

These findings suggest that *IP Address* values are comparatively easier for *LLM4Code* to acquire, while *Password*, *Username*, and most *Key* instances remain considerably more difficult to learn. This outcome aligns with intuition: *Password*, *Username*, and *Key* typically consist of irregular or randomly generated character sequences that inherently lack structural regularities, making them difficult to memorize, even for humans.

Quantitative analysis via metric value distribution. Although the Dataset Map provides qualitative insights, we also present quantitative analyses in Fig. 7 and Fig. 8. Specifically, we discretize the confidence or variability spaces into bins and compute the distribution of instances for each *PII* type. From the results across all models, several key observations emerge:

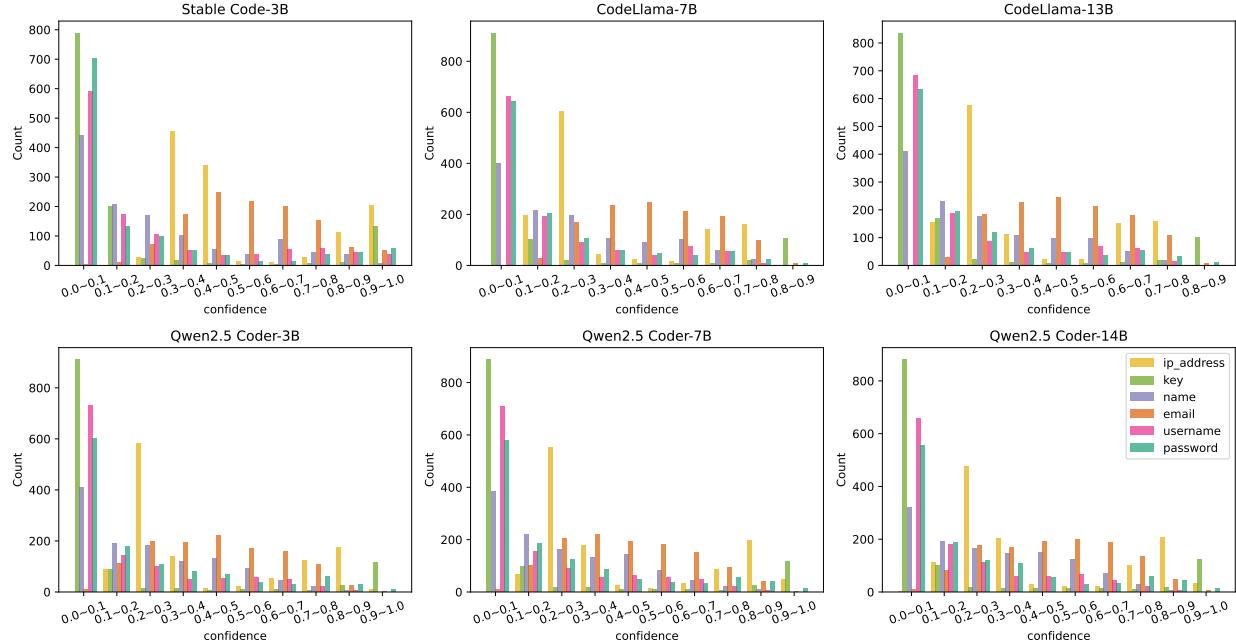


Figure 7: Distribution of prediction confidence across *PII* types for each fine-tuned *LLM4Code*.

- A 13% of *IP Address* and *Key* instances consistently achieve confidence values between 0.8 and 1.0 in epochs, indicating that these examples are relatively easy for the models to learn and predict reliably.
- In contrast, less than 1% of *IP Address* instances have confidence below 0.1, suggesting that *IP Address* is rarely categorized as hard to learn compared to other *PII* types.
- The most challenging instances are found among *Key* values: of the 1,200 examined, at least 66% instances consistently maintain confidence below 0.1 across epochs, highlighting their inherent difficulty for model learning.
- Almost half of the *Username* and *Password* instances fall into the hard-to-learn region, as reflected in their low confidence values across models.
- At least 42% of *Email*, 45% of *Name* instances, and 69% of *IP Address* instances occupy the ambiguous region, characterized by moderate confidence (0.1–0.5) and relative high variability, indicating inconsistent model behavior across epochs.

Taken together, these results demonstrate that different *PII* types exhibit heterogeneous learning difficulty: *IP Address* is generally easier to learn, *Key* shows highly polarized behavior with both easy and most hard instances, and *Username* and *Password* emerge as particularly challenging.

RQ₁: The quantitative analysis of training dynamics reveals that *IP Address* instances are systematically easier for models to acquire, whereas the majority of *Key*, along with most *Password* and *Username* instances, remain considerably difficult to learn.

6.2 RQ₂: Learning Difficulty–Leakage Risk Relationship

In **RQ₂**, we investigate whether the training dynamics obtained in RQ1 correlate with *PII* attack success rates. To this end, we perform the *PII* attack. Prior studies often constructed query templates using contextual information surrounding sensitive values from GitHub [Huang et al.(2024), Niu et al.(2023)]. In

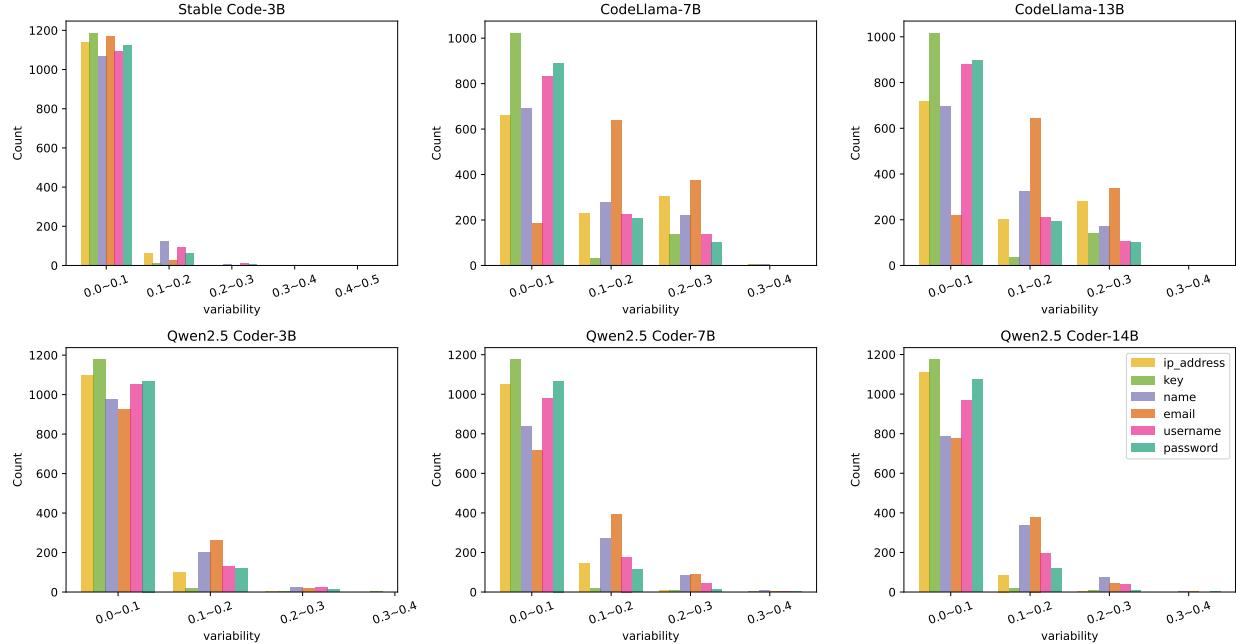


Figure 8: Distribution of prediction variability across *PII* types for each fine-tuned *LLM4Code*.

our setting, we have direct access to such information from the training dataset. For each *PII* instance, we randomly retain part of its surrounding context while removing the target *PII* itself along with 50 neighboring tokens. We then query the model with these inputs for all *PII* instances, resulting in 1,200 queries per type. An attack is counted as successful if the model’s output contains the removed *PII*. The results are reported in Table 4.

Table 4: Complete list of pre-check rules for each *PII* type

	<i>Email</i>	<i>Key</i>	<i>IP Address</i>	<i>Name</i>	<i>Username</i>	<i>Password</i>
<i>starcoder2-3b</i>	42	74	170	56	<u>155</u>	90
<i>Qwen2.5-Coder-3B</i>	60	16	31	58	63	28
<i>CodeLlama-7b</i>	16	8	27	31	30	8
<i>Qwen2.5-Coder-7B</i>	93	26	68	85	117	48
<i>CodeLlama-13b</i>	26	5	13	26	45	9
<i>Qwen2.5-Coder-14B</i>	123	40	77	131	127	52

Successful *PII* attack instances by *LLM4Code* and *PII* type. Cells show valid reproductions.

bold = largest, underline = second largest.

We summarize the relationship between training dynamics and attack success rates as follows:

- *Key* and *Password* instances are predominantly hard-to-learn (low confidence) in the training dynamics. Consistently, they exhibit relatively low attack success rates (e.g., for CodeLlama-7B, *Key* = 8 and *Password* = 8).
- *IP Address* instances are characterized by high confidence and low variability, indicating that they are easy to learn. In line with this, they also show high attack success rates (e.g., 170 for Stable Code-3B).
- *Username* exhibits a remarkably high leakage counts (e.g., 155 in Stable Code-3B, 117 in Qwen2.5-Coder-7B, and 127 in Qwen2.5-Coder-14B). At the same time, *Username* instances are roughly split, with half categorized as hard-to-learn and half as ambiguous. This trend differs from that observed for other *PII* types. Notably, at most 12.92% of *Username* instances are leaked during attacks (Stable Code-3B), making it difficult to establish whether a causal relationship exists between the two.

Overall, these results indicate a generally positive correlation between training dynamics and *PII* attack success rates: easy-to-learn types (e.g., *IP Address*) are more likely to leak, whereas hard-to-learn types (e.g.,

Table 5: Causal Analysis Results (Easy vs Hard to Learn)

	starcoder2-3b		Qwen2.5-Coder-3B		CodeLlama-7b		Qwen2.5-Coder-7B		CodeLlama-13b		Qwen2.5-Coder-14B	
PII Type	Pearson	ATE	Pearson	ATE	Pearson	ATE	Pearson	ATE	Pearson	ATE	Pearson	ATE
<i>name</i>	-0.166	-0.030	-0.112	-0.035	-0.112	-0.035	-0.065	-0.045	-0.079	-0.008	-0.079	-0.008
<i>key</i>	-0.630	-0.452	-0.327	-0.103	-0.327	-0.103	-0.345	-0.151	-0.352	-0.168	-0.352	-0.168
<i>username</i>	-0.401	-0.194	-0.249	-0.291	-0.249	-0.291	-0.043	-0.067	-0.041	-0.002	-0.041	-0.002
<i>password</i>	-0.383	-0.193	-0.118	-0.035	-0.118	-0.035	-0.180	-0.043	-0.186	-0.033	-0.186	-0.033
<i>email</i>	-0.057	-0.031	-0.111	-0.024	-0.111	-0.024	-0.043	-0.077	-0.028	0	-0.028	0
<i>ip_address</i>	-0.025	-0.135	-0.127	-0.021	-0.127	-0.021	-0.120	-0.038	-0.216	-0.110	-0.216	-0.110

bold: – correlation, bold underlined: + correlation, background purple: – causal effect, background grey: + causal effect

Table 6: Causal Analysis Results (Easy vs Ambiguous to Learn)

	starcoder2-3b		Qwen2.5-Coder-3B		CodeLlama-7b		Qwen2.5-Coder-7B		CodeLlama-13b		Qwen2.5-Coder-14B	
PII Type	Pearson	ATE	Pearson	ATE	Pearson	ATE	Pearson	ATE	Pearson	ATE	Pearson	ATE
<i>name</i>	0.094	0.065	0.043	0.018	0.043	0.018	0.059	0.035	0.104	0.132	0.104	0.132
<i>key</i>	-0.323	-0.280	-0.187	-0.102	-0.187	-0.102	-0.080	-0.125	0.067	0.100	0.067	0.100
<i>username</i>	-0.002	0.041	-0.072	-0.158	-0.072	-0.158	0.058	0.101	0.050	0.081	0.050	0.081
<i>password</i>	0.046	0.100	0.067	0.048	0.067	0.048	0.112	0.100	0.121	0.075	0.121	0.075
<i>email</i>	0.008	-0.001	0.041	0.011	0.041	0.011	0.047	0.033	0.080	0.047	0.080	0.047
<i>ip_address</i>	0.008	0.021	-0.039	-0.029	-0.039	-0.029	0.062	0.042	-0.006	-0.007	-0.006	-0.007

bold: – correlation, bold underlined: + correlation, background purple: – causal effect, background grey: + causal effect

Key and *Password*) are less prone to leakage. However, *Username* stands out as a significant exception: although categorized as hard-to-learn or ambiguous in the training dynamics, it consistently exhibits high leakage across different models. To better understand this discrepancy, we further conduct an instance-level causal analysis to investigate the relationship between training dynamics and attack success rates.

RQ₂: Easy-to-learn types, such as *IP Address*, exhibit high leakage risks under attack, whereas hard-to-learn types, such as *Key* and *Password*, correspondingly show low leakage. However, *Username* emerges as a significant exception: despite being mostly classified as hard-to-learn or ambiguous, it consistently demonstrates high leakage across models, which warranting further causal analysis.

6.3 RQ₃: Causal Effect of Learning Dynamics on Leakage Risk

Having analyzed learning difficulty in Sec. 6.1 and its correlation with leakage in Sec. 6.2, we now turn to causal inference in **RQ₃**. The goal is to move beyond correlation and determine whether training dynamics themselves exert a causal effect on leakage risk. To do so, we estimate the Average Treatment Effect (*ATE*) when comparing easy-to-learn instances against those that are hard or ambiguous. *ATEs* allows us to test whether differences in learning dynamics directly shape the likelihood of *PII* leakage.

Table 5 shows the comparison between easy and hard instances. The *Key* and *Username* consistently yield negative *ATEs* across models, meaning that attack success is lower for hard cases than for easy ones. This confirms that when these types are easier to learn, they are also more likely to be leaked, reinforcing the correlations observed in Sec. 6.2. The result ties back to Table 4, where both types showed subsets of easy-to-learn examples that now appear as the main drivers of leakage. The *Password* and *IP Address* also lean negative, but their smaller and less stable *ATEs* suggest that ease of learning only partially explains their leakage behavior. The *Name* and *Email* remain close to zero, which is consistent with their ambiguous or mixed learning dynamics and their weaker correlations with leakage.

Table 6 compares easy and ambiguous instances. For the *Key*, most *ATEs* are negative, which means ambiguous cases leak less than easy ones. This fits with earlier results, where many keys were either very easy or very hard, and leakage was mainly driven by the easiest cases. The *Username* shows a different pattern: in smaller models like *starcoder2-3b* and *Qwen2.5-Coder-3B*, ambiguity reduces leakage, but in larger models like *Qwen2.5-Coder-14B* it slightly increases it. This helps explain why usernames leaked heavily even though they were hard to learn. The *Name* and *Password* sometimes show positive *ATEs* in larger models (e.g., *Qwen2.5-Coder-7B* and *Qwen2.5-Coder-14B*), meaning that ambiguous cases can leak more often than easy ones. This suggests that when models are uncertain, they may still memorize fragments

or fall back on memorized patterns, which can increase exposure instead of reducing it. The *IP Address* stays close to zero, consistent with its stable behavior across analyses.

Taken together, the causal analysis demonstrates that learning dynamics influence leakage in type-specific and model-dependent ways. Easy learning directly drives leakage for the *Key* and *Username*, but ambiguity plays a more complex role: it consistently protects the *Key*, produces scale-dependent behavior for the *Username*, and can increase leakage for the *Name* and *Password*.

RQ₃: Learning dynamics causally shape *PII* leakage risk. The *Key* and *Username* are more likely to leak when easy to learn. Ambiguity consistently protects the *Key*, but for the *Username* its effect varies with model scale. For the *Name* and *Password*, ambiguous cases can increase leakage in larger models. The *IP Address* shows little response to learning dynamics.

7 Discussion

7.1 Findings and Implications

LLM Refinement with proper instruction can be an effective step to reduce false positives. A common challenge in *PII* detection is reducing false positives—for example, *Password* in test files are often non-sensitive, yet most tools cannot systematically exclude them. We demonstrate that carefully designed prompts can mitigate this issue. By instructing the LLM with rules such as “*If the VALUE is used in a test case, such as in a Test function or @Test annotation, it should be given a low score.*” The refinement process eliminates the vast majority of test-related false positives. In our experiments, only 9.09% of *Password* instances met the criteria for sensitive *PII*, underscoring the effectiveness of our LLM refinement.

PII type matters: measuring privacy risk needs with a fine-grained lens. Our study demonstrates that privacy risks are not uniform across *PII* types. Instead, leakage varies with both type and learning dynamics. For example, *Key* and *Username* exhibit higher leakage when they are easy to learn, while *Name* and *Password* sometimes leak more in ambiguous cases for larger models. By contrast, *IP Address* shows high leakage in correlation analysis but little sensitivity in the causal analysis. Our results indicate that mitigation cannot rely on a single defense strategy but must account for the distinct risk profiles of different *PII* types.

Prioritize detection efforts wisely based on learning difficulties. Our findings indicate that different types of *PII* exhibit varying levels of learnability for *LLM4Code*, suggesting that defenses should be type-specific rather than uniform. In particular, RQ1 shows that *IP Address* is one of the easiest types for *LLM4Code* to learn and subsequently leak. The risk is especially severe because *IP Address* often serves as an entry point to larger systems. For instance, many *IP Address* addresses correspond to internal or cloud-hosted databases, where simply knowing the network endpoint can drastically reduce the effort required for unauthorized access. Therefore, it is crucial to design specialized detection and defense mechanisms tailored to *IP Address* leakage.

Protecting PII by more than just detection: injecting synthetic and harmless data can be a new potential method. Current defense approaches primarily focus on removing detected *PII* from training corpora, but our dataset construction shows that detectors for code are often noisy and inaccurate, which limits the effectiveness of removal alone. Our results suggest that defenses should be both type-aware and learnability-aware. Easy-to-learn cases pose the highest risk of leakage and therefore require special attention. Causal results confirm that lowering ease of learning reduces leakage for *Key* and *Username*, while ambiguity has mixed effects for other types. Our results reveal a new and practical defense method that involves adding synthetic but harmless data, which follows the same format and context as the true values. For instance, easy-to-learn synthetic data of *Key* and *Username* can increase the chance that models memorize fake values instead of real ones.

7.2 Threats to Validity

We first discuss **the internal validity of our study**, i.e., the correctness of the experimental implementation. To ensure the reliability of the collected *PII* dataset, we recruited two experienced researchers in software engineering to perform manual verification, as described in Section 4.4. At a 95% confidence level and a 5% margin of error, we estimate that 84.2% to 94.5% of the dataset consists of genuine sensitive *PII* instances. For fine-tuning, we reused the official Hugging Face implementation for FIM-based fine-tuning² and verified that all models employed in our experiments natively support the FIM objective. **Threats to external validity** arise from the representativeness of the studied *PII* types. To address this concern, we extracted *PII* directly from real-world training corpora to ensure authenticity. However, our study focuses exclusively on Java, which may limit the generalizability of our findings to other programming languages where contextual patterns could differ. Nevertheless, since our methodology itself does not depend on language-specific features, we consider this threat to be minor, and leave the evaluation of additional programming languages to future work. Another external validity concern is the size of our dataset, which is relatively small compared to the massive corpora used for pre-training *LLM4Code*. Although our dataset is representative of common fine-tuning scenarios, it may not capture the full range of memorization behaviors that emerge in large-scale training regimes with millions or billions of samples. This limitation introduces uncertainty in generalizing our results to production-level fine-tuning. Finally, **threats to construct validity** refer to the suitability of our evaluation. In this study, we evaluated fine-tuning effectiveness using code completion token accuracy. For privacy leakage, we simulated realistic *PII* attack strategies and measured the *PII* leakage rate, which is consistent with prior research[Huang et al.(2024), Niu et al.(2023)]. Together, these evaluations provide a sound methodological basis for assessing the learning and leakage of different *PII* types in *LLM4Code*.

7.3 *PII* Attacks for *LLM4Code*

GitHub-hosted code is often used to train *LLM4Code*, whose strong memorization raises risks of *PII* leakage. Despite GDPR and HIPAA requiring data protection, commercial *LLM4Code* rarely reveal how they handle *PII*. Once memorized, *PII* can persist in these models even if removed from the original repositories. Carlini et al. demonstrated this risk by extracting hundreds of verbatim training sequences, including *PII* in the training set, from GPT-2 [Carlini et al.(2021)]. Niu et al. developed a semi-automated pipeline to extract sensitive personal information from Codex, the model underlying GitHub Copilot, and showed that 8% of their prompts successfully elicited sensitive data [Niu et al.(2023)]. Lukas et al. pointed out that data-scrubbing techniques cannot fully prevent *PII* leakage, as they must balance the trade-off between minimizing disclosure and preserving dataset utility, and in practice, scrubbing is often imperfect [Lukas et al.(2023)]. More recently, Huang et al. (2024) constructed prompts from GitHub code files containing credentials and demonstrated that *LLM4Code* can reproduce precise training data, including commercial systems such as Gemini and ChatGPT [Huang et al.(2024)].

However, these works have not examined the problem from the training perspective, *i.e.*, whether the success of a *PII* attack is related to what *PII* is being targeted. For example, it remains crucial to study the real-world prevalence of different *PII* types in code repositories and how they are memorized during training, in order to determine whether distinct *PII* types exhibit significant differences in learnability and leakage risk. Such findings can directly inform the design of more effective defense and mitigation strategies.

7.4 Safeguarding *PII* for *LLM4Code*

Protecting *PII* from leakage is a necessary technique for the secure deployment of *LLM4Code*, yet current defenses against *PII* attacks remain limited. Prior studies have explored several directions: for example, Microsoft Presidio [Microsoft(2025)] has been used to anonymize *PII* data before training; Hintersdorf et al. [Hintersdorf et al.(2024)] proposed disabling neurons responsible for memorization in diffusion models to mitigate memorization risks; Geng et al. [Geng et al.(2025)] introduced advanced unlearning algorithms

²https://github.com/huggingface/transformers/blob/main/examples/pytorch/language-modeling/run_fim.py

that can reduce privacy risks in LLMs4Code while preserving their code generation capabilities; and Guo et al. [Guo et al.(2025)] proposed pairing *PII* attack frameworks with CodeLLMs as adversarial dual models, leveraging an improved Group Relative Policy Optimization process to realign the models and enhance their robustness against *PII* extraction attacks.

8 Conclusion

This paper introduced a causality-based approach to interpret privacy risks in *LLM4Code* by linking training dynamics with *PII* leakage behavior. Through a carefully constructed multi-type *PII* dataset, fine-tuning experiments, and structural causal modeling, we showed that leakage risks are not uniform but shaped by both the type of *PII* and its learnability. Easy-to-learn cases such as *IP addresses* tend to leak readily, while hard-to-learn cases like *keys* and *passwords* generally resist extraction, and ambiguous cases exhibit more nuanced, model-dependent behavior. By establishing causal evidence for these patterns, our work provides actionable insights for designing type- and learning-aware defenses, highlighting that effective safeguards must move beyond one-size-fits-all strategies to account for the heterogeneous privacy risks inherent in code models.

Data Availability

To promote transparency and facilitate reproducibility, we make our artifacts available to the community at: https://anonymous.4open.science/r/pii_final-42A1. It includes the implementation details of our *PII* dataset construction procedure, training dynamic computation, and causal inference framework.

References

- [Allal et al.(2023)] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988* (2023).
- [Basak et al.(2023)] Setu Kumar Basak, Lorenzo Neil, Bradley Reaves, and Laurie Williams. 2023. Secret-bench: A dataset of software secrets. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 347–351.
- [Carlini et al.(2022)] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. 2022. Quantifying memorization across neural language models. In *The Eleventh International Conference on Learning Representations*.
- [Carlini et al.(2021)] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. 2021. Extracting training data from large language models. In *30th USENIX security symposium (USENIX Security 21)*. 2633–2650.
- [Ethayarajh et al.(2022)] Kawin Ethayarajh, Yejin Choi, and Swabha Swayamdipta. 2022. Understanding Dataset Difficulty with \mathcal{V} -Usable Information. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.). PMLR, 5988–6008. <https://proceedings.mlr.press/v162/ethayarajh22a.html>
- [GBHackers on Security(2024)] GBHackers on Security. 2024. Massive GitHub Leak Exposed 39 Million API Keys and Secrets. https://gbhackers.com/massive-github-leak/?utm_source=chatgpt.com. Accessed: 2025-09-06.

- [Geng et al.(2025)] Ruotong Geng, Mingyang Geng, Shangwen Wang, Haotian Wang, Zhipeng Lin, and Dezun Dong. 2025. Mitigating Sensitive Information Leakage in LLMs4Code through Machine Unlearning. *arXiv preprint arXiv:2502.05739* (2025).
- [GitHub Inc.([n. d.])] GitHub Inc. [n. d.]. GitHub. <https://github.com/>. Accessed: 2025-09-01.
- [Guo et al.(2025)] Yongjian Guo, Wanlun Ma, Xi Xiao, Sheng Wen, Peng Di, and Xiaogang Zhu. 2025. Patch the Leak: Strengthening CodeLLMs Against Privacy Extraction Threats. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 195–199.
- [He et al.(2025)] Jinwen He, Yiyang Lu, Zijin Lin, Kai Chen, and Yue Zhao. 2025. PrivacyXray: Detecting Privacy Breaches in LLMs through Semantic Consistency and Probability Certainty. *arXiv preprint arXiv:2506.19563* (2025).
- [Hintersdorf et al.(2024)] Dominik Hintersdorf, Lukas Struppek, Kristian Kersting, Adam Dziedzic, and Franziska Boenisch. 2024. Finding nemo: Localizing neurons responsible for memorization in diffusion models. *Advances in Neural Information Processing Systems 37* (2024), 88236–88278.
- [Hu et al.(2022)] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.
- [Huang et al.(2024)] Yizhan Huang, Yichen Li, Weibin Wu, Jianping Zhang, and Michael R. Lyu. 2024. Your Code Secret Belongs to Me: Neural Code Completion Tools Can Memorize Hard-Coded Credentials. *Proc. ACM Softw. Eng.* 1, FSE, Article 111 (July 2024), 23 pages. doi:10.1145/3660818
- [Hugging Face(2023)] Hugging Face. 2023. StarPII. <https://huggingface.co/bigcode/starpII>. Accessed: 2025-09-05.
- [Hugging Face Transformers contributors([n. d.])] Hugging Face Transformers contributors. [n. d.]. run_fim.py example script. https://github.com/huggingface/transformers/blob/main/examples/pytorch/language-modeling/run_fim.py. Accessed: 2025-09-05.
- [Hui et al.(2024)] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).
- [Joshi et al.(2015)] Ankur Joshi, Saket Kale, Satish Chandel, and D Kumar Pal. 2015. Likert scale: Explored and explained. *British journal of applied science & technology* 7, 4 (2015), 396.
- [Karamchetti et al.(2021)] Siddharth Karamchetti, Ranjay Krishna, Li Fei-Fei, and Christopher Manning. 2021. Mind Your Outliers! Investigating the Negative Impact of Outliers on Active Learning for Visual Question Answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Online, 7265–7281. doi:10.18653/v1/2021.acl-long.564
- [Krause et al.(2023)] Alexander Krause, Jan H Klemmer, Nicolas Huaman, Dominik Wermke, Yasemin Acar, and Sascha Fahl. 2023. Pushed by accident: A {Mixed-Methods} study on strategies of handling secret information in source code repositories. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2527–2544.
- [Li et al.([n. d.])] Raymond Li, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, LI Jia, Jenny Chim, Qian Liu, et al. [n. d.]. StarCoder: may the source be with you! *Transactions on Machine Learning Research* ([n. d.]).

- [Lozhkov et al.(2024)] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE]
- [Lukas et al.(2023)] Nils Lukas, Ahmed Salem, Robert Sim, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. 2023. Analyzing leakage of personally identifiable information in language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 346–363.
- [McDaniel(2022)] David McDaniel. 2022. Toyota suffered a data breach by accidentally exposing a secret key publicly on GitHub. <https://blog.gitguardian.com/toyota-accidentally-exposed-a-secret-key-publicly-on-github-for-five-years/>. Accessed: 2025-09-01.
- [Meli et al.(2019)] Michael Meli, Matthew R McNiece, and Bradley Reaves. 2019. How bad can it git? characterizing secret leakage in public github repositories.. In *NDSS*.
- [Microsoft(2025)] Microsoft. 2025. Presidio: Data Protection and De-identification SDK. <https://github.com/microsoft/presidio>. Accessed: 2025-09-11.
- [Nader Palacio et al.(2024)] David Nader Palacio, Alejandro Velasco, Nathan Cooper, Alvaro Rodriguez, Kevin Moran, and Denys Poshyvanyk. 2024. Toward a Theory of Causation for Interpreting Neural Code Models . *IEEE Transactions on Software Engineering* 50, 05 (May 2024), 1215–1243. doi:10.1109/TSE.2024.3379943
- [Niu et al.(2023)] Liang Niu, Shujaat Mirza, Zayd Maradni, and Christina Pöpper. 2023. {CodexLeaks}: Privacy leaks from code generation language models in {GitHub} copilot. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2133–2150.
- [OpenAI and et al(2024)] OpenAI and Josh Achiam et al. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] <https://arxiv.org/abs/2303.08774>
- [Pearl and Mackenzie(2018)] Judea Pearl and Dana Mackenzie. 2018. *The Book of Why: The New Science of Cause and Effect* (1st ed.). Basic Books, Inc., USA.
- [Pinnaparaju et al.(2024)] Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, Ashish Datta, Maksym Zhuravinskyi, Dakota Mahan, Marco Bellagente, Carlos Riquelme, et al. 2024. Stable code technical report. *arXiv preprint arXiv:2404.01226* (2024).
- [Rahman et al.(2022)] Md Rayhanur Rahman, Nasif Imtiaz, Margaret-Anne Storey, and Laurie Williams. 2022. Why secret detection tools are not enough: It's not just about false positives-an industrial case study. *Empirical Software Engineering* 27, 3 (2022), 59.
- [Roziere et al.(2023)] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaojing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

- [Sharma and Kiciman(2020)] Amit Sharma and Emre Kiciman. 2020. DoWhy: An End-to-End Library for Causal Inference. *arXiv preprint arXiv:2011.04216* (2020).
- [Shen and Sanghavi(2019)] Yanyao Shen and Sujay Sanghavi. 2019. Learning with bad training data via iterative trimmed loss minimization. In *International conference on machine learning*. PMLR, 5739–5748.
- [Swayamdipta et al.(2020)] Swabha Swayamdipta, Roy Schwartz, Nicholas Lourie, Yizhong Wang, Hanneh Hajishirzi, Noah A Smith, and Yejin Choi. 2020. Dataset cartography: Mapping and diagnosing datasets with training dynamics. *arXiv preprint arXiv:2009.10795* (2020).
- [Toneva et al.(2018)] Mariya Toneva, Alessandro Sordoni, Remi Tachet des Combes, Adam Trischler, Yoshua Bengio, and Geoffrey J Gordon. 2018. An empirical study of example forgetting during deep neural network learning. *arXiv preprint arXiv:1812.05159* (2018).
- [Truffle Security(2023)] Truffle Security. 2023. Thousands of GitHub Comments Leak Live API Keys. <https://trufflesecurity.com/blog/thousands-of-github-comments-leak-live-api-keys>. Accessed: 2025-09-06.
- [Truffle Security(2025)] Truffle Security. 2025. TruffleHog: Find and Analyze Leaked Secrets. <https://github.com/trufflesecurity/trufflehog>. Accessed: 2025-09-11.
- [Wiz(2023)] Wiz. 2023. 38 Terabytes of Private Data Accidentally Exposed by Microsoft AI Researchers. <https://www.wiz.io/blog/38-terabytes-of-private-data-accidentally-exposed-by-microsoft-ai-researchers> Accessed: 2025-09-01.
- [Yang et al.(2024)] Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, Donggyun Han, and David Lo. 2024. Unveiling memorization in code models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [Yelp(2025)] Yelp. 2025. detect-secrets: Enterprise-friendly way to detect secrets. <https://github.com/Yelp/detect-secrets>. Accessed: 2025-09-11.