# Scrub It Out! Erasing Sensitive Memorization in Code Language Models via Machine Unlearning

Zhaoyang Chu*
Huazhong University of Science and Technology
Wuhan, China
chuzhaoyang@hust.edu.cn

Yao Wan*†
Huazhong University of Science and Technology
Wuhan, China
wanyao@hust.edu.cn

Zhikun Zhang
Zhejiang University
Hangzhou, China
zhikun@zju.edu.cn

Di Wang
King Abdullah University of Science and Technology
Thuwal, Saudi Arabia
di.wang@kaust.edu.sa

Zhou Yang
University of Alberta
Edmonton, Canada
zy25@ualberta.ca

Hongyu Zhang
Chongqing University
Chongqing, China
hyzhang@cqu.edu.cn

Pan Zhou
Huazhong University of Science and Technology
Wuhan, China
panzhou@hust.edu.cn

Xuanhua Shi*
Huazhong University of Science and Technology
Wuhan, China
xhshi@hust.edu.cn

Hai Jin*
Huazhong University of Science and Technology
Wuhan, China
hjin@hust.edu.cn

David Lo
Singapore Management University
Singapore, Singapore
davidlo@smu.edu.sg

## Abstract

While *Code Language Models* (CLMs) have demonstrated superior performance in software engineering tasks such as code generation and summarization, recent empirical studies reveal a critical privacy vulnerability: these models exhibit unintended memorization of sensitive training data, enabling verbatim reproduction of confidential information when specifically prompted. To address this issue, several approaches, including training data de-duplication and differential privacy augmentation, have been proposed. However, these methods require full-model retraining for deployed CLMs, which incurs substantial computational costs. In this paper, we aim to answer the following research question: *Can sensitive information memorized by CLMs be erased effectively and efficiently?*

We conduct a pioneering investigation into erasing sensitive memorization in CLMs through machine unlearning—a *post-hoc* modification method that removes specific information from trained models without requiring full retraining. Specifically, we first quantify the memorization risks of sensitive data within CLM training datasets and curate a high-risk dataset of 50,000 sensitive memorized samples as unlearning targets. We study two widely used gradient ascent-based unlearning approaches: the vanilla and constraint-based methods, and introduce CodeEraser, an advanced variant that *selectively* unlearns sensitive memorized segments in code while preserving the structural integrity and functional correctness of the surrounding code. Extensive experiments on three families of CLMs, *i.e.*, CodeParrot, CodeGen-Mono, and Qwen2.5-Coder, validate the effectiveness and efficiency of CodeEraser in erasing targeted sensitive memorization while maintaining model utility.

## CCS Concepts

• **Software and its engineering** → **Software development techniques**; • **Security and privacy**;

## Keywords

Code Language Models, Code Generation, Privacy, Sensitive Memorization, Machine Unlearning

**ACM Reference Format:**
Zhaoyang Chu, Yao Wan, Zhikun Zhang, Di Wang, Zhou Yang, Hongyu Zhang, Pan Zhou, Xuanhua Shi, Hai Jin, and David Lo. 2026. Scrub It Out! Erasing Sensitive Memorization in Code Language Models via Machine Unlearning. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3764573

*Also with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.
†Yao Wan is the corresponding author.


*ICSE '26, Rio de Janeiro, Brazil*

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3764573

## 1 Introduction

Recently, *Code Language Models* (CLMs), such as CodeGen [50], Code Llama [54], and Qwen2.5-Coder [34], have demonstrated significant potential in automating various aspects of software engineering, including code generation [22, 38, 63], code summarization [2, 5, 62], program repair [65, 66], and type inference [44]. Their success can be attributed to pre-training with autoregressive language modeling on large-scale code corpora [60, 61], where the model predicts the next token given a sequence of previous tokens.

However, studies have shown that the current pre-training paradigm may retain sensitive data, *e.g.*, emails and passwords, encountered during the training phase [3, 15, 17, 51, 68]. This retention occurs because CLMs, trained on vast amounts of code collected from GitHub repositories, may inadvertently memorize sensitive data embedded within these repositories, including personally identifiable information (*e.g.*, *names*, *emails*, and *phone numbers*) and private authentication credentials (*e.g.*, *passwords*, *API keys*, and *cryptographic secrets*) [7, 27, 33, 46, 51, 68].

From another perspective, to strengthen individual control over personal data, global legislative frameworks such as the European Union's *General Data Protection Regulation* (GDPR) [57] and the *California Consumer Privacy Act* (CCPA) [52] have established the "*Right to Be Forgotten*" (RTBF) [58, 59]. These regulations empower individuals to request the deletion of their personal data, providing a critical safeguard for privacy protection.

To mitigate the privacy risks posed by potential data leaks in CLMs and ensure compliance with the RTBF, we investigate the following question: *Can sensitive information memorized by CLMs be erased effectively and efficiently?*

**Intuitive Approaches and Limitations.** Our investigation reveals two distinct research directions. The first focuses on training data de-duplication, as illustrated in Figure 1 (a). Prior studies [15, 39, 43] have demonstrated that de-duplication can mitigate the memorization tendencies of LMs. However, experimental evidence indicates that LMs still retain substantial memorization capabilities even under this paradigm [8, 9, 37].

Another line of research falls into *Differential Privacy* (DP) [1, 74], as shown in Figure 1 (b). This approach enforces the formal guarantee that the addition or removal of any training data point does not substantially affect the final model [73], thereby providing formal privacy guarantees for individual training samples. However, DP-based training fundamentally limits LMs' ability to capture long-tail patterns in data distributions, resulting in significant utility degradation [4, 25, 26].

Furthermore, both DP and de-duplication methods are typically applied during the initial training phase. For already deployed CLMs, these methods lack the ability to selectively remove specific data as requested by users, often necessitating retraining the entire model [18, 37, 48]. Such retraining is costly and time-consuming, especially given the escalating scale of contemporary CLMs. This limitation prevents their flexibility in addressing dynamic user requests and evolving privacy demands in real-world scenarios.

**Our Work: A Machine Unlearning Perspective.** More recently, machine unlearning has emerged as a promising alternative for LMs, as shown in Figure 1 (c), which seeks to remove specific information by *post-hoc* modifying the trained model [11, 14, 18, 30, 37]. Existing approaches typically employ gradient ascent to reverse the learning of specific data, thus proactively removing its influence [18, 37]. Compared to DP and de-duplication techniques, machine unlearning enables LMs to quickly forget certain information with just a few parameter updates without full retraining, thereby reducing the training time from 900∼1800 to ∼0.001 A100 GPU days [37]. Nevertheless, we argue that these approaches often indiscriminately forget entire text instances rather than selectively targeting specific sensitive information. As a result, they struggle to erase sensitive segments (*e.g.*, API key strings) embedded in code without disrupting the structural integrity and functional correctness of the surrounding code.

In this paper, we perform a pioneering investigation into sensitive memorization erasure [1] in CLMs through machine unlearning. Specifically, we first quantify the memorization risks of sensitive data within CLM training corpora and curate a high-risk dataset of 50,000 sensitive memorized samples as unlearning targets. We investigate two widely used gradient ascent-based unlearning approaches: the vanilla method and the constraint-based method, and further develop an advanced variant, termed CodeEraser, which selectively unlearns sensitive memorized segments in code while preserving the surrounding code's integrity and functionality.

To assess the effectiveness and efficiency of CodeEraser, we conduct extensive experiments on three widely used suites of CLMs, *i.e.*, CodeParrot [24], CodeGen-Mono [50], and Qwen2.5-Coder [34]. The results demonstrate CodeEraser's ability to effectively and efficiently mitigate the memorization issue in CLMs, thus protecting sensitive data against potential extraction attacks. Using the Qwen2.5-Coder-7B model as an example, CodeEraser successfully reduces memorization by 93.89% on the targeted forgotten set (sampled from the sensitive memorization dataset), while retaining 99.00% of the model's original performance, with an average processing time of 46.88 seconds per sample.

**Contributions.** The key contributions of this paper are as follows.

- **New Problem and Dataset.** To the best of our knowledge, we are the first to formulate the problem of erasing sensitive memorization within CLMs. As an initial step, we curate a sensitive memorization dataset to support further research in this area.
- **Pioneering Study.** We conduct the first comprehensive study on sensitive memorization erasure in CLMs via machine unlearning. We also introduce a selective gradient-ascent approach CodeEraser to target and remove sensitive memorized segments while preserving code integrity.
- **Extensive Evaluation.** We conduct comprehensive experiments on three widely used families of CLMs, namely CodeParrot [24], CodeGen-Mono [50], and Qwen2.5-Coder [34]. The results demonstrate the effectiveness and efficiency of CodeEraser in erasing sensitive memorization within CLMs while maintaining acceptable levels of model utility.

## 2 Background

We begin by introducing the background of LMs, followed by a formal definition of memorization in these models.

[1] We adopt the term *sensitive memorization* to denote the phenomenon where CLMs retain and reproduce sensitive training data (*e.g.*, API keys). Thus, by *memorization erasure*, we mean techniques designed to remove such retained sensitive content.

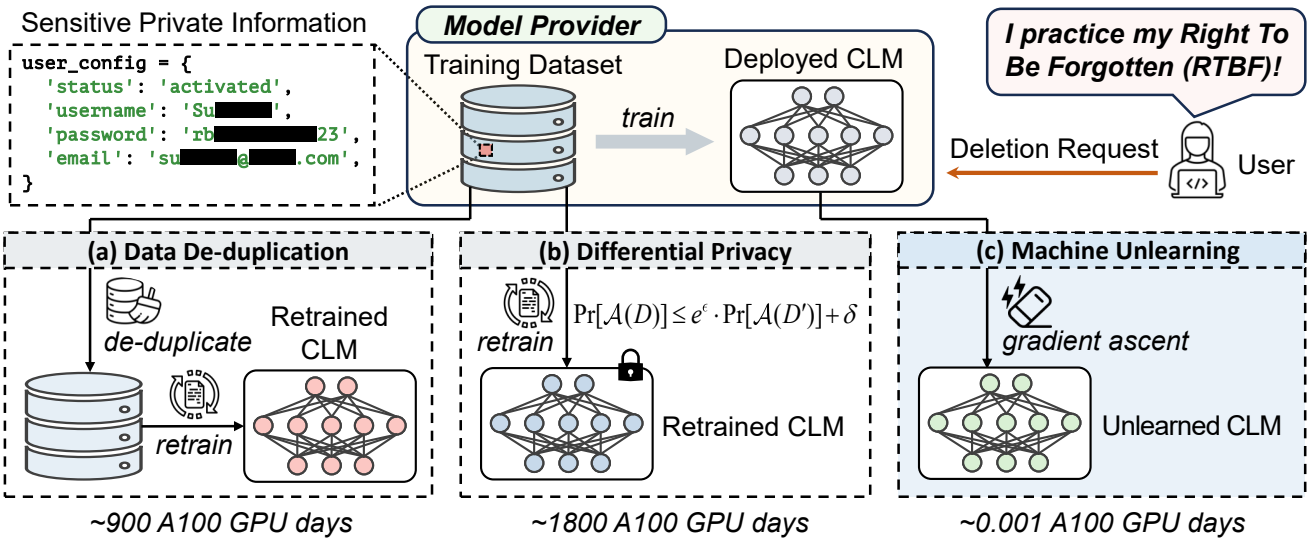


**Figure 1: An illustration of existing methods, *i.e.*, (a) data de-duplication, (b) differential privacy, and (c) machine unlearning, to mitigate the memorization of sensitive information in CLMs. We mask the personal details for ethical considerations.**

## 2.1 Language Models

*Language Models* (LMs) are designed to predict the probability of a token sequence by utilizing the empirical distribution of token occurrences in the training data. One widely adopted unsupervised approach for training LMs is autoregressive language modeling, also known as "next-token prediction", where the model sequentially predicts tokens from left to right [12, 47, 53].

**Autoregressive Language Modeling.** Given a token sequence $\mathbf{x} = (x_1, x_2, \ldots, x_N)$, the LM employs the chain rule to model its joint probability as the product of conditional probabilities:

$$\mathbf{Pr}(x_1, x_2, \ldots, x_N) = \prod_{i=1}^{N} \mathbf{Pr}(x_i \mid x_1, \ldots, x_{i-1}) \,. \tag{1}$$

In this paradigm, a neural network model $f$, parameterized by $\theta$, is typically employed to estimate the likelihood of each token $x_i$ conditioned on preceding tokens, denoted as $f_\theta(x_i \mid x_1, \ldots, x_{i-1})$. The parameters $\theta$ are optimized by maximizing the probability of each sample within the training dataset $\mathcal{D}$. This is achieved by minimizing the loss function as follows:

$$\mathcal{L}^{LM}(\mathbf{x}) = -\log \prod_{i=1}^{N} f_\theta(x_i \mid x_1, \ldots, x_{i-1}) \,. \tag{2}$$

**Model Inference via Prefix Prompt.** Once the LM $f_\theta$ is trained, it can generate outputs based on a given prefix prompt during inference. Given a prefix prompt $p = (x_1, \ldots, x_{i-1})$, the trained LM iteratively predicts the next tokens to complete the suffix $s$. Specifically, the model samples $\hat{x}_i \sim f_\theta(x_i \mid x_1, \ldots, x_{i-1})$ and subsequently feeds $\hat{x}_i$ back into the model to sample $\hat{x}_{i+1} \sim f_\theta(x_i \mid x_1, \ldots, \hat{x}_i)$, iteratively. Each newly generated token is conditioned on both the initial prompt and all previously generated tokens. This decoding process is repeated until a termination condition is met, *e.g.*, generating a special token </s>, indicating the end of the sequence, or reaching a predefined maximum length of the token sequence.

## 2.2 Memorization in Language Models

Memorization, often seen as the antithesis of generalization, arises from overfitting, causing models to retain specific details of their training data [3, 25]. This phenomenon raises remarkable privacy concerns in the context of LMs, as these models may inadvertently memorize sensitive information and regurgitate it verbatim in response to certain prompts.

Extensive research has been undertaken to qualitatively and quantitatively examine memorization in LMs [3, 15–17, 33, 35, 49, 51, 68]. Following these prior studies, we define memorization in LMs grounded in the extractability of training data. In particular, we conceptualize memorization as a model's ability to store and reproduce exact pieces of information encountered during training.

**Definition 1 (Verbatim Memorization).** A string $s$ is considered memorized by an LM $f_\theta$ if there exists a prefix $p$ such that:

$$s = \arg\max_{\hat{s}} f_\theta(\hat{s} \mid p) \;\wedge\; [p \,||\, s] \in \mathcal{D} \,. \tag{3}$$

Here, $f_\theta(\hat{s} \mid p)$ denotes the model's likelihood of generating an entire sequence $\hat{s}$ given the prefix $p$, $[p \,||\, s]$ represents the concatenation of strings $p$ and $s$, and $\mathcal{D}$ denotes the training dataset of $f_\theta$. The arg max operation can be replaced by an appropriate decoding strategy (*e.g.*, greedy sampling) to determine the model's outputs in practical applications.

Example 1. Assume that the LM's training dataset contains a sequence "*# Copyright (C) [2003] Daniel <daniel@gmail.com>*". If the model is prompted with "*# Copyright (C) [2003] Daniel* ___" and the most likely continuation is "*<daniel@gmail.com>*", then the generated string is deemed memorized.

## 3 Preliminary Study

We first conduct a preliminary study to quantitatively examine the presence and severity of sensitive memorization in CLMs.

**Table 1: A toy example to illustrate the calculation process of MA and EL$_n$ with $n = 3$.**

| **Target Sequence (with 12 tokens):** This file is part of EasyBuild created by the HPC team **.** | | | | | |
|---|---|---|---|---|---|
| **Prefix** $x_{<i}$ | **MA = 6 / (12 - 1) = 0.5455** | | **EL$_3$ = (0.2222 + 0.25 + 0.1429 + 0.1667 + 0.2 + 0 + 0 + 0 + 0) / (12 - 3) = 0.1091** | | |
| | **True Continuation** $x_i$ | **Generated Token** $\hat{x}_i$ | **True Continuation** $x_{\geq i}$ *3-gram match* | **Generated Sequence** $\hat{x}_{\geq i}$ | **OVERLAP$_3$** |
| This | file | program | file is part of EasyBuild created by the HPC team . | program is part of pyNLO , which is created by the | 2 / 9 = 0.2222 |
| This file | is | is | is part of EasyBuild created by the HPC team . | is part of PyGithub created by the PYG team . | 2 / 8 = 0.2500 |
| This file is | part | part | part of EasyBuild created by the HPC team . | part of PyGithub created by the PYG team . | 1 / 7 = 0.1429 |
| … file is part | of | of | of EasyBuild created by the HPC team . | of PyGithub created by the PYG team . | 1 / 6 = 0.1667 |
| … is part of | EasyBuild | PyGithub | EasyBuild created by the HPC team . | PyGithub created by the PYG team . | 1 / 5 = 0.2000 |
| … of EasyBuild | created | . | created by the HPC team . | . EasyBuild is free software ; | 0 / 4 = 0.0000 |
| … created | by | by | by the HPC team . | by the VSC team , | 0 / 3 = 0.0000 |
| … created by | the | the | the HPC team . | the VSC team , | 0 / 2 = 0.0000 |
| … created by the | HPC | VSC | HPC team . | VSC team , | 0 / 1 = 0.0000 |
| … by the HPC | team | team | - | - | - |
| … the HPC team | . | , | - | - | - |

*token match*

## 3.1 Study Subjects

**Studied CLMs.** To systematically analyze model memorization, we select representative CLMs varying in size and architecture. Following [3, 68], we examine four widely used models: CodeParrot-small (110M), CodeParrot (1.5B) [24], and CodeGen-{350M, 2B}-Mono[50]. Our analysis also includes Qwen2.5-Coder-7B [34], a state-of-the-art CLM with over 35.3k monthly downloads on HuggingFace at the time of writing. All selected models are accessible on HuggingFace Hub, enabling ethical and reproducible memorization analysis.

**Studied Datasets.** We utilize `codeparrot-clean-train` [21], a 50GB dataset comprising ~5 million Python files. We select it for two key reasons: (1) It is a high-quality, cleaned subset of GitHub corpora, extracted via Google's BigQuery [31], making it representative of standard CLM training data. (2) It offers the repository source for each code instance, enabling realistic unlearning simulations where users request the removal of sensitive data unknowingly included in their repositories. These make it ideal for standardized memorization analysis and unlearning evaluations across CLMs.

## 3.2 Memorization Quantification

*3.2.1 Memorization Metrics.* To accurately assess whether and to what extent CLMs retain specific data, our analysis adopts two quantitative metrics: *Memorization Accuracy* (MA) [56] and *Extraction Likelihood* (EL) [37]. As illustrated in Table 1, these metrics measure memorization by comparing the CLM's generation with the true continuation, at the token and $n$-gram levels, respectively.

***Memorization Accuracy* (MA) [56].** Given a specific token sequence $\mathbf{x} = (x_1, x_2, \ldots, x_N)$, we let the CLM $f_\theta$ sequentially process this sequence from left to right and predict each token based on its preceding context. Then, MA calculates the accuracy of these predictions by comparing the generated tokens with their corresponding actual tokens in the sequence $\mathbf{x}$:

$$\mathrm{MA}(\mathbf{x}) = \frac{\sum_{i=2}^{N} \mathbb{1}\{\arg\max_{\hat{x}_i} f_\theta(\hat{x}_i \mid x_{<i}) = x_i\}}{N-1}, \tag{4}$$

where $\mathbb{1}$ is the indicator function that returns 1 if the condition within the braces is true (*i.e.*, the CLM's most likely prediction $\hat{x}_i$ accurately matches the actual token $x_i$) and 0 otherwise, and the notion $x_{<i}$ denotes the sequence of all tokens before position $i$.

Example 2. As shown in columns 1-3 of Table 1, given various prefixes $x_{<i}$, $\hat{x}_i$ matches $x_i$ with 6 times (marked in green), resulting in an MA score of 0.5455. We can see that MA measures the proportion of tokens in a sequence that the CLM can recall exactly, reflecting its capacity to memorize and reproduce training data.

***Extraction Likelihood* (EL) [37].** Compared with MA computed at the token level, EL enables a stricter standard for quantifying memorization by assessing the extent to which the generated sequence $\hat{x}_{\geq i}$ matches the true continuation $x_{\geq i}$ at the $n$-gram level:

$$\mathrm{EL}_n(\mathbf{x}) = \frac{\sum_{i=2}^{N} \mathrm{Overlap}_n(\arg\max_{\hat{x}_{\geq i}} f_\theta(\hat{x}_{\geq i} \mid x_{<i}), x_{\geq i})}{N-n},$$
$$\mathrm{Overlap}_n(\mathbf{a}, \mathbf{b}) = \frac{\sum_{c \in ng(\mathbf{a})} \mathbb{1}\{c \in ng(\mathbf{b})\}}{|ng(\mathbf{a})|}, \tag{5}$$

where Overlap$_n$ measures the overlap of $n$-grams between two sequences, $ng(\cdot)$ denotes the list of $n$-grams within a sequence. Higher $n$ values represent stricter standards for memorization quantification, adhering to higher privacy requirements. Following [37], our study chooses $n$ values of 3, 5, and 10.

Example 3. As shown in columns 1 and 4-6 of Table 1, in the first row, given the prefix "This", the number of the 3-gram matches between $\hat{x}_{\geq i}$ and $x_{\geq i}$ is 2 (marked in green), leading to an Overlap$_3$ score of 0.2222. After iterating through all the prefixes $x_{<i}$, all the Overlap$_3$ values are averaged to obtain a final EL$_3$ score of 0.1091.

*3.2.2 Memorization Thresholds.* Memorization in CLMs varies, ranging from rarely reproduced sequences to verbatim repetition easily exploitable by adversaries. Without clear boundaries between them, it is difficult to prioritize and address genuine privacy vulnerabilities. To this end, we empirically establish explicit memorization thresholds based on the metrics MA and EL$_n$:

$$T_{\mathrm{MA}} = \frac{1}{|\mathcal{D}'|} \sum_{\mathbf{x}' \in \mathcal{D}'} \mathrm{MA}(\mathbf{x}'), \; T_{\mathrm{EL}_n} = \frac{1}{|\mathcal{D}'|} \sum_{\mathbf{x}' \in \mathcal{D}'} \mathrm{EL}_n(\mathbf{x}'), \tag{6}$$

where $\mathcal{D}'$ denotes a dataset consisting entirely of samples ***unseen*** during the CLM's training phase. Intuitively, a training sample

**Table 2: Memorization thresholds for the studied CLMs.**

| CLM | MA (%) | $EL_3$ (%) | $EL_5$ (%) | $EL_{10}$ (%) |
|---|---|---|---|---|
| CodeParrot-small | 45.57 | 17.66 | 10.82 | 5.49 |
| CodeParrot | 46.34 | 16.56 | 10.17 | 5.14 |
| CodeGen-350M-Mono | 48.79 | 18.24 | 11.03 | 5.92 |
| CodeGen-2B-Mono | 53.61 | 19.32 | 11.71 | 6.28 |
| Qwen2.5-Coder-7B | 40.99 | 15.65 | 12.45 | 8.82 |

with memorization scores below these thresholds, appearing as if the model had never seen it, indicates safe memorization and is thus resistant to extraction attacks. Conversely, samples exceeding these thresholds indicate potential risks of exposure and leakage. The resulting memorization thresholds for the studied CLMs are presented in Table 2.

**Unseen Dataset.** For the Qwen2.5-Coder-7B model, we compile $\mathcal{D}'$ from two popular evaluation datasets, *i.e.*, HumanEval [19] and MBPP [6], which have been explicitly excluded from the CLM's training corpus via data decontamination [34]. For the CodeParrot and CodeGen-Mono families, we compile $\mathcal{D}'$ using a data crawling tool provided by [67], collecting 10,000 high-quality de-duplicated code files from GitHub repositories. Repository selection criteria include: each must have at least 500 stars and be created after the release dates of the CLMs. Moreover, to address potential concerns that files might be copied from older versions already exposed to the CLMs, we only collect code files written in programming languages absent from the CLMs' training, *e.g.*, Ruby, PHP, Rust, and Lua. This strategy ensures that $\mathcal{D}'$ is high-quality and genuinely unseen by the studied CLMs.

## 3.3 Sensitive Memorization Identification

Not all memorization poses privacy risks; for instance, retaining public code snippets is far less concerning than memorizing private keys. While several techniques have been developed to extract memorized contents from CLMs [3, 35, 68], they mainly focus on analyzing non-sensitive code memorization. Recent studies [33, 51] have highlighted privacy risks by eliciting sensitive information from CLMs using well-crafted prompts. However, they only reveal small-scale, isolated examples of sensitive memorization, lacking a systematic analysis of the broader extent of sensitive data retained by CLMs. Thus, we aim to address the question: *To what extent do CLMs memorize sensitive information from their training data?*

**Sensitive Data Identification.** To comprehensively identify sensitive data within code (*e.g.*, emails, IP addresses, and API/SSH keys), we employ `detect-secrets` [71], a widely used regular expression-based detection tool, to scan the entire `codeparrot-clean-train` dataset. After filtering out local IPs and emails containing "example", we find that 939,665 out of 5,300,000 training samples (approximately 18%) contain sensitive information.

**Sensitive Memorization Quantification.** We assess the memorization levels of sensitive segments in identified samples using the MA metric. MA is preferred over $EL_n$ due to its efficiency in token matching, making it more suitable for large-scale analysis than the $n$-gram approach of $EL_n$. Given computational constraints, we restrict our analysis to two relatively small models, *i.e.*, CodeParrot-small and CodeGen-350M-Mono, and limit the examined samples to those containing sensitive data within a maximum token length

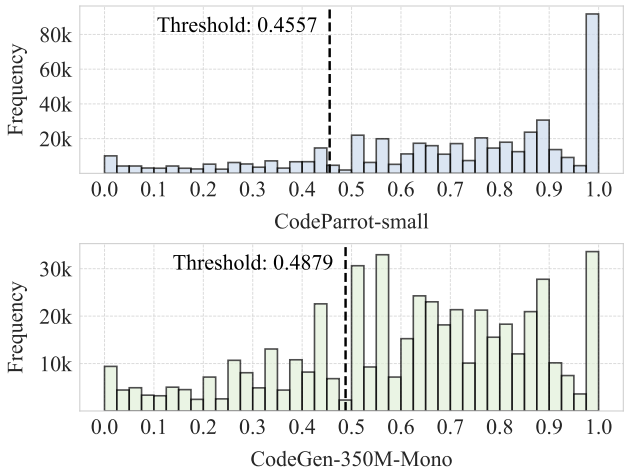


**Figure 2: The distribution of MA across sensitive data.**

(*e.g.*, 512). Moreover, only sensitive segments are considered in this quantification; surrounding non-sensitive code is excluded. For each sensitive segment, we prepend a fixed non-sensitive prefix (up to 128 tokens) when computing memorization. For instances containing multiple sensitive segments, we calculate the average MA score across all segments. These measures allow us to complete quantification on the full training dataset within 6 hours using a single GPU equipped with 80GB of memory.

As shown in Figure 2, we find that 376,740 out of 473,994 training samples in CodeParrot-small and 363,806 out of 501,549 in CodeGen-350M-Mono (approximately 7% of training data) exhibit sensitive memorization, with MA scores exceeding the established memorization thresholds.

**Finding 💡:** CLMs such as CodeParrot-small and CodeGen-350M-Mono memorize **approximately 7%** training samples containing sensitive data, posing considerable privacy risks.

Building upon this finding, we extend our analysis to additional models, *i.e.*, CodeParrot, CodeGen-2B-Mono, and Qwen2.5-Coder-7B. For each studied CLM, we ultimately collect 10,000 highly memorized sensitive samples (*e.g.*, MA $\geq$ 0.9), resulting in **50,000** samples in total. We compile them into a **Sensitive Memorization Dataset**, which documents the positions of all sensitive segments within each code sample along with their corresponding memorization scores. This dataset serves as the foundation for subsequent unlearning experiments, providing a standardized benchmark for evaluation. The overall dataset collection pipeline is illustrated in Figure 3.

# 4 Unlearning Techniques

Our preliminary study reveals that CLMs memorize substantial sensitive data from training corpora. To mitigate this issue, we explore unlearning techniques that enable targeted forgetting of memorized content. We formally define the unlearning problem for CLMs and introduce three gradient ascent-based unlearning approaches: the vanilla method, the constraint-based method, and our proposed CodeEraser.

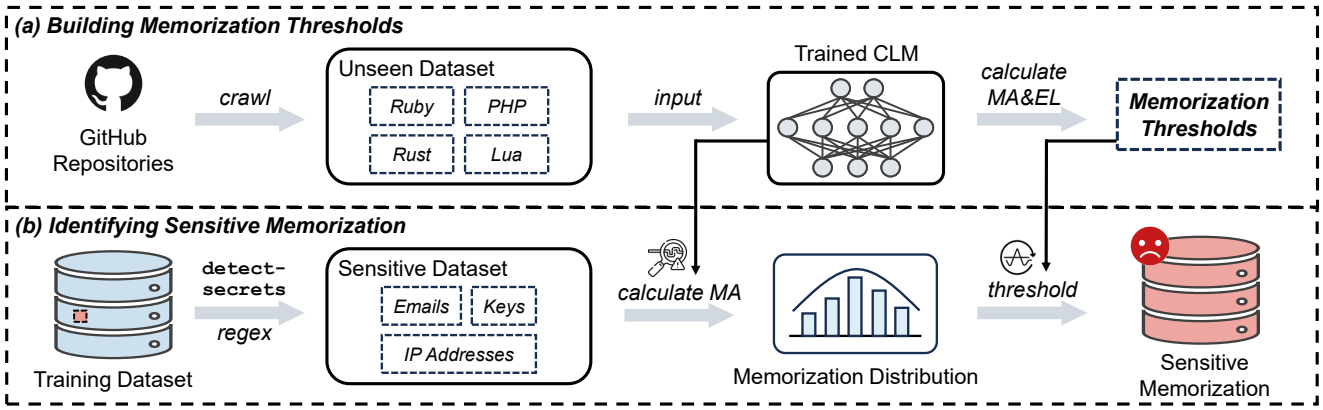


**Figure 3: An illustration of the sensitive memorization detection pipeline.**

## 4.1 Problem Statement

Forgetting, the inverse of memorization, is typically studied in the context of *catastrophic forgetting* [40, 41], where models lose prior knowledge when learning new tasks. These studies treat forgetting as an undesirable trait in training. Recently, Jagielski et al. [36] reinterpret forgetting positively, viewing it as a relaxed form of differential privacy. However, they mainly examine passive forgetting during large-scale training. In contrast, our study embraces an active form of forgetting, *i.e.*, machine unlearning [14], which intentionally modifies trained models to erase previously memorized information. Conceptually, we define forgetting as a reduction in the model's memorization of specific training samples.

Formally, let $f_\theta$ be a CLM trained on a dataset $\mathcal{D}$, and let $\mathcal{D}^f = \{\mathbf{x}^f\} \subset \mathcal{D}$ denote the *forgotten set*, where each forgotten sample $\mathbf{x}^f = (x_1^f, x_2^f, \ldots, x_N^f)$ is a token sequence containing sensitive data. The set size $|\mathcal{D}^f| = k$ represents the number of samples undergoing unlearning simultaneously. The goal of unlearning is to update the CLM to a new version, $f'_\theta$, that no longer retains any information from each $\mathbf{x}^f$. Specifically, after unlearning, each $\mathbf{x}^f$ should satisfy the following conditions, appearing as if never seen by the CLM:

$$\mathrm{MA}(\mathbf{x}^f) \le T_{\mathrm{MA}}\,,\ \mathrm{EL}_n(\mathbf{x}^f) \le T_{\mathrm{EL}_n}\,. \tag{7}$$

## 4.2 Gradient Ascent-Based Unlearning

**Vanilla Unlearning.** *Gradient Ascent* (GA) [13, 37, 45] is a simple yet effective unlearning method designed to reduce the model's likelihood of predicting specific forgotten samples, thereby actively encouraging the removal of their information. Specifically, for each $\mathbf{x}^f$, GA reverses the standard autoregressive language modeling objective by maximizing the negative log-likelihood, forcing the CLM to deviate from its original predictions. Formally, as illustrated in Figure 4 (a), GA updates the unlearned CLM $f'_\theta$ using the following loss function:

$$\mathcal{L}^{GA}(\mathbf{x}^f) = -1 \cdot \mathcal{L}^{LM}(\mathbf{x}^f) = \log \prod_{i=1}^{N} f'_\theta(x_i^f \mid x_1^f, \ldots, x_{i-1}^f)\,. \tag{8}$$

**Constraint-Based Unlearning.** A key challenge in unlearning is to remove targeted data without degrading the model's original utility. Directly applying gradient ascent to the CLM may risk erasing unrelated yet valuable code knowledge. To address this, the *Constraint-Based Unlearning* (CU) method [18, 42, 69, 70] seeks to minimize the *Kullback-Leibler* (KL) divergence between the predictions of the original CLM $f_\theta$ and the unlearned CLM $f'_\theta$ on the data to be retained, while maximizing divergence for the data targeted for forgetting. Formally, given a *retained set* $\mathcal{D}^r = \{(\mathbf{x}^r)\} \subset \mathcal{D}$, the CLM is updated using the following contrastive loss:

$$\mathcal{L}^{KL}(\mathbf{x}^f, \mathbf{x}^r) = -\sum_{\mathbf{x}^f} KL(f_\theta(\mathbf{x}^f) || f'_\theta(\mathbf{x}^f)) + \alpha \cdot \sum_{\mathbf{x}^r} KL(f_\theta(\mathbf{x}^r) || f'_\theta(\mathbf{x}^r))\,, \tag{9}$$

where $\alpha$ is a hyperparameter controlling the balance between forgetting $\mathbf{x}^f$ and retaining $\mathbf{x}^r$. Minimizing KL divergence on retained data ensures alignment with the original predictions, while maximizing it on forgotten data actively encourages effective forgetting. In practice, as illustrated in Figure 4 (b), this KL divergence-based loss $\mathcal{L}^{KL}$ is typically combined with the GA-based loss $\mathcal{L}^{GA}$ for collaborative optimization:

$$\mathcal{L}^{CU} = \mathcal{L}^{GA}(\mathbf{x}^f) + \lambda \cdot \mathcal{L}^{KL}(\mathbf{x}^f, \mathbf{x}^r)\,, \tag{10}$$

where the hyperparameter $\lambda$ balances the intensity between gradient ascent updates and KL divergence-based constraints.

**CodeEraser: Proposed Selective Unlearning.** While techniques like gradient ascent and KL divergence-based constraint enable effective unlearning, they typically indiscriminately forget entire code samples, unnecessarily removing non-sensitive content. Motivated by insights from our preliminary study, we propose CodeEraser, an adapted unlearning method that selectively targets and erases sensitive memorized segments (*e.g.*, API keys) without compromising the integrity and functionality of surrounding code. This segmentation design builds on the tool-based identification of sensitive elements within code (*e.g.*, via `detect-secrets`) described in Section 3.3, enabling accurate and targeted unlearning.

Formally, for each forgotten sample $\mathbf{x}^f = (x_1^f, x_2^f, \ldots, x_N^f) \in \mathcal{D}^f$, we segment sensitive sequences $\mathbf{s}^f = (s_1^f, s_2^f, \ldots, s_m^f)$ from their non-sensitive contexts $\mathbf{c}^f = (c_1^f, c_2^f, \ldots, c_n^f)$. To achieve selective unlearning, we apply gradient ***ascent*** exclusively on $\mathbf{s}^f$ to actively diminish their memorization, while applying gradient ***descent*** on $\mathbf{c}^f$ to preserve their integrity. Accordingly, we apply a targeted KL divergence-based constraint on sensitive segments $\mathbf{s}^f$ rather than

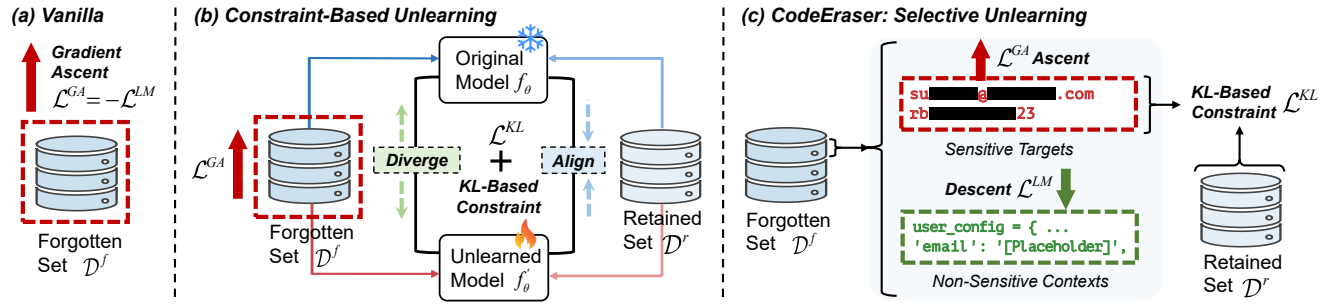


**Figure 4: An illustration of gradient ascent-based unlearning methods: (a) vanilla unlearning, (b) constraint-based unlearning, and (c) our proposed CodeEraser. Personal details are masked for ethical considerations.**

the entire sample $\mathbf{x}^f$. Specifically, as illustrated in Figure 4 (c), we define the selective unlearning loss as follows:

$$\mathcal{L}^{SU} = (\mathcal{L}^{GA}(\mathbf{s}^f) + \gamma \cdot \mathcal{L}^{LM}(\mathbf{c}^f)) + \lambda \cdot \mathcal{L}^{KL}(\mathbf{s}^f, \mathbf{x}^r), \quad (11)$$

where the hyperparameter $\gamma$ balances the trade-off between forgetting $\mathbf{s}^f$ and preserving $\mathbf{c}^f$. This selective framework precisely restricts ascent updates to sensitive segments, minimizing the impact of unlearning on the CLM's broader utility.

Example 4. Given a piece of code snippet "*user_config = {'email': 'daniel@gmail.com', 'password': 'ABC'}*", the sensitive segments $s^f$ are "*daniel@gmail.com*" and "*ABC*", while the non-sensitive contexts $c^f$ are "*user_config = {'email': '[placeholder]', 'password': '[placeholder]'}*". This segmentation preserves the original sequential structure required by autoregressive CLMs.

To stabilize the *max-min* optimization of $\mathcal{L}^{KL}$ during unlearning, we adopt an iterative training strategy. Specifically, we alternate training epochs between the forgotten set $\mathcal{D}^f$ and the retained set $\mathcal{D}^r$. This strategy ensures balanced training dynamics, preventing either term from excessively dominating the optimization process.

### 4.3 Connections and Discussion

Here, we establish connections between our selective unlearning framework and other methods in Figure 4. Specifically, when removing the segmentation between sensitive and non-sensitive parts and setting the hyperparameters $\gamma$ and $\lambda$ in Eq. (11) to 0, the selective unlearning loss collapses directly into the standard gradient ascent loss defined in Eq. (8). In this scenario, the CLM indiscriminately performs gradient ascent on entire code instances rather than selectively targeting sensitive segments. Similarly, by removing segment-level targeting in Eq. (11) and applying the KL divergence-based constraint to entire forgotten samples $\mathbf{x}^f$, the selective unlearning loss reverts to the original constraint-based formulation in Eq. (10). In this case, the constraint-based unlearning considers whole-sample consistency, without explicitly distinguishing between sensitive and non-sensitive segments. These derivations demonstrate that our framework subsumes prior unlearning methods while extending them to support fine-grained, code-specific erasure of sensitive memorized segments.

## 5 Experiments and Analysis

To evaluate the performance of various unlearning techniques for CLMs, we investigate the following *Research Questions* (RQs):

- **RQ1: Effectiveness and Efficiency.** *How do the unlearning methods perform in terms of removing targeted sensitive information from CLMs (effectiveness) with minimal computational resources (efficiency)?*
- **RQ2: Model Utility Post-Unlearning.** *How do the unlearning methods affect the original utility of CLMs, particularly the code generation performance on the HumanEval benchmark?*
- **RQ3: Analysis on Forgotten Data.** *How do the characteristics of forgotten data (e.g., the number of samples k, their occurrence frequency in training, and the types of sensitive segments) impact unlearning performance?*
- **RQ4: Impact of Hyperparameters.** *How do hyperparameter settings (e.g., learning rate, $\gamma$, $\alpha$, and $\lambda$) impact unlearning effects?*

### 5.1 Experimental Setup

**Forgotten Set.** Following [37], we build the forgotten set for each studied CLM by **randomly sampling $k$ instances** from the corresponding sensitive memorization dataset, which are then subjected to unlearning. To reduce the potential bias of random selection, we report the average result from 5 independent runs for each unlearning routine. Unless otherwise specified, we report experimental results with a default setting of $k = 32$. We examine various values of $k$ from $\{8, 16, 32, 64, 128, 256, 512\}$, as detailed in Section 5.4, to demonstrate CodeEraser's scalability in handling varying numbers of unlearning requests from users.

**Retained Set.** The retained set is built to include non-targeted, non-sensitive data, serving as a benchmark for measuring the CLM's memorization retention after the unlearning process. We leverage a code benchmark [3] that offers 1,000 non-sensitive samples from BigQuery [31]. These samples have been demonstrated to be memorized by various CLMs, such as CodeParrot [24], CodeGen [50], and InCoder [28], making them suitable for building the retained set in our experiments. Specifically, for each studied CLM, we extract its corresponding memorized data and **randomly sample an equivalent number of $k$ instances** to form the retained set.

**Implementation Details.** In our experiments, the maximum token lengths are set to 512 for the forgotten set and 128 for the unseen dataset and the retained set, with any excess truncated. These lengths are chosen based on computational constraints while ensuring sufficient data is available for analysis. For computing MA and $EL_n$ scores, we adopt a greedy sampling strategy. Following [37], we set the global batch size equal to $k$ during unlearning. When

**Table 3: Evaluation of unlearning effectiveness. All values are reported as percentages (with % symbol omitted).**

| CLM | Method | MA | $EL_3$ | $EL_5$ | $EL_{10}$ | Red. |
|---|---|---|---|---|---|---|
| ***CodeParrot -small*** | Original | 99.74 | 98.55 | 98.12 | 97.97 | - |
| | *Threshold* | *45.57* | *17.66* | *10.82* | *5.49* | - |
| | GA | 30.71 | 10.17 | 7.07 | 4.18 | 86.85 |
| | CU | 22.14 | 7.79 | 6.19 | 4.72 | 89.69 |
| | CodeEraser | 18.69 | 6.69 | 5.78 | 5.18 | 90.82 |
| ***CodeParrot*** | Original | 99.69 | 98.90 | 98.36 | 97.62 | - |
| | *Threshold* | *46.34* | *16.56* | *10.17* | *5.14* | - |
| | GA | 27.53 | 6.33 | 4.21 | 3.47 | 89.54 |
| | CU | 24.18 | 6.11 | 4.39 | 3.09 | 90.48 |
| | CodeEraser | 15.22 | 6.36 | 5.40 | 4.65 | 92.01 |
| ***CodeGen -350M-Mono*** | Original | 99.25 | 97.14 | 96.39 | 95.93 | - |
| | *Threshold* | *48.79* | *18.24* | *11.03* | *5.92* | - |
| | GA | 25.53 | 8.45 | 6.98 | 4.95 | 88.29 |
| | CU | 18.65 | 6.98 | 5.73 | 4.88 | 90.75 |
| | CodeEraser | 45.13 | 11.44 | 7.05 | 3.46 | 82.96 |
| ***CodeGen -2B-Mono*** | Original | 99.89 | 99.79 | 99.76 | 99.70 | - |
| | *Threshold* | *53.61* | *19.32* | *11.71* | *6.28* | - |
| | GA | 17.95 | 6.80 | 5.52 | 4.83 | 91.21 |
| | CU | 11.80 | 6.40 | 5.99 | 5.54 | 92.55 |
| | CodeEraser | 31.66 | 10.01 | 7.73 | 6.05 | 86.11 |
| ***Qwen2.5 -Coder-7B*** | Original | 96.26 | 84.71 | 81.07 | 75.15 | - |
| | *Threshold* | *40.99* | *15.65* | *12.45* | *8.82* | - |
| | GA | 24.15 | 14.23 | 10.49 | 8.24 | 83.55 |
| | CU | 16.63 | 8.77 | 6.84 | 5.48 | 89.16 |
| | CodeEraser | 8.49 | 4.93 | 3.99 | 3.68 | 93.89 |

processing a group of $k$ instances, we average their MA and $EL_n$ scores to empirically decide whether they have been forgotten. The learning rate is fixed at 3e-6, selected through empirical testing from the range {1e-5, 8e-6, 5e-6, 3e-6, 1e-6}, and we maintain a constant learning rate schedule throughout unlearning. Dropout and weight decay rates are both set to 0 to avoid regularization that might interfere with the unlearning process. We select $\alpha = 1.0$ from {0.5, 0.8, 1.0, 1.2, 1.5}, and $\gamma = 0.5$ and $\lambda = 0.1$ from {0.1, 0.2, 0.3, 0.4, 0.5} through a systematic grid search, with detailed hyperparameter analysis in Section 5.5.

## 5.2 RQ1: Effectiveness and Efficiency

We assess the effectiveness and efficiency of various unlearning techniques when applied to the studied CLMs. In our context, effectiveness refers to the capability of the unlearning approach to successfully erase specific sensitive information retained by the CLM, while efficiency refers to the computational costs required to achieve this unlearning.

**Unlearning Effectiveness.** To quantitatively evaluate the effectiveness of unlearning, we calculate MA, $EL_3$, $EL_5$, and $EL_{10}$ for the sensitive data targeted for removal in the forgotten set. An unlearning approach is deemed effective when the targeted sensitive data becomes difficult to extract from the model post-unlearning, characterized by MA and $EL_n$ scores falling below their respective **memorization thresholds** defined in Section 3.2.2. We also report the average memorization **reduction** rate of these metrics post-unlearning (abbreviated as *Red.*).

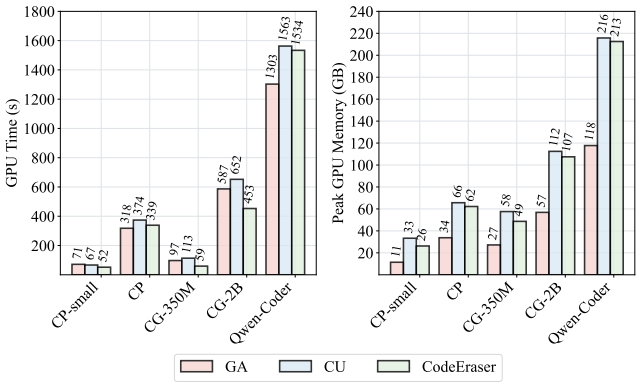


**Figure 5: Evaluation of unlearning efficiency.**

As shown in Table 3, the results indicate that CodeEraser achieves a substantial reduction in MA and $EL_n$ scores across all models, successfully lowering them below the predetermined memorization thresholds outlined in Table 2. For instance, with the Qwen2.5-Coder-7B model, CodeEraser results in an average memorization reduction of **93.89%**. It is important to note that an unlearning method is considered effective as long as it reaches the forgetting criteria; it is not required to reduce more memorization than the baselines (*i.e.*, GA and CU), as over-unlearning could lead to a loss of model utility.

**Unlearning Efficiency.** For efficiency, we measure the cumulative GPU time required to perform unlearning updates on the CLM until the memorization thresholds are reached for a group of $k = 32$ instances. Additionally, we monitor peak memory usage across 4 GPUs during unlearning by leveraging PyTorch's memory check API, and report the total footprint as the sum of these values. These two metrics are chosen because they directly reflect the computational resources consumed during unlearning.

As shown in Figure 5, with the Qwen2.5-Coder-7B model, our proposed CodeEraser completes the unlearning process within approximately 1500 seconds (averaging **46.88** seconds per sample), with a peak memory usage of around 200GB. This cost is considerably lower than alternatives such as differentially-private training or retraining the CLM after de-duplication, which are reported to typically require on the order of hundreds of A100 GPU days [37]. Moreover, unlike the baselines that focus on the unlearning of entire code samples, CodeEraser exclusively targets the forgetting of specific sensitive data, enabling it to complete unlearning in a relatively shorter duration. Although CodeEraser exhibits higher memory usage than GA (due to additional training steps on the retained set), it outperforms in terms of preserving the post-unlearning utility of CLMs, which will be further discussed in Section 5.3.

**Answer to RQ1:** CodeEraser demonstrates effectiveness and efficiency in erasing specific sensitive information from CLMs, thereby reducing potential security and privacy risks without incurring excessive computational costs.

**Table 4: Evaluation of model utility post-unlearning. All values are reported as percentages (with % symbol omitted). ↑ indicates that higher values correspond to better preservation of model utility. The best-performing unlearning method in each column is highlighted in bold.**

| CLM | Method | P@1↑ | P@5↑ | P@10↑ | Ret.↑ |
|---|---|---|---|---|---|
| ***CodeParrot -small*** | Original | 3.48 | 4.56 | 4.96 | - |
| | GA | 2.14 | 3.02 | 3.20 | 64.08 |
| | CU | 2.62 | 3.43 | 3.66 | 74.77 |
| | CodeEraser | **3.74** | **4.59** | **4.87** | **102.10** |
| ***CodeParrot*** | Original | 4.34 | 5.81 | 6.22 | - |
| | GA | 2.08 | 3.29 | 3.83 | 55.38 |
| | CU | 2.04 | 2.94 | 3.28 | 50.11 |
| | CodeEraser | **3.86** | **5.08** | **5.63** | **88.96** |
| ***CodeGen -350M-Mono*** | Original | 13.37 | 18.79 | 21.12 | - |
| | GA | 11.68 | 16.59 | 18.51 | 87.76 |
| | CU | 10.79 | 14.91 | 16.41 | 79.25 |
| | CodeEraser | **13.36** | **18.02** | **19.96** | **96.78** |
| ***CodeGen -2B-Mono*** | Original | 24.72 | 31.49 | 34.16 | - |
| | GA | 21.20 | 28.34 | 31.40 | 89.23 |
| | CU | 20.57 | 27.73 | 30.63 | 86.98 |
| | CodeEraser | **23.00** | **29.91** | **32.94** | **94.82** |
| ***Qwen2.5 -Coder-7B*** | Original | 61.07 | 73.61 | 77.23 | - |
| | GA | 40.67 | 53.81 | 57.63 | 71.44 |
| | CU | 48.54 | 64.70 | 69.59 | 85.83 |
| | CodeEraser | **61.65** | **73.41** | **76.69** | **99.99** |

## 5.3 RQ2: Model Utility Post-Unlearning

Ensuring robust privacy protections necessitates a delicate balance: the CLM must selectively forget targeted sensitive information to safeguard privacy without compromising its inherent capacity to perform general coding tasks. We evaluate the efficacy of CodeEraser in achieving this balance.

**Setup.** To evaluate the impact of unlearning on the CLM's utility, we adopt the HumanEval benchmark [19], a widely recognized standard for assessing code generation performance in CLMs [10], with over 95.9k monthly downloads on HuggingFace at the time of writing. This benchmark measures the CLM's ability to solve programming tasks, where we report the Pass@1, Pass@5, and Pass@10 scores [19], which measure the accuracy of generating correct solutions within 1, 5, and 10 attempts for each task, respectively (abbreviated as *P@1*, *P@5*, and *P@10*). By comparing these scores pre- and post-unlearning, we can observe the changes in the CLM's general coding performance. To quantify these changes, we also report the average performance **retention** rate across these metrics post-unlearning (abbreviated as *Ret.*).

**Results.** As shown in Table 4, CodeEraser has only a minor impact on model utility compared to the baselines. Take Qwen2.5-Coder-7B as an example, CodeEraser preserves **99.99%** of the CLM's code generation performance. This lesser degree of degradation can be attributed to CodeEraser's sensitive information-targeted selective unlearning mechanism, which minimizes the impact of unlearning on model utility, ensuring that the code knowledge outside the specified forgotten set remains intact.

Among the baselines, a notable performance decline is observed in most cases when applying GA to the studied CLMs. This decline may stem from the gradient ascent updates, which, although performed only on the forgotten set, tend to *soften* the probability distribution of generating each token across the vocabulary. This results in a more uniform distribution, which inadvertently dilutes the CLM's inherent knowledge base and reduces its overall utility. Moreover, the CU approach does not demonstrate the expected level of utility preservation compared to GA in some cases. This may be due to the alignment of model behavior on shorter instances (128 tokens) being insufficient to offset the impact of forgetting longer instances (512 tokens). Instead, it could affect the stability of the model's updates, further compromising the integrity of the CLM's knowledge base. Given this unexpected phenomenon, we plan to investigate it further in future work to fully understand the dynamics of unlearning in CLMs.

**Answer to RQ2:** CodeEraser has only a minor impact on the CLM's coding performance compared to baselines, validating the efficacy of our selective unlearning mechanism in preserving model utility while achieving targeted forgetting of sensitive information in code.

## 5.4 RQ3: Analysis on Forgotten Data

Recent studies have highlighted the importance of training data characteristics, such as duplication frequency and sensitive data type, in influencing memorization patterns of CLMs [3, 15, 68]. These findings reveal the intricate nature of memorization in CLMs and imply a potential impact of such data attributes on the efficacy of unlearning. To examine this, we evaluate CodeEraser on the CodeParrot model, focusing on targeted sensitive data samples that vary in number, duplication frequency, and type. For each setting, we report the average results of the HumanEval scores pre- and post-unlearning in 5 independent runs.

**Influence of Forgotten Sample Number $k$.** Our analysis examines how varying the number of forgotten samples ($k$) influences the efficacy of CodeEraser. As shown in Figure 6 (a), CodeEraser remains robust when unlearning a moderate number of sensitive samples (*e.g.*, $k \leq 128$), with the CLM effectively preserving its utility post-unlearning. However, as the size of the forgotten set increases significantly (*e.g.*, $k = 256$ or $k = 512$), utility scores such as P@5 and P@10 exhibit a noticeable decline. These results indicate potential scalability limitations in CodeEraser, particularly for larger-scale unlearning tasks involving extensive sensitive datasets (*e.g.*, 10,000 samples). Ensuring effective unlearning at scale while minimizing utility degradation remains a key challenge, which we leave for future work.

**Influence of Data Duplication.** To examine the impact of data duplications on unlearning, we utilize the duplication frequency statistics of training samples provided by the `codeparrot-clean-train` dataset. We roughly divide duplication frequencies into four ranges: $[5, 10)$, $[10, 25)$, $[25, 50)$, and $[50, )$, which allows us to assess how varying levels of data duplication, from relatively low to very high frequencies, affect the unlearning process. For each duplication level, we randomly select $k = 16$ samples that contain sensitive information to perform unlearning.

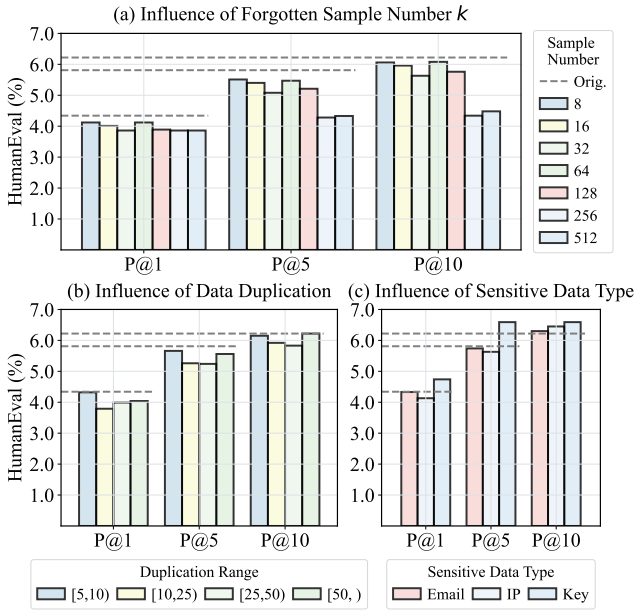


**Figure 6: Analysis on forgotten data. Dashed lines "- -" represent the initial HumanEval scores of the CodeParrot model.**

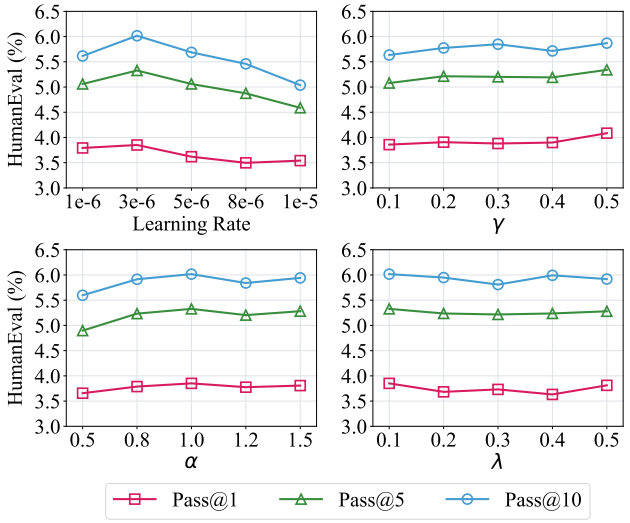


**Figure 7: Parameter analysis of learning rate, $\gamma$, $\alpha$, and $\lambda$.**

As shown in Figure 6 (b), the frequency of duplication significantly influences the CLM's utility post-unlearning. Interestingly, we can see that CodeEraser exhibits higher utility-preserving performance at the extremes of the duplication spectrum (*i.e.*, $[5, 10)$ and $[50, ))$ compared to the intermediate duplication levels. This phenomenon may be explained by the nature of the data involved. Low-duplicated memorized samples, potentially acting as outliers within the data distribution [16], may have a less entrenched influence on the model, making their removal less disruptive. On the other hand, highly duplicated samples are likely to cause model overfitting, meaning that their removal could reduce redundancy and mitigate overfitting, resulting in a minor impact on overall model performance. These findings suggest that the impact of unlearning on model utility is not uniform across different levels of data duplication, and understanding these dynamics is crucial for optimizing our unlearning approach in the future.

**Influence of Sensitive Data Type.** To evaluate the influence of distinct sensitive data types (*e.g.*, email, IP address, and API/SSH Key) on the unlearning process, we leverage the constructed sensitive memorization dataset for the CodeParrot model. For each type, we randomly select $k = 32$ samples containing only the corresponding sensitive data for unlearning.

As shown in Figure 6 (c), the influence on the CLM's utility post-unlearning varies depending on the type of sensitive data. This variation is likely due to differences in how the CLM memorizes these data types. Surprisingly, the removal of API/SSH keys results in an improvement in model performance. This improvement may be attributed to the fact that, unlike emails and IP addresses, specific secret keys are usually atypical patterns within the data and are less likely to be heavily duplicated in the training dataset, making them outliers in the data distribution. Such atypical data outliers often distract the model and negatively impact its overall generalization. Therefore, eliminating them can refine the CLM's representation space and shift its focus to more representative data, thereby enhancing overall performance. Given these preliminary insights, future work will explore the effects of unlearning across a broader spectrum of sensitive data types.

**Answer to RQ3:** The characteristics of the targeted sensitive data, *e.g.*, number, duplication frequency, and type, significantly influence the CLM's utility post-unlearning. These findings reveal that unlearning effects are conditioned by data attributes, motivating future exploration of unlearning dynamics and robust strategies to minimize negative impacts on models.

## 5.5 RQ4: Impact of Hyperparameters

We analyze the impact of hyperparameters, including the learning rate and regularization factors (*i.e.*, $\gamma$, $\alpha$, and $\lambda$), on model utility post-unlearning. As illustrated in Figure 7, the learning rate substantially influences model utility after unlearning, with excessively large values resulting in noticeable declines in utility metrics (*e.g.*, P@5 and P@10). This underscores the importance of carefully tuning the learning rate to balance forgetting effectiveness against maintaining model performance. In contrast, varying the hyperparameters $\gamma$, $\alpha$, and $\lambda$ within reasonable ranges results in only minor changes in post-unlearning utility, indicating robustness and flexibility of our method toward these parameters. Nonetheless, moderate values for these parameters are recommended, as overly aggressive settings may still negatively impact utility or insufficiently support effective forgetting. Based on these insights, we select empirically determined optimal values for all hyperparameters to balance the trade-off between effective forgetting and model performance retention. The final settings employed in our experiments are detailed in Section 5.1.

**Answer to RQ4:** The learning rate substantially influences model utility after unlearning and must be carefully tuned. Regularization hyperparameters ($\gamma$, $\alpha$, $\lambda$) have comparatively minor impacts, allowing greater flexibility in their selection.

## 6 Threats to Validity

**Threats to Internal Validity.** Internal validity concerns whether our methodology introduces biases or errors that may distort the results. Our study identifies sensitive segments within code using a regular expression-based method and then quantifies their memorization. However, this method constrains both the accuracy and coverage of secret detection, since regex rules can capture only a limited set of patterns. To mitigate this threat, we employ `detect-secrets` [71], a state-of-the-art tool widely used for secret detection in large-scale code bases. This tool covers a broad spectrum of high-risk categories (*e.g.*, API keys, tokens, and credentials) that are most relevant to real-world security incidents. Future work may incorporate additional detection methods [7, 27, 33] to broaden coverage; however, such extensions are unlikely to alter our principal finding that CLMs manifest substantial sensitive memorization.

**Threats to External Validity.** External validity concerns the extent to which our findings can be generalized to other settings. This study focuses on three CLM families, *i.e.*, CodeParrot, CodeGen, and Qwen2.5-Code, spanning 110M to 7B parameters; however, the results may not generalize to other CLMs. To alleviate this threat, we select these families since they are widely adopted in research and practice, making them representative of the current CLM landscape. Moreover, the observed patterns are consistent across different model sizes within these families, suggesting that our findings are not tied to a specific scale. This limitation is also shared by many prior studies, which typically examine a few representative families rather than exhaustively covering all models. Thus, while additional CLM families might provide further evidence, our methodological choices sufficiently support external validity.

## 7 Related Work

**Memorization in LMs.** Memorization, often seen as the antithesis of generalization, arises from overfitting, leading models to remember specific details of their training data [3, 25]. This phenomenon raises remarkable privacy concerns in the context of LMs, as these models may memorize and regurgitate sensitive information verbatim. Extensive research has been undertaken to understand memorization in LMs qualitatively and quantitatively [8, 15–17, 55, 56, 72]. Recent research [3, 35, 68] has also explored memorization within CLMs, offering empirical studies to examine the extent to which CLMs inadvertently memorize and disclose their training data. Additionally, recent studies [33, 51] have highlighted privacy risks by extracting sensitive information from CLMs using well-crafted prompts. Following this line, in this paper, we conduct a pioneering investigation into mitigating sensitive memorization in CLMs through machine unlearning.

**Machine Unlearning.** Machine unlearning, first proposed by Cao et al. [14], also known as *selective forgetting* [30] or *data removal/deletion* [29, 32], aims to remove the influence of a specific set of training samples from the trained model. Existing studies in this field can be categorized into two groups: *1) Exact Unlearning:* Exact unlearning seeks to remove specific samples' influence from the model completely. A straightforward method is to retrain the whole model from scratch after removing targeted data from the training set. However, this method is computationally infeasible for large datasets. Despite efforts to reduce the computational cost, they either primarily cater to simple machine learning models [14, 29] or rely on training data partitioning [11, 20], limiting their applicability to complex and large CLMs. *2) Approximate Unlearning:* Approximate unlearning has recently emerged as a promising alternative, prioritizing efficiency and cost by relaxing the requirement for exactness. Existing methods typically adjust the model's weights via gradient-based updates to approximate the weights of the model retrained from scratch [23, 30, 32, 37]. Building on this paradigm, gradient ascent-based methods [18, 37, 69] have emerged as a dominant direction for efficient unlearning by reversing the learning of specific data, which also constitutes the focus of this study. However, they often indiscriminately erase entire text instances rather than selectively targeting specific sensitive data (*e.g.*, API keys embedded in code). While Wang et al. [64] proposed a heuristic that designates high-perplexity tokens in plain text as privacy tokens for unlearning, this approach is unsuitable for source code: identifiers are often assigned unique names with high perplexity, resulting in erroneous removal, whereas actual secrets such as API key strings typically follow predictable patterns with lower perplexity and may therefore escape removal. In contrast, our approach employs a specialized tool (*i.e.*, detect-secrets) to precisely identify secrets in code, enabling targeted unlearning while preserving the integrity and functionality of the surrounding code.

## 8 Conclusion

In this paper, we pioneer the use of machine unlearning to erase sensitive memorization in CLMs. We first construct a novel dataset by systematically identifying and assessing high-risk code instances in the CLM's training data. Then, we introduce CodeEraser, a selective unlearning approach that uses gradient ascent to remove sensitive information while preserving surrounding non-sensitive code via gradient descent. Additionally, CodeEraser employs a KL divergence-based constraint to maintain model utility post-unlearning. Extensive experiments on CodeParrot, CodeGen-Mono, and Qwen2.5-Coder demonstrate that CodeEraser effectively eliminates sensitive memorization while preserving overall model performance. Our study highlights the potential of unlearning in reinforcing data privacy in CLMs, providing a practical technique to actively mitigate the harms of model memorization.

**Data Availability.** All the experimental data and code used in this paper are available at https://github.com/CGCL-codes/naturalcc/tree/main/examples/code-unlearning.

## Acknowledgment

This work is supported by the Major Program (JD) of Hubei Province (Grant No. 2023BAA024). We would like to thank all the anonymous reviewers for their insightful comments.

## References


[1] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 308–318.

[2] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. 2023. Extending Source Code Pre-Trained Language Models to Summarise Decompiled Binaries. In *Proceedings of the 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '23)*. 260–271.

[3] Ali Al-Kaswan, Maliheh Izadi, and Arie van Deursen. 2024. Traces of Memorisation in Large Language Models for Code. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. IEEE Computer Society, Los Alamitos, CA, USA, 862–862.

[4] Rohan Anil, Badih Ghazi, Vineet Gupta, Ravi Kumar, and Pasin Manurangsi. 2022. Large-Scale Differentially Private BERT. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP '22)*. Association for Computational Linguistics, Abu Dhabi, United Arab Emirates.

[5] Shushan Arakelyan, Rocktim Das, Yi Mao, and Xiang Ren. 2023. Exploring Distributional Shifts in Large Language Models for Code Analysis. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP '23)*. Association for Computational Linguistics, Singapore, 16298–16314.

[6] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).

[7] Setu Kumar Basak, Lorenzo Neil, Bradley Reaves, and Laurie Williams. 2023. SecretBench: A Dataset of Software Secrets. In *Proceedings of the 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR '23)*. 347–351.

[8] Stella Biderman, Usvsn Sai Prashanth, Lintang Sutawika, Hailey Schoelkopf, Quentin Anthony, Shivanshu Purohit, and Edward Raff. 2023. Emergent and Predictable Memorization in Large Language Models. In *Proceedings of the 37th Annual Conference on Neural Information Processing Systems (NeurIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1219, 19 pages.

[9] Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, Usvsn Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar Van Der Wal. 2023. Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling. In *Proceedings of the 40th International Conference on Machine Learning (ICML '23)*. JMLR.org, Article 102, 34 pages.

[10] Bigcode. 2024. Big Code Models Leaderboard. https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard. Accessed: 2025-09-01.

[11] Lucas Bourtoule, Varun Chandrasekaran, Christopher A. Choquette-Choo, Hengrui Jia, Adelin Travers, Baiwu Zhang, David Lie, and Nicolas Papernot. 2021. Machine Unlearning. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP '21)*. 141–159.

[12] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Proceedings of the 34th Annual Conference on Neural Information Processing Systems (NeurIPS '20)*. Curran Associates, Inc., Red Hook, NY, USA, Article 159, 1877–1901 pages.

[13] George-Octavian Bărbulescu and Peter Triantafillou. 2024. To Each (Textual Sequence) Its Own: Improving Memorized-Data Unlearning in Large Language Models. In *Proceedings of the 41st International Conference on Machine Learning (ICML '24)*. Article 121, 21 pages.

[14] Yinzhi Cao and Junfeng Yang. 2015. Towards Making Systems Forget with Machine Unlearning. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, USA, 463–480.

[15] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramèr, and Chiyuan Zhang. 2023. Quantifying Memorization Across Neural Language Models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*.

[16] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. 2019. The Secret Sharer: Evaluating and Testing Unintended Memorization in Neural Networks. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC '19)*. USENIX Association, USA, 267–284.

[17] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting Training Data from Large Language Models. In *Proceedings of the 30th USENIX Conference on Security Symposium (SEC '21)*. USENIX Association, 2633–2650.

[18] Jiaao Chen and Diyi Yang. 2023. Unlearn What You Want to Forget: Efficient Unlearning for LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP '23)*. Association for Computational Linguistics, Singapore, 12041–12052.

[19] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremb. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[20] Min Chen, Zhikun Zhang, Tianhao Wang, Michael Backes, Mathias Humbert, and Yang Zhang. 2022. Graph Unlearning. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 499–513.

[21] CodeParrot. 2022. Codeparrot-clean-train. https://huggingface.co/datasets/codeparrot/codeparrot-clean-train. Accessed: 2025-09-01.

[22] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 81, 13 pages.

[23] Ronen Eldan and Mark Russinovich. 2023. Who's Harry Potter? Approximate Unlearning in LLMs.

[24] Hugging Face. 2022. CodeParrot. https://huggingface.co/codeparrot. Accessed: 2025-09-01.

[25] Vitaly Feldman. 2020. Does Learning Require Memorization? A Short Tale about a Long Tail. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC '20)*. Association for Computing Machinery, New York, NY, USA, 954–959.

[26] Vitaly Feldman and Chiyuan Zhang. 2020. What Neural Networks Memorize and Why: Discovering the Long Tail via Influence Estimation. In *Proceedings of the 34th Annual Conference on Neural Information Processing Systems (NeurIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 242, 11 pages.

[27] Runhan Feng, Ziyang Yan, Shiyan Peng, and Yuanyuan Zhang. 2022. Automated Detection of Password Leakage from Public GitHub Repositories. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 175–186.

[28] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*.

[29] Antonio A. Ginart, Melody Y. Guan, Gregory Valiant, and James Zou. 2019. Making AI Forget You: Data Deletion in Machine Learning. In *Proceedings of the 33rd Annual Conference on Neural Information Processing Systems (NeurIPS '19)*. Curran Associates Inc., Red Hook, NY, USA, Article 316, 14 pages.

[30] Aditya Golatkar, Alessandro Achille, and Stefano Soatto. 2020. Eternal Sunshine of the Spotless Net: Selective Forgetting in Deep Networks. In *Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR '20)*. 9301–9309.

[31] Google. 2016. BigQuery. https://console.cloud.google.com/marketplace/details/github/github-repos. Accessed: 2025-09-01.

[32] Chuan Guo, Tom Goldstein, Awni Hannun, and Laurens Van Der Maaten. 2020. Certified Data Removal from Machine Learning Models. In *Proceedings of the 37th International Conference on Machine Learning (ICML '20)*. JMLR.org, Article 359, 11 pages.

[33] 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. *Proceedings of the ACM on Software Engineering* 1, FSE, Article 111 (2024), 23 pages.

[34] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).

[35] Daniel Huynh. 2023. StarCoder Memorization Experiment Highlights Privacy Risks of Fine-Tuning On Code. https://huggingface.co/blog/dhuynh95/starcoder-memorization-experiment. Accessed: 2025-09-01.

[36] Matthew Jagielski, Om Thakkar, Florian Tramer, Daphne Ippolito, Katherine Lee, Nicholas Carlini, Eric Wallace, Shuang Song, Abhradeep Guha Thakurta, Nicolas Papernot, and Chiyuan Zhang. 2023. Measuring Forgetting of Memorized Training Examples. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*.

[37] Joel Jang, Dongkeun Yoon, Sohee Yang, Sungmin Cha, Moontae Lee, Lajanugen Logeswaran, and Minjoon Seo. 2023. Knowledge Unlearning for Mitigating Privacy Risks in Language Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL '23)*. Association for Computational Linguistics, Toronto, Canada, 14389–14408.

[38] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-Planning Code Generation with Large Language Models. *ACM Transactions on Software Engineering and Methodology* 33, 7, Article 182 (2024), 30 pages.

[39] Nikhil Kandpal, Eric Wallace, and Colin Raffel. 2022. Deduplicating Training Data Mitigates Privacy Risks in Language Models. In *Proceedings of the 39th International Conference on Machine Learning (ICML '22)*. PMLR, 10697–10707.

[40] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler L. Hayes, and Christopher Kanan. 2018. Measuring Catastrophic Forgetting in Neural Networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence (AAAI '18/IAAI '18/EAAI '18)*. AAAI Press, Article 415, 9 pages.

[41] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. 2017. Overcoming Catastrophic Forgetting in Neural Networks. *Proceedings of the National Academy of Sciences* 114, 13 (2017), 3521–3526.

[42] Meghdad Kurmanji, Peter Triantafillou, Jamie Hayes, and Eleni Triantafillou. 2023. Towards Unbounded Machine Unlearning. In *Proceedings of the 37th Annual Conference on Neural Information Processing Systems (NeurIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 95, 31 pages.

[43] Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. 2022. Deduplicating Training Data Makes Language Models Better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL '22)*. Association for Computational Linguistics, Dublin, Ireland, 8424–8445.

[44] Gen Li, Yao Wan, Hongyu Zhang, Zhou Zhao, Wenbin Jiang, Xuanhua Shi, Hai Jin, and Zheng Wang. 2025. Dataflow-Guided Neuro-Symbolic Language Models for Type Inference. In *Proceedings of the 42nd International Conference on Machine Learning (ICML '25)*.

[45] Bo Liu, Qiang Liu, and Peter Stone. 2022. Continual Learning and Private Unlearning. In *Proceedings of The 1st Conference on Lifelong Learning Agents*, Vol. 199. PMLR, 243–254.

[46] Michael Meli, Matthew R. McNiece, and Bradley Reaves. 2019. How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS '19)*. The Internet Society.

[47] Tomás Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. 2010. Recurrent Neural Network Based Language Model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association (INTERSPEECH '10)*. ISCA, 1045–1048.

[48] Thanh Tam Nguyen, Thanh Trung Huynh, Zhao Ren, Phi Le Nguyen, Alan Wee-Chung Liew, Hongzhi Yin, and Quoc Viet Hung Nguyen. 2025. A Survey of Machine Unlearning. *ACM Transactions on Intelligent Systems and Technology* (2025).

[49] Yuqing Nie, Chong Wang, Kailong Wang, Guoai Xu, Guosheng Xu, and Haoyu Wang. 2025. Decoding Secret Memorization in Code LLMs Through Token-Level Characterization. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE '25)*. IEEE, 2880–2892.

[50] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23)*.

[51] Liang Niu, Shujaat Mirza, Zayd Maradni, and Christina Pöpper. 2023. CodexLeaks: Privacy Leaks from Code Generation Language Models in GitHub Copilot. In *Proceedings of the 32nd USENIX Conference on Security Symposium (SEC '23)*. USENIX Association, Anaheim, CA, USA, Article 120, 18 pages.

[52] State of California. 2023. California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa. Accessed: 2025-09-01.

[53] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. *OpenAI Blog* 1, 8 (2019), 9.

[54] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2023).

[55] Ali Satvaty, Suzan Verberne, and Fatih Turkmen. 2024. Undesirable Memorization in Large Language Models: A Survey. *arXiv preprint arXiv:2410.02650* (2024).

[56] Kushal Tirumala, Aram H. Markosyan, Luke Zettlemoyer, and Armen Aghajanyan. 2022. Memorization Without Overfitting: Analyzing the Training Dynamics of Large Language Models. In *Proceedings of the 36th Annual Conference on Neural Information Processing Systems (NeurIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 2773, 17 pages.

[57] European Union. 2018. General Data Protection Regulation (GDPR). https://gdpr-info.eu. Accessed: 2025-09-01.

[58] Michael Veale, Reuben Binns, and Lilian Edwards. 2018. Algorithms that Remember: Model Inversion Attacks and Data Protection Law. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 376, 2133 (2018), 20180083.

[59] Eduard Fosch Villaronga, Peter Kieseberg, and Tiffany Li. 2018. Humans Forget, Machines Remember: Artificial Intelligence and the Right to Be Forgotten. *Computer Law & Security Review* 34, 2 (2018), 304–313.

[60] Yao Wan, Zhangqian Bi, Yang He, Jianguo Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Hai Jin, and Philip Yu. 2024. Deep Learning for Code Intelligence: Survey, Benchmark and Toolkit. *ACM Comput. Surv.* 56, 12, Article 309 (2024), 41 pages.

[61] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What Do They Capture? A Structural Analysis of Pre-Trained Language Models for Source Code. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2377–2388.

[62] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. Association for Computing Machinery, New York, NY, USA, 397–407.

[63] Chenlong Wang, Zhaoyang Chu, Zhengxiang Cheng, Xuyi Yang, Kaiyue Qiu, Yao Wan, Zhou Zhao, Xuanhua Shi, Hai Jin, and Dongping Chen. 2025. CodeSync: Synchronizing Large Language Models with Dynamic Code Evolution at Scale. In *Proceedings of the 42nd International Conference on Machine Learning (ICML '25)*.

[64] Lingzhi Wang, Xingshan Zeng, Jinsong Guo, Kam-Fai Wong, and Georg Gottlob. 2025. Selective Forgetting: Advancing Machine Unlearning Techniques and Evaluation in Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence* 39, 1 (2025), 843–851.

[65] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, 1482–1494.

[66] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*. Association for Computing Machinery, New York, NY, USA, 819–831.

[67] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS '22)*. Association for Computing Machinery, New York, NY, USA, 1–10.

[68] 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 (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 72, 13 pages.

[69] Jin Yao, Eli Chien, Minxin Du, Xinyao Niu, Tianhao Wang, Zezhou Cheng, and Xiang Yue. 2024. Machine Unlearning of Pre-trained Large Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL '24)*. Association for Computational Linguistics, Bangkok, Thailand, 8403–8419.

[70] Yuanshun Yao, Xiaojun Xu, and YangLiu. 2024. Large Language Model Unlearning. In *Proceedings of the 38th Annual Conference on Neural Information Processing Systems (NeurIPS '24)*. Curran Associates, Inc., 105425–105475.

[71] Yelp. 2024. Detect-secrets. https://github.com/Yelp/detect-secrets. Accessed: 2025-09-01.

[72] Chiyuan Zhang, Daphne Ippolito, Katherine Lee, Matthew Jagielski, Florian Tramèr, and Nicholas Carlini. 2023. Counterfactual Memorization in Neural Language Models. In *Proceedings of the 37th Annual Conference on Neural Information Processing Systems (NeurIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1708, 42 pages.

[73] Tianqing Zhu, Gang Li, Wanlei Zhou, and Philip S. Yu. 2017. Differentially Private Data Publishing and Analysis: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 29, 8 (2017), 1619–1638.

[74] Tianqing Zhu, Dayong Ye, Wei Wang, Wanlei Zhou, and Philip S. Yu. 2022. More Than Privacy: Applying Differential Privacy in Key Areas of Artificial Intelligence. *IEEE Transactions on Knowledge and Data Engineering* 34, 6 (2022), 2824–2843.