Traces of Memorisation in Large Language Models for Code

Ali Al-Kaswan
a.al-kaswan@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Maliheh Izadi
m.izadi@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Arie van Deursen
arie.vandeursen@tudelft.nl
Delft University of Technology
Delft, The Netherlands

ABSTRACT

Large language models have gained significant popularity because of their ability to generate human-like text and potential applications in various fields, such as Software Engineering. Large language models for code are commonly trained on large unsanitised corpora of source code scraped from the internet. The content of these datasets is memorised and can be extracted by attackers with data extraction attacks. In this work, we explore memorisation in large language models for code and compare the rate of memorisation with large language models trained on natural language. We adopt an existing benchmark for natural language and construct a benchmark for code by identifying samples that are vulnerable to attack. We run both benchmarks against a variety of models, and perform a data extraction attack. We find that large language models for code are vulnerable to data extraction attacks, like their natural language counterparts. From the training data that was identified to be potentially extractable we were able to extract 47% from a CodeGen-Mono-16B code completion model. We also observe that models memorise more, as their parameter count grows, and that their pre-training data are also vulnerable to attack. We also find that data carriers are memorised at a higher rate than regular code or documentation and that different model architectures memorise different samples. Data leakage has severe outcomes, so we urge the research community to further investigate the extent of this phenomenon using a wider range of models and extraction techniques in order to build safeguards to mitigate this issue.

CCS CONCEPTS

• **Security and privacy**; • **Software and its engineering**; • **Computing methodologies → Machine learning**;

KEYWORDS

Large Language Models, Privacy, Memorisation, Data Leakage

1 INTRODUCTION

In recent years, Large Language Models (LLMs) have garnered considerable interest in the realm of Natural Language Processing (NLP) owing to their exceptional accuracy in performing a broad spectrum of NLP tasks [36]. These models, trained on extensive amounts of data, exhibit increased accuracy and emergent abilities as their parameter count grows from millions to billions [52]. LLMs designed for coding are also trained on vast amounts of data and can effectively learn the structure and syntax of programming languages. As a result, they are highly adept at tasks like generating [21], summarising [1], and completing code [30].

Large language models also exhibit emergent capabilities [50]. These abilities cannot be predicted by extrapolating scaling laws and only emerge at a certain critical model size threshold [50]. This makes it appealing to train ever-larger models, as capabilities such as chain-of-thought prompting [51] and instruction tuning [42] only become feasible in models with more than 100B parameters [50].

Many have noted that large language models trained on natural language are capable of memorising extensive amounts of training data [2, 5, 9, 11, 12, 15, 19, 23, 29, 32, 37, 46, 48].

The issue of memorisation in source code is distinct from that of natural language. Source code is governed by different licences that reflect different values than natural language [16, 23]. Hence, in addition to privacy considerations, the memorisation of source code can have legal ramifications. The open-source code used in LLM training for code is frequently licenced under nonpermissive copy-left licences, such as GPL or the CC-BY-SA licence employed by StackOverflow [2].[1] Reusing code covered by these licences without making the source code available under the same licence is considered a violation of copyright law. In some jurisdictions, this leaves users of tools such as CoPilot at legal risk [2, 16, 23]. Licences are unavoidably linked to the source code, as they enforce the developers' commitment to sharing, transparency, and openness [2, 16]. Sharing code without proper licences is also ethically questionable [2, 23, 46].

Memorised data can also include private information [10, 13, 28]. These privacy concerns extend to code, which can contain credentials, API keys, emails, and other sensitive information as well [2, 4]. Memorisation could therefore put the private information contained in the training data at risk.

Recently, attacks which leverage memorisation have successfully extracted (or reconstructed) training data from LLMs [3, 5, 13, 29]. The US National Institute of Standards and Technology (NIST) considers data reconstruction attacks to be the most concerning type of privacy attack against machine learning models [41]. OWASP

---

[1]StackOverflow Licence: https://stackoverflow.com/help/licensing

classifies Sensitive Information Disclosure (LLM06) as the sixth most critical vulnerability in LLM applications. [2]

Larger models are more likely to memorise more data and are more vulnerable to data extraction [5, 13, 29, 41]. The effort to create ever larger LLMs, therefore, creates models which carry more risk.

To our knowledge, previous studies have investigated data memorisation and extraction attacks in natural language, but there has been no empirical investigation of LLMs for code. In this work, we investigate to which extent large language models for code memorise their training data and how this compares to memorisation in large language models trained on natural language. There is no comprehensive framework or approach for measuring memorisation.

We start by defining a data extraction security game that is grounded in the theory behind membership inference attacks and the notion of k-extractability. Using this security game we define a framework to quantify memorisation in LLMs. We use data extraction as an estimator of memorisation. While memorisation of training data can manifest in the form of non-exact duplication, measuring the rate of data extraction data extraction provides a lower bound of memorisation in a model.

We perform experiments leveraging the SATML training data extraction challenge, an existing dataset for natural language. [3] We extend this benchmark by testing memorisation on more models.

We construct a similar dataset for code, by mining data from the Google BigQuery GitHub dataset and by using a CodeGen code generation model [39]. Similarly to the natural language dataset, we first identify samples vulnerable to attack to build a benchmark. We then tested a variety of models on this benchmark. We finally compare the rate of memorisation between text and code models.

Our key result: *Large language models trained on code memorise their training data like their natural language counterparts and are vulnerable to attack.* To summarise, the main contributions of this paper are:

- A novel approach, using a data extraction security game, to quantify memorisation rates of code or natural language models
- A benchmark of key memorisation characteristics for 10 different models of different sizes
- An empirical assessment of memorisation in code models demonstrating that (1) code models memorise training data, albeit at a lower rate than natural language models; (2) larger models, with more parameters, exhibit more memorisation; (3) data carriers (such as dictionaries) are memorised at a higher rate than, e.g., regular code, documentation, or tests; (4) different model architectures memorise different samples.
- We make the code to run the evaluation available to allow others to replicate our results and to evaluate other models. [4]

2 BACKGROUND AND RELATED WORK

2.1 Memorisation

In the context of language models, memorisation refers to the ability of a model to remember and recall specific details of the data it has been trained on. This occurs when a model overfits the training data, meaning it becomes overly specialized and fails to generalise well to new or unseen data [17, 19]. As a result, the model can accurately recall specific phrases, sentences, or even entire documents from the training data. Besides the privacy concerns explained in section 1, memorisation also causes an overestimation of performance. It has, for instance, been observed that CodeX can complete HackerRank problems without receiving the full task description [32].

While memorisation can lead to high accuracy, it is not necessarily an indication of good generalisation performance. A model that has memorised the training data may struggle to perform well on new or unseen data, leading to poor performance in real-world applications. Additionally, memorisation can reduce the ability of the model to adapt its output to specific use cases. For example, when slightly changing HackerRank problems, CodeX [14] struggles to produce a correct solution, instead regurgitating solutions for the original problem [32, 47].

2.2 Membership Inference Attacks

Membership inference attacks are a type of attack that aims to determine whether a specific data point was included in the training data of a machine learning model. The goal of these attacks is to infer whether a given data point was used to train the model or not, without having access to the training data itself.

The first membership inference attack against machine learning models was proposed by Shokri et al. to target classification models deployed by Machine Learning as a Service (MLaaS) providers [45]. Since then the field has expanded and attacks have been proposed that target generative models [24] and LLMs [25]. Recently, membership inference attacks have been proposed against transformer-based image diffusion models such as Stable Diffusion [18].

We refer to the security game defined by Carlini et al. [9] to define a membership inference attack in Definition 1. In this game, the adversary wins if they have a non-negligible advantage $> \frac{1}{2} + \epsilon$. In simpler terms, the adversary needs to be able to distinguish between data that was included and which was not included in the training data for a given model, while only being allowed query access to the model and data distribution.

Membership inference attacks are primitive for measuring the leakage of a machine learning model and are often a starting point for more extensive attacks [9, 26, 38]. While membership inference is a weaker privacy violation than memorisation, the National Institute of Standards and Technology (NIST) still considers membership inference to be a violation of the confidentiality of training data [26].

DEFINITION 1 (MEMBERSHIP INFERENCE SECURITY GAME [9]). *The game proceeds between a challenger C, an adversary $\mathcal{A}$, a data distribution $\mathbb{D}$ and a model f:*

(1) The challenger samples a training dataset $D \leftarrow \mathbb{D}$ and trains a model $f_\theta \leftarrow \mathcal{T}(D)$ on the dataset D.

(2) *The challenger flips a bit b, and if b = 0, samples a fresh challenge point from the distribution $(x, y) \leftarrow \mathbb{D}$ (such that $(x, y) \notin D$). Otherwise, the challenger selects a point from the training set $(x, y) \leftarrow D$.*

(3) *The challenger sends (x, y) to the adversary.*

(4) *The adversary gets query access to the distribution $\mathbb{D}$, and to the model f_θ, and outputs a bit $\hat{b}$*

(5) *Output 1 if $\hat{b} = b$, and 0 otherwise.*

2.3 Data Extraction Attacks

Data extraction attacks are a stronger type of attack where an adversary extracts a data point used to train a model. Attacks can be divided into two types for LLMs, namely guided and unguided attacks [3].

In an unguided attack, the adversary does not know the sample to be extracted from the model. The adversary simply attempts to extract any training point, contained anywhere in the training corpus [10, 12, 13, 40].

In this work, we focus on targeted attacks. In a targeted attack, the adversary is provided with a prefix, which is the first half of the sequence and is then tasked with recovering the suffix, which is the second half of the sequence. Targeted attacks are more security-critical as they allow the targeting of specific information, such as the extraction of emails [3, 10, 23, 27, 38].

We ground our definition of memorisation and extractability in the definition of k-extractability provided by Biderman et al., which was originally inspired by the framework of k-eidetic memorisation introduced by Carlini et al. [13].

DEFINITION 2 (K-EXTRACTABILITY [5]). *A string s is said to be k-extractable if it (1) exists in the training data, and (2) is generated by the language model by prompting with k prior tokens.*

2.4 Natural Language Dataset

The dataset used for the attack on natural language models is provided by the SATML'23 Language Model Data Extraction Challenge[5]. The dataset consists of 15K training, 1K validation, and 1K test samples. The test samples were not released and were only used by the competition organisers. Each sample is divided into a 50-token prefix and a 50-token suffix. For our evaluation, we use the validation set.[5]

The participants had to use a GPT-NEO 1.3B model to extract the suffix using the prefix. The winning entry prompted the model with the prefix, extracted 100 suffixes for each prefix, and trained a binary classifier to select the most correct suffix [3].

The dataset was constructed by analysing Pile [22], which is the corpus used to train the GPT-NEO family of models [7]. The Pile is an 825GB English language dataset, which itself consists of 22 high-quality sub-datasets, ranging from books, academic papers and even code [22]. The Pile was constructed to improve the cross-domain applicability of LLMs. The Pile [22] is also used as a pretraining dataset for a variety of code models [2].[6]

The organisers extracted all the unique 150 token sequences from the 800GB corpus. Sequences were filtered to include only those that are duplicated at least 5 times. They were then split into a pre-prefix, prefix, and suffix, each 50 tokens long. The GPT-NEO model was then prompted with the pre-prefix and prefix (100 tokens). If the model produces the suffix, using greedy decoding, the sequence is considered extractable. The challenge dataset was constructed from the extractable sequences and only includes the prefix and suffix.[5]

3 APPROACH

To measure memorisation in LLMs4Code we first formally define a data extraction game and we construct a dataset of code samples.

3.1 Data Extraction Security Game

We consider the models as black-box systems. We define a security game inspired by the membership inference attack security game in Definition 1 and the notion of k-extractability in Definition 2:

DEFINITION 3 (DATA EXTRACTION SECURITY GAME). *Given a challenger C, an adversary $\mathcal{A}$, a data distribution $\mathbb{D}$ and a model f the game is defined as follows:*

(1) *The challenger samples a training dataset $D \leftarrow \mathbb{D}$ and trains a model $f_\theta \leftarrow \mathcal{T}(D)$ on the dataset D.*

(2) *C samples a sample $D_n = (p, s)$ where $D_n \in D$. The prefix p is provided to the adversary $\mathcal{A}$.*

(3) *$\mathcal{A}$ is allowed query access to the model f_θ and may perform any other polynomial-time operations*

(4) *$\mathcal{A}$ outputs his prediction sequence $\hat{s}$*

(5) *If $\hat{s} = s$, $\mathcal{A}$ wins, otherwise C wins*

In other words, given a prefix (1), the adversary is challenged to extract the correct suffix in the training data from the model. The adversary can query the model (2), but has no access to the weights, unlike the game proposed by Al-Kaswan et al. [3]. The adversary then predicts the suffix (3) and wins if it matches the actual suffix in the training data.

There are some difficulty modifiers to adjust the difficulty of the challenge:

(1) The selection of the dataset $D \subset \mathbb{D}$. As observed by previous works, not all training samples are as hard to extract as others. In particular, samples that are highly duplicated[5] or outliers [12] are more vulnerable to attack.

(2) The choice of model M_θ. Some models are more likely to memorise samples than others, namely larger models have been observed to memorise more samples [5, 8, 10, 11, 13, 29].

(3) The length of the prefix p. It has been found that longer prefixes elicit more memorisation[5] [11, 13, 29]. Note that this length is equivalent to the k in definition Definition 2.

(4) The victory condition $\hat{s} = s$, instead of targeting verbatim memorisation, a fuzzy match could also be considered [29].

In this work, we take inspiration from the competition organised by Carlini et al. and use modifiers (1) and (3) to construct a set of extractable samples. We shorten the prefix of the extractable samples and use this set of hard but extractable samples to perform an evaluation on different models (2). We also measure fuzzy match scores (4) and compare them with the extract match rate.

3.2 Code Dataset Construction

To measure the memorisation in LLMs for code, we first need to construct a dataset similar to the one used in the SATML'23 Language Model Data Extraction Challenge. As there is no code benchmark available, we build one from scratch. This presents several challenges:

Firstly, for some code models, the training data is not published by the authors, which makes it impossible to determine what data were included in the training of these models. We must therefore experimentally determine which data points were presumably included in the training data for each of the models. This has implications for the transferability of the benchmark set, as the training data might differ for each model. Not all models are trained in all programming languages as well, so we must select a common language to test multiple models.

Secondly, since all publicly available code is potentially part of the training data, the search space for extractable data points is massive.

We limit our evaluation to Python since we found that the vast majority of models support Python and have some Python in their training corpus. We source the potentially memorised data from GitHub. We mine Python files using the Google BigQuery Github dataset. [7]

We filter the files to include only nonbinary files longer than 150 tokens. We only consider files that have five or more duplicates on GitHub and randomly select 150 token spans from anywhere in the file. Similarly to the natural language dataset [8], we split the 150 token span into a pre-prefix, a prefix, and a suffix, each 50 tokens long. We prompt a CodeGen-2B-Mono model [39] with the pre-prefix and prefix. We select this model because it is decently sized (there are smaller and larger variants of the model), it is specifically trained on Python and it is the highest performing publically-available model for the Human-Eval benchmark [39].

If the model can predict the suffix, with the 100-token prompt, we consider the sample to be extractable. We randomly select 1K extractable samples to perform our evaluation. We construct the dataset from the prefixes and suffixes.

Our dataset construction procedure differs from the procedure used by Carlini et al. in one aspect. Our dataset does not guarantee that for every $D_n = (s, p)$ there does not exist a $(s, p') \in D$ where $p \neq p'$. There are two main reasons for omitting this step:

- For many models in our evaluation we do not have access to the training data and possible pre-training data. The organisers could guarantee that the model under investigation was only exposed to the Pile. We want our approach to work for settings in which the investigator has no access to the training data.
- The computational cost of identifying all unique samples $D_n = (s, p)$ is extremely large for a dataset of this size and our aim is to create an approach that does not require such enormous compute capabilities.

Table 1: Natural language (top 4 rows) and code models under investigation

Model	Developers	Parameters
GPT-NEO	EleutherAI	125M, 1.3B, 2.7B
GPT-2	OpenAI	117M, 345M, 774M, 1.5B
Pythia	EleutherAI	70M, 160M, 410M, 1B 1.4B, 2.8B, 6.9B
CodeGen-NL	Salesforce	350M, 1B, 3B, 7B, 16B
CodeGen-Mono	Salesforce	350M, 1B, 3B, 7B, 16B
CodeGen-Multi	Salesforce	350M, 1B, 3B, 7B, 16B
CodeGen2	Salesforce	1B, 3.7B, 16B
CodeParrot	Huggingface	110M, 1.5B
InCoder	Facebook	1.5B
PyCodeGPT	Microsoft	110M
GPT-Code-Clippy	CodedotAI	125M

4 METHODOLOGY

4.1 Research Questions

RQ1: *How does the rate of memorisation compare between Natural Language and Code trained LLMs?* To compare the rate of memorisation, we run both the attack on natural language as well as code models and compare the results. Intuitively we expect code models to be able to memorise more since code is more structured and there is much more natural language data available.

RQ2: *What type of data are memorised by code-trained LLMs?* We want to know if there is a code pattern that is memorised. To do this we take the set of samples vulnerable to attack and we manually analyse them by constructing a classification of the samples.

RQ3: *How much overlap is there between the memorised samples in different code-trained LLMs?* Do some models memorise different samples than others? Could we perhaps leverage a selection of different models to extract more data and do some models memorise more of a certain type of sample than others?

RQ4: *To what extent do LLMs trained in code leak their pre-training data?* Finally, we want to see if pre-trained models can also leak their pre-training data. To investigate this, we select a code model that has been pre-trained on the Pile and perform the natural language attack. We compare the performance of the original base model with that of the code-trained model to see how much training data is retained. When referring to a base model in this paper, we only mean models that were initialised with the **architecture** and **weights** of a different model.

4.2 Models

The models, their developers, and their respective sizes are shown in Table 1. We limit our evaluation to left-to-right autoregressive models, which are available on the HuggingFace Hub.

---

[7]GitHub on BigQuery: https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyse-all-the-open-source-code

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

Table 2: Categories of memorised samples

Category	Purpose	Count
Code	Code Logic	679
Testing	Test Code	87
License	Licence information	13
Docs	Documentation	86
Dicts	Dictionaries or other data carriers	135

For natural language evaluations, we used GPT-NEO [7], the models used to build the natural language dataset [5]. We select GPT-2 [43] to test the transferability of the prompts to a model trained on a different corpus. GPT-2 is trained on the WebText corpus, which was mined by finding all the outlinks on Reddit with more than 3 karma. We also investigate the Pythia [6] suite of models, which are trained on the Pile [22].

The CodeGen suite of models [39] features a number of different models in a variety of sizes. The models were initialised and first pre-trained on the Pile; these models are the CodeGen-NL models. The CodeGen-NL models are then further trained on a dataset containing multiple programming languages to create the CodeGen-Multi models. The Multi models were finally trained on a dataset consisting of only Python code to create the CodeGen-Mono models. The CodeGen2 and Incoder models are both designed for infilling but have autoregressive capabilities as well [21, 39]. CodeParrot is a pre-trained GPT-2 model fine-tuned on the APPS dataset [44]. PyCodeGPT is a small and efficient code generation model based on the GPT-NEO architecture [53]. GPT-Code-Clippy is a pre-trained GPT-NEO model fine-tuned on code.

4.3 Categorisation
We build a classification of the 1K extractable 150-token samples by doing an explorative study. We find the following categories and classify each of the samples into one category. For simplicity, we classify each sample which has two purposes, into its majority category. The different categories are shown in Table 2. We identified 5 different categories as shown in Table 2.

4.4 Extraction
We prompt the model under investigation with the prefix. We use the standard generation pipeline and the default generation configuration of the model as defined in the model configuration. For models which use a different tokeniser than the CodeGen tokeniser used for the dataset construction. We simply tokenise the sample again using the new tokeniser. Any samples that are too short under the new tokeniser are discarded.

4.5 Evaluation Metrics
The models are prompted in a one-shot fashion with greedy decoding. We measure the exact match rate (EM). Additionally, we also measure the fuzzy match, using the BLEU-4 score. For the model size, we measure the total parameter count.

Table 3: Code attack performance on Large Language Models for Code

Model	Parameters (M)	Memorisation rate	
		EM	BLEU-4
CodeGen-350M-Mono	357	0.101	0.567
CodeGen-2B-Mono	2779	0.303	0.712
CodeGen-6B-Mono	7074	0.382	0.756
CodeGen-16B-Mono	16032	0.471	0.801
CodeGen-350M-Multi	357	0.100	0.536
CodeGen-2B-Multi	2779	0.204	0.628
CodeGen-6B-Multi	7074	0.258	0.659
CodeGen-16B-Multi	16032	0.297	0.695
CodeGen-2B-nl	2779	0.077	0.465
CodeGen2-1B	1015	0.082	0.482
CodeGen2-3.7B	3641	0.106	0.517
CodeGen2-7B	6863	0.116	0.530
CodeParrot-small	111	0.088	0.529
CodeParrot	1510	0.314	0.721
InCoder	1312	0.115	0.559
PyCodeGPT	111	0.079	0.567
GPT-NEO	2651	0.058	0.454

4.6 Configuration
We process and visualise the data with Modin 0.20.0 and Pandas 2.0.1. We run inference using Transformers version 4.16.2 running on Torch 1.9.0+cu111. The experiments were conducted on a cluster running RedHat 7, we allocated 8 CPU cores with 32GB of RAM and an Nvidia A40 GPU with 48GB of video memory. The GPU is running Nvidia driver version 530.30.02 with Cuda 12.1.

For replication purposes, we only consider models that are runnable on our hardware. We found that the limitation was the GPU memory, so there are some models that we did consider but did not fit the GPU memory (such as InCoder-6.7B and StarCoder-base).

5 RESULTS
We present the results of our experiments to answer the research questions, results are grouped per research question.

5.1 Natural Language vs Code
The results of the attack are shown in Table 4. We found that we are able to extract 56% of the samples with the largest GPT-NEO model. The medium-sized model, which was used to construct the dataset, achieved an exact match rate of 46%. The models which were not trained on the Pile [22] did not memorise much if any of the samples.

As shown in Figure 1, for the models that are trained on the Pile [22], memorisation scales with the size of the model. We do not observe a clear difference between the Pythia and Pythia-dedup models, indicating that their deduplication was unsuccessful in preventing the memorisation which we measure. As the number
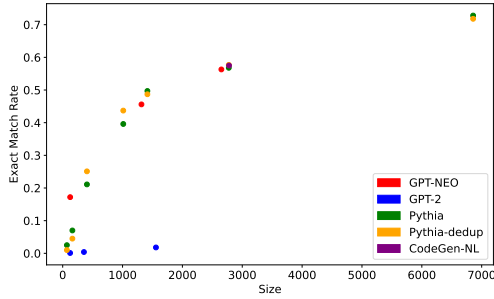
Ali Al-Kaswan, Maliheh Izadi, and Arie van Deursen

Figure 1: Parameter size and exact match rate for natural language models
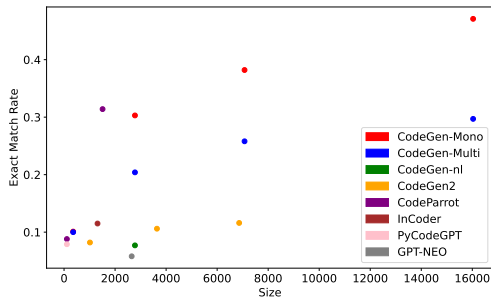
Figure 2: Parameter size and exact match rate for code models

Table 4: Natural language attack performance on natural language models

Model	Parameters (M)	Memorisation rate	
		EM	BLEU-4
GPT-NEO-125M	125	0.172	0.529
GPT-NEO-1.3B	1316	0.456	0.767
GPT-NEO-2.7B	2651	0.563	0.829
GPT-2	124	0.001	0.328
GPT-2-Medium	355	0.004	0.375
GPT-2-Large	1558	0.018	0.396
Pythia-70M	70	0.025	0.261
Pythia-160M	162	0.070	0.355
Pythia-410M	405	0.211	0.509
Pythia-1B	1012	0.396	0.658
Pythia-1.4B	1415	0.497	0.742
Pythia-2.8B	2775	0.568	0.793
Pythia-6.9B	6857	0.728	0.880
Pythia-dedup-70M	70	0.010	0.273
Pythia-dedup-160M	162	0.045	0.372
Pythia-dedup-410M	405	0.251	0.550
Pythia-dedup-1B	1012	0.437	0.679
Pythia-dedup-1.4B	1415	0.487	0.712
Pythia-dedup-2.8B	2775	0.577	0.805
Pythia-dedup-6.9B	6857	0.718	0.877
CodeGen-2B-NL	2779	0.575	0.860

of parameters increases for each model architecture, it becomes evident that the rate of memorization grows logarithmically.

Table 3 and Figure 2 show the results of the experiments. We found that we were able to extract 38% of the samples from the largest CodeGen-Mono model we tested. The 1B parameter model, which was used to generate the test set, was only able to extract 30% of the samples, which is lower than the performance of GPT-NEO 1.3B on the natural language dataset. This indicates that our constructed code dataset is harder than the natural language dataset, but that difficulty modifier (2) from section 3 which was supported by previous works and Definition 1 also holds for our code dataset.

Figure 3 shows the relation between the Exact Match rate and the BLEU-4 score for code-trained models. We can observe that there is a clear relation between the exact match rate and the BLEU4 score, especially above an exact match rate of 0.2. We see a similar pattern in Figure 3. The Pearson correlation coefficient between the Exact Match rate and the BLEU4 score is 0.982 and 0.967 for natural languageand code, respectively, indicating a very strong positive correlation.

In our evaluation, we also tested multiple models that were not primarily trained on programming languages. We found that CodeGen-nl and GPT-NEO were unable to memorise as much as similarly sized code-trained models, but were still able to achieve an exact match score of around 10%.
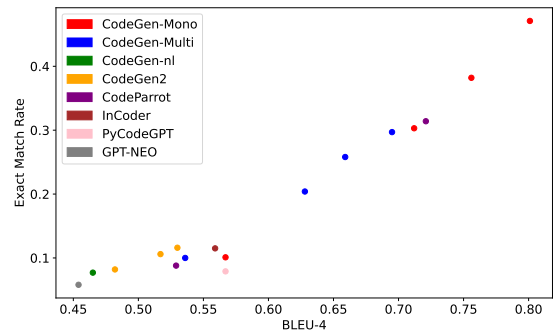

Figure 3: BLEU-4 score and Exact match rate for code models

Similarly to natural language models, we also find that memorisation scales with model size in Figure 2. But in this case, we see the logarithmic relationship between the same model architectures. We also observe that the CodeGen-Mono models memorise more natural language than the CodeGen-Multi models for every model size. This indicates that the extra training on Python code increases the memorisation rate. We find a Pearson's correlation coefficient between the Exact Match rate and the size of the model of 0.797 and
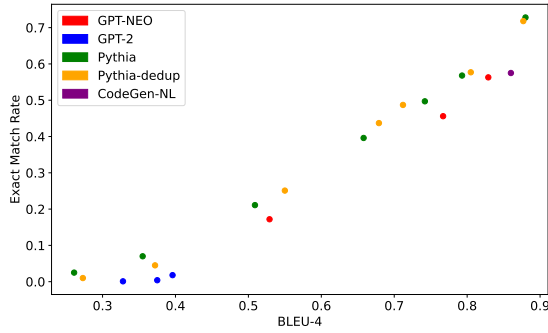
Figure 4: BLEU-4 score and Exact match rate for natural language models

0.704 for the natural language and the code, respectively, indicating a strong positive correlation.

> **RQ1:** Code-trained LLMs memorise their training data at a lower rate than Natural Language trained LLMs. In both natural language and code-trained models, the rate of memorisation scales with the model size. Continued exposure to the same data increases the rate of memorisation.

5.2 Type of Memorised Samples

As can be observed in Figure 5, the majority of samples in our dataset are code logic followed by dictionaries. We colour-coded the samples to make a distinction between memorised and non-memorised samples. We find that data carriers and licence information are being memorised at a higher rate than code logic, documentation, and test code.

During the tagging process, we did find multiple examples of names, emails, and usernames being memorised by the model. Such as the example in Figure 6 We also found an example of some API keys, further investigation shows that this instance was a sample that was easily findable using search engines.

> **RQ2:** LLMs trained on code memorise data carriers and license information at a higher rate than regular source code, documentation, and testing code. Code-trained LLMs are also able to memorise and emit sensitive information.

5.3 Which Model Memorises What

In Figure 7 we plot the overlap in memorised samples between different models. We limit the investigation to the Codegen, CodeGen2 and CodeParrot family of models.

For instance, we find that 86% of all samples which were memorised by CodeParrot-small are also memorised by CodeParrot, while only 24% of the samples memorised by CodeParrot-small are memorised by CodeParrot. We find similar patterns when comparing the different-sized CodeGen models. The CodeGen-2 family of models memorised fewer samples and is in line with the CodeGen-350M models despite the size difference. The larger models in a family
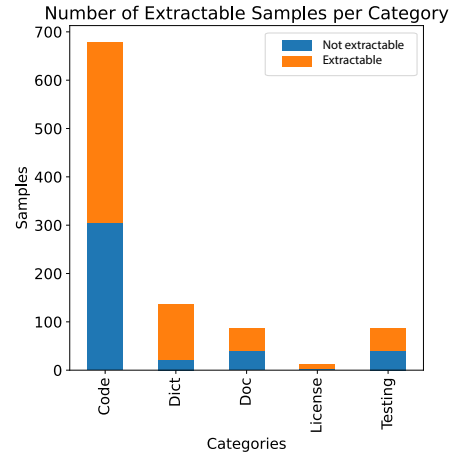

Figure 5: Categories of memorised samples

Figure 6: Instance of memorised API keys. Actual keys are replaced with placeholder values.

```
{
    'oauth_token: '#############',
    'oauth_token_secret: '###############',
    'oauth_verifier: '###########',
}
>>> oauth_session
```

memorise more samples, there are a few distinct samples that are only memorised by the small models, but we find that is generally limited.

We find that the CodeGen-Multi models tend to memorise around 50% of the samples memorised by their respectively sized Mono variant, while the Mono models memorise around 70% of the samples memorised by the Multi variant. The only exception is the smallest model, where the Multi and Mono models memorised very similar amounts of samples. In Figure 8 we find that 40% of the samples are not memorised by any model at all. But there are 73 samples that are memorised by 12 of all the 13 models. This indicates that there is an inherent difficulty in some samples.

Figure 9 shows the memorisation of each of the categories per model. We find that all plotted models memorise more code and data carriers than any of the other categories, which is supported by Figure 5. As models grow larger they memorise relatively more code and fewer data carriers. In absolute terms, the number of memorised samples from the Dict category still increases.

Combined with the findings in RQ1 we can therefore conclude that the extra training on Python, makes the models memorise more and many of the same samples and that the smaller models lack the capacity to memorise more data.

> **RQ3:** Each model family memorises a unique set of samples, and smaller models within the same family remember only a subset of what their larger counterparts do.
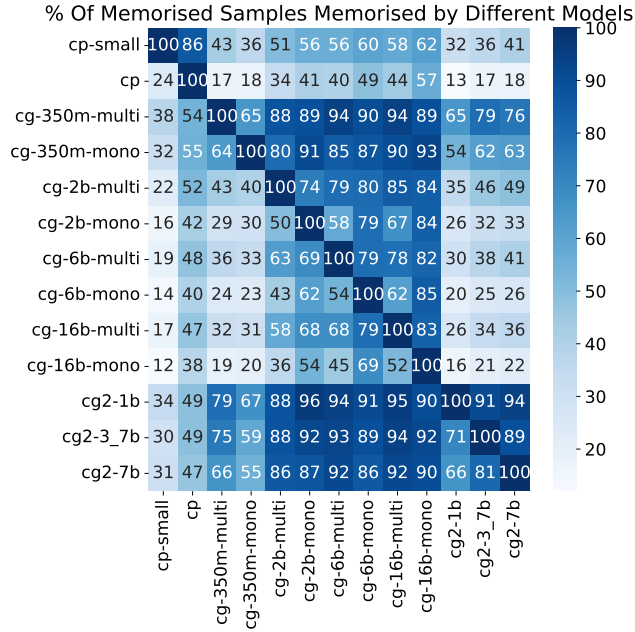
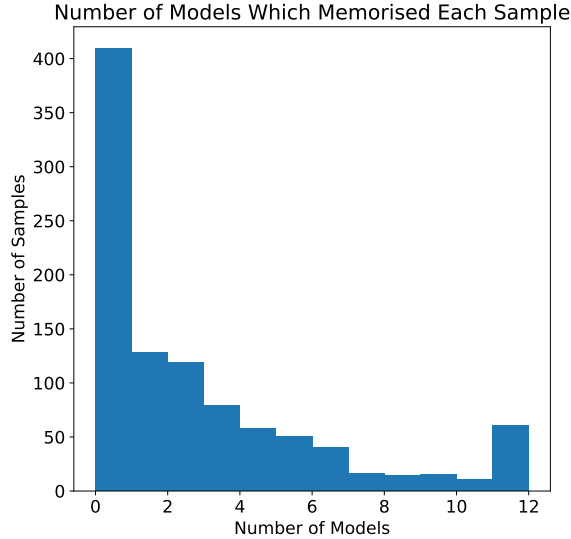Figure 7: Memorisation overlap between CodeParrot (cp) and CodeGen (cg) Models

Figure 8: Memorisation counts

5.4 Pre-Training Data Leakage

In Table 5 and Figure 10 we plot the results for the leakage of pre-training data. We find that we can extract 58% of all natural language samples from the CodeGen-NL model. This result aligns with the similarly sized Pythia and GPT-NEO models in Table 4. Tuning the model on code data reduces the extraction rate to 31% and tuning on Python code further reduces the extraction rate to 20%.
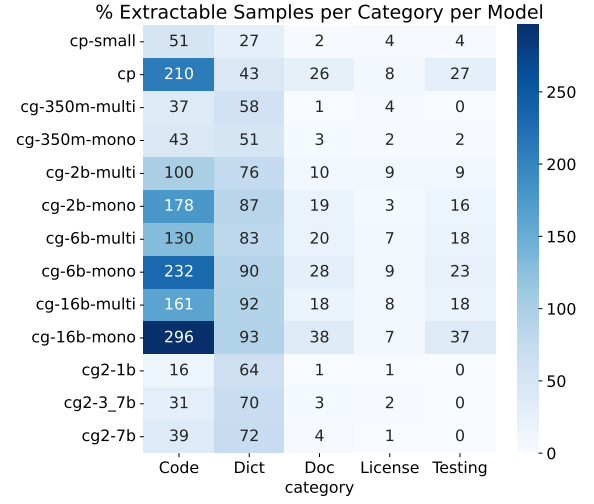

Figure 9: Percentage of extractable samples belonging to each category for CodeParrot (cp) and CodeGen (cg) Models

Table 5: Text extraction rate on code models

		Memorisation rate	
Model	Parameters (M)	EM	BLEU-4
CodeGen-350M-NL	357	0.295	0.676
CodeGen-2B-NL	2779	0.575	0.860
CodeGen-6B-NL	7064	0.708	0.915
CodeGen-16B-NL	16032	0.779	0.934
CodeGen-350M-Multi	357	0.248	0.539
CodeGen-2B-Multi	2779	0.310	0.588
CodeGen-6B-Multi	7064	0.414	0.595
CodeGen-16B-Multi	16032	0.351	0.618
CodeGen-350M-Mono	357	0.149	0.454
CodeGen-2B-Mono	2779	0.202	0.502
CodeGen-6B-Mono	7064	0.175	0.518
CodeGen-16B-Mono	16032	0.223	0.546
GPT-NEO	125	0.172	0.529
GPT-Code-Clippy	125	0.000	0.148

We are unable to extract any text samples from GPT-Code-Clippy. The GPT-NEO-125M base model already shows very little extractability in Table 3.

> **RQ4:** While fine-tuning does incrementally reduce the extractability of pre-training data, the pre-training data is still vulnerable to attack, especially as the models grow larger.

6 DISCUSSION

The results in section 5 show that large language models pre-trained on source code also memorise their training data and that they are susceptible to targeted training data extraction attacks.
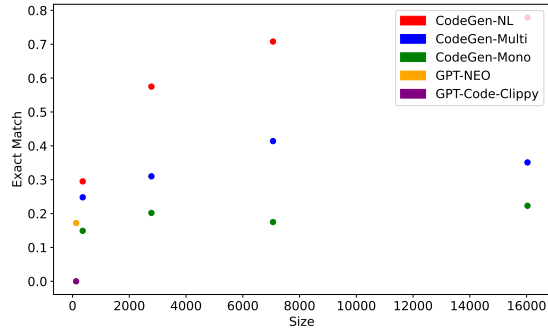
Figure 10: Parameter size and exact match rate for pre-trained models

6.1 Interpretation

Multi vs Mono. The findings indicate that the CodeGen-Mono models memorised more than the Multi models. This is explainable by the fact that the Mono models have had more exposure to Python code and therefore code in our dataset. Recall that the models are first trained on the Pile which contains all the GitHub repos with more than 100 stars [22]. The models are further trained on a general dataset of code, and finally on a dataset of Python code. This means that the models could have possibly been trained on the same file three times.

Size and Memorisation. We find that the rate of memorisation scales with the size of the model, across all models we find that the rate of memorisation increases as the size increases. This is in line with the findings of previous work which found that larger LLMs memorise training data faster [48] and at a higher rate than small models [5, 8, 10, 11, 13]. Our results also confirm that the log-linear relation between size and memorisation, which has been observed by other works [11, 29] holds for LLMs trained on code as well.

Our experiments which investigate the overlap of memorised sequences in different sizes of code models show that the memorised samples of smaller models are mostly a subset of the large models. This indicates that as a model grows larger it mostly memorises more and not necessarily different data.

Biderman et al. investigated memorisation in the Pythia suite of models [6] and found that 94% of the sequences memorised by the 70M model were also memorised by the 12B model, but those only accounted for 19% of the sequences that the 12B model memorised. We find a similar relation between the largest and smallest CodeGen-Mono models: CodeGen-Mono-16B memorised 93% of the samples which were memorised by CodeGen-Mono-350M, conversely only 20% of the samples memorised by CodeGen-Mono-16B were memorised by CodeGen-Mono-350M.

Rate of Memorisation. Note that the results obtained from experiments in section 5 suggest that memorisation in LLMs trained on code is less than in those trained in natural language. The largest 6.9B parameter Pythia model memorised 55% more samples than the best-performing CodeGen-Mono model. Intuitively we would

expect the memorisation to be more in code models (as explained in section 4), but there might be multiple reasons for this observation:

- Our dataset construction procedure differs from the procedure used by Carlini et. al. The natural language dataset guarantees that for every $D_n = (s, p)$ there does not exist a $(s, p') \in D$ where $p \neq p'$. This means that for some prefixes the model might predict a suffix that is also in the training data, which would be counted as a non-memorised sample. This was not possible in our case, since we do not exactly know the training data for the code models under investigation. The training dataset was only deduplicated on the file level.
- The structured nature of code might illicit less memorisation in general. This is supported by the higher rate of memorisation in dictionaries compared to regular code especially in smaller models. Their relative information density makes it hard to generalise for these samples specifically and the models might therefore revert to memorisation.

Deduplication. The deduplicated Pythia [6] models are not significantly more robust against our extraction than their regular counterparts. At first glance, this is a surprising finding. It has been reported that deduplicating the training data makes LLMs more secure against data extraction [13, 31, 33].

A similar investigation by Biderman et al. on memorisation on the Pythia suite of models also found a relatively small difference between the two variants [5]. The authors theorise that this observation might be due to the training setup. The deduplicated models were trained for 1.5 epochs to offset the smaller data size and to train on the same number of epochs. This effectively oversamples the entire dataset.

Based on our observations we can offer two alternative explanations:

(1) The training was deduplicated on the file level [6]. Our evaluation concerns spans of tokens that can be duplicated across files. The same licence information, for instance, is present in the preamble of many different files and will still be present in the deduplicated dataset.
(2) The samples memorised by the Pythia models might be outliers that illicit memorisation. We observed that information carriers are more likely to be memorised than other types of samples, so the deduplication might not have had much impact on these samples.

6.2 Implications

We propose a novel framework to measure the memorisation and extractability of training data in LLMs.

Model training. This work serves to inform researchers and practitioners who aim to train their own LLMs. We can confidently say that larger LLMs leak more and that smaller LLMs are therefore preferable from a safety perspective. In light of emergence [50], larger models are however often preferable. We are already able to extract 73% and 47% of the text and code samples, even larger models like CodeX [14] or Starcoder [34] might memorise even more data.

Secondly, we have shown that LLMs also leak their pre-training data even after multiple training rounds. The ability to recover pre-training samples has additional privacy and security implications for the transfer learning paradigm [2]. When creating and publishing a model, the base model is also something to be considered as the pre-training data can be unintentionally exposed as well.

Finally, some types of data are more vulnerable to extraction than others. This information can be used to inform the data selection procedure. Some categories like dictionaries can be omitted entirely to reduce the amount of memorisation. Future work can investigate how training data can be curated and sanitised to reduce memorisation in LLMs.

Model deployment. The black-box setting of our evaluation has implications for MLaaS services as well. Since we do not require additional information about the model, our data extraction approach could be used against models that are offered through public APIs such as OpenAI's Copilot [14]. While Copilot does employ a memorisation filter, it is relatively easy to bypass [28]. There is a need to develop stronger countermeasures to prevent data extraction from these models.

Framework. The framework and dataset provided can be used the evaluate different models. While our focus has been on left-to-right causal language models, different architectures, such as encoder-only models like CodeBERT [20] or encoder-decoder models like CodeT5 [49] might memorise different amounts and different types of training data.

Fair Use. Many existing LLMs for code make use of code licenced under copyleft and other non-permissive licences [2]. The use of public code to train LLMs for code is an instance of fair use, which is a defence that allows the use of copyrighted works in new and unexpected ways and exists in many jurisdictions [23]. If the output of the model is similar to the copyrighted input fair use might no longer be applicable. The output needs to conform to the licence terms of the copied input [23], which can include share-alike and attribution clauses [2].

Memorisation can therefore put the creators and users of LLMs for code at legal risk [23]. This risk extends to pre-trained models, as some pre-training corpora, including the Pile [22], also contain code licenced under non-permissive licences [2]. The risk can be avoided by training models with code licenced under permissive licences (such as BSD-3 or MIT) or providing provenance information to trace the code back to its source so that the user of the output can abide by the original licence [23, 34].

Extraction techniques. We were able to show that using relatively simple greedy decoding and the notion of k-extractability, most text models and all code models are leaking data. This only proves the inherent leakiness of these models and serves as a stepping stone for more advanced and powerful attacks. One approach worth investigating is the use of prompt engineering to extract data. With hard or soft-prompts [35] the model could be enticed to output more memorised data. Our work only prompts the models with the prefix, while different prompts might elicit more memorisation. Another approach is to explore the use of Membership Inference Attacks to increase the abilities of the attacks further. One could take inspiration from untargeted attacks and generate multiple

suffixes per prefix using a different decoding method. The MIA can then serve to select the correct suffix [1].

6.3 Limitations and Threats to Validity

6.3.1 Internal validity. In our evaluation, we did not take into account the location of the samples. The samples are of a fixed token length but can originate from any arbitrary location in the file. Furthermore, Byte-Pair Tokenisation can cause the sample to start or end in the middle of a word. We based our dataset construction on existing work [3, 5], but samples from the beginning or end of the file could be easier to extract. Initially, untargeted extractions were attempted, and it was discovered that samples were predominantly obtained from the beginning of the file. Nevertheless, the current approach was chosen as it would enhance the versatility of our attack and enable us to extract samples from any location within the file.

6.3.2 External validity. Our evaluation focuses on a limited number of models, other models might exhibit more or less memorisation. Our benchmark was constructed using a single model, and while we were able to show that our benchmark gave promising results for other models, other data sources and models should be used to construct more benchmarks.

The constructed datasets only consider duplicated sequences; this inherently limits the applicability of our attack on low-duplication data. While other works do state that models can also memorise unduplicated data, we cannot experimentally confirm this as we only apply coarse file-level deduplication.

In the construction of our dataset, we only considered Python code. We selected Python because it is supported by almost all code generation models. Other less-expressive languages could show different patterns and different degrees of extractability. Python is a very popular language, so these results might also not apply to less popular languages. We plan to extend our evaluation to include more programming languages in the future.

6.3.3 Construct validity. We mainly use the exact match metric to measure memorisation in code models. This metric likely underestimates the actual number of memorised samples, as some might be slightly changed by the model. For this specific study, we are more interested in exact reproductions by the model, since we are more interested in the privacy and security aspect of memorisation. When examining the licensing aspects of memorization, fuzzy match metrics may provide better insights. We included BLEU4 to account for this, but we found that it is highly correlated with the exact match rate. However, there are no automated metrics available to measure non-literal infringement based on current legal standards [23].

6.3.4 Ethical Considerations. While this work does describe techniques that can potentially be used to extract sensitive information from models, we do so ethically. Our goal is to bring attention to the issue of memorisation in LLMs for code and inform the users and creators of these models and provide them with tools to measure this. In this work, we, therefore, do not needlessly expose any private information, and we urge users of our framework to refrain from doing so as well. We target randomly selected sequences from popular and public repositories to avoid accidentally exposing

private information. However, we still found some instances of usernames, emails, and API keys in our data, but we found that these are easily findable using search engines and are part of popular and well-indexed public repositories. We believe that the benefits outweigh the risks, and we decide to share our datasets.

7 CONCLUSION

To conclude, we presented an extensive study on memorisation in LLMs for code. We formally define a data extraction security game grounded in the existing notion of k-extractability and membership inference attacks. We utilised this game to create a dataset to measure memorisation in LLMs for code. We compared the rate of memorisation between models of code and natural language, we compared the rate and type of memorisation between different models, and we investigated the rate of memorisation of pre-training data in LLMs for code.

We found that LLMs for code memorise their training data like their natural language counterparts, albeit at a lower rate. We further found that the rate of memorisation increases as a model grows and that different model architectures memorise distinct sets of samples, while smaller versions of the same family tend to memorise a smaller subset of their larger sibling. We found that data carriers and licence information are being memorised at a higher rate than code, documentation, and tests. Finally, we found that the pre-training data is still vulnerable to extraction even after multiple tuning rounds.

Our work is a first step and provides a framework to measure memorisation in LLMs for code. We strongly advise the research community to conduct a more comprehensive investigation into the extent of data leakage and employ a diverse range of models and extraction techniques to develop safeguards that can effectively mitigate this issue. The consequences of data leakage can be severe, so it is crucial to take proactive measures to address this problem.

REFERENCES

[1] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Prem Devanbu, and Arie van Deursen. 2023. Extending Source Code Pre-Trained Language Models to Summarise Decompiled Binaries. In *Proceedings of the 30th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

[2] Ali Al-Kaswan and Maliheh Izadi. 2023. The (ab)use of Open Source Code to Train Large Language Models. In *Proceedings of the 2nd International Workshop on Natural Language-based Software Engineering (NLBSE)*.

[3] Ali Al-Kaswan, Maliheh Izadi, and Arie van Deursen. 2023. Targeted Attack on GPT-Neo for the SATML Language Model Data Extraction Challenge. *ArXiv* abs/2302.07735 (2023).

[4] Setu Kumar Basak, Lorenzo Neil, Bradley Reaves, and Laurie Williams. 2023. SecretBench: A Dataset of Software Secrets. *arXiv preprint arXiv:2303.06729* (2023).

[5] Stella Biderman, USVSN Sai Prashanth, Lintang Sutawika, Hailey Schoelkopf, Quentin Anthony, Shivanshu Purohit, and Edward Raf. 2023. Emergent and predictable memorization in large language models. *arXiv preprint arXiv:2304.11158* (2023).

[6] 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. arXiv:2304.01373 [cs.CL]

[7] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow. (March 2021). https://doi.org/10.5281/zenodo.5297715

[8] Hannah Brown, Katherine Lee, Fatemehsadat Mireshghallah, Reza Shokri, and Florian Tramèr. 2022. What does it mean for a language model to preserve privacy?. In *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency*. 2280–2292.

[9] Nicholas Carlini, Steve Chien, Milad Nasr, Shuang Song, Andreas Terzis, and Florian Tramer. 2022. Membership inference attacks from first principles. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1897–1914.

[10] Nicholas Carlini, Jamie Hayes, Milad Nasr, Matthew Jagielski, Vikash Sehwag, Florian Tramer, Borja Balle, Daphne Ippolito, and Eric Wallace. 2023. Extracting training data from diffusion models. *arXiv preprint arXiv:2301.13188* (2023).

[11] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. 2022. Quantifying memorization across neural language models. *preprint arXiv:2202.07646* (2022).

[12] Nicholas Carlini, Matthew Jagielski, Chiyuan Zhang, Nicolas Papernot, Andreas Terzis, and Florian Tramer. 2022. The privacy onion effect: Memorization is relative. *Advances in Neural Information Processing Systems* 35 (2022), 13263–13276.

[13] 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.

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde 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, Josh 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 Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[15] Zitao Chen and Karthik Pattabiraman. 2024. Overconfidence is a Dangerous Thing: Mitigating Membership Inference Attacks by Enforcing Less Confident Prediction. In *Network and Distributed System Security (NDSS) Symposium*.

[16] Madiha Zahrah Choksi and David Goedicke. 2023. Whose Text Is It Anyway? Exploring BigCode, Intellectual Property, and Ethics. *ArXiv* abs/2304.02839 (2023).

[17] Christopher A. Choquette-Choo, Florian Tramer, Nicholas Carlini, and Nicolas Papernot. 2021. Label-Only Membership Inference Attacks. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 1964–1974. https://proceedings.mlr.press/v139/choquette-choo21a.html

[18] Jan Dubiński, Antoni Kowalczuk, Stanisław Pawlak, Przemysław Rokita, Tomasz Trzciński, and Paweł Morawiecki. 2023. Towards More Realistic Membership Inference Attacks on Large Diffusion Models. *arXiv preprint arXiv:2306.12983* (2023).

[19] 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*. 954–959.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[21] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *preprint arXiv:2204.05999* (2022).

[22] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv preprint arXiv:2101.00027* (2020).

[23] Peter Henderson, Xuechen Li, Dan Jurafsky, Tatsunori Hashimoto, Mark A Lemley, and Percy Liang. 2023. Foundation Models and Fair Use. *arXiv preprint arXiv:2303.15715* (2023).

[24] Benjamin Hilprecht, Martin Härterich, and Daniel Bernau. 2019. Monte Carlo and Reconstruction Membership Inference Attacks against Generative Models. *Proc. Priv. Enhancing Technol.* 2019, 4 (2019), 232–249.

[25] Sorami Hisamoto, Matt Post, and Kevin Duh. 2020. Membership inference attacks on sequence-to-sequence models: Is my data in your machine translation system? *Transactions of the Association for Computational Linguistics* 8 (2020), 49–63.

[26] Hongsheng Hu, Zoran Salcic, Lichao Sun, Gillian Dobbie, Philip S Yu, and Xuyun Zhang. 2022. Membership inference attacks on machine learning: A survey. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–37.

[27] Jie Huang, Hanyin Shao, and Kevin Chen-Chuan Chang. 2022. Are Large Pre-Trained Language Models Leaking Your Personal Information?. In *Findings of the Association for Computational Linguistics: EMNLP 2022*. Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 2038–2047. https://aclanthology.org/2022.findings-emnlp.148

[28] Daphne Ippolito, Florian Tramèr, Milad Nasr, Chiyuan Zhang, Matthew Jagielski, Katherine Lee, Christopher A Choquette-Choo, and Nicholas Carlini. 2022. Preventing Verbatim Memorization in Language Models Gives a False Sense of Privacy. *arXiv preprint arXiv:2210.17546* (2022).

[29] Shotaro Ishihara. 2023. Training Data Extraction From Pre-trained Language Models: A Survey. *arXiv:2305.16157* [cs.CL]

[30] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: Multi-Token Code Completion by Jointly Learning from Structure and Naming Sequences. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. ACM, 401–412. https://doi.org/10.1145/3510003.3510172

[31] Nikhil Kandpal, Eric Wallace, and Colin Raffel. 2022. Deduplicating training data mitigates privacy risks in language models. In *International Conference on Machine Learning*. PMLR, 10697–10707.

[32] Anjan Karmakar, Julian Aron Prenner, Marco D'Ambros, and Romain Robbes. 2022. Codex Hacks HackerRank: Memorization Issues and a Framework for Code Synthesis Evaluation. *arXiv preprint arXiv:2212.02684* (2022).

[33] 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 (Volume 1: Long Papers)*. 8424–8445.

[34] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL]

[35] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[36] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

[37] 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 Computer Society, 346–363.

[38] Fatemehsadat Mireshghallah, Kartik Goyal, Archit Uniyal, Taylor Berg-Kirkpatrick, and Reza Shokri. 2022. Quantifying Privacy Risks of Masked Language Models Using Membership Inference Attacks. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 8332–8347. https://aclanthology.org/2022.emnlp-main.570

[39] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv preprint* (2022).

[40] Myung Gyo Oh, Leo Hyun Park, Jaeuk Kim, Jaewoo Park, and Taekyoung Kwon. 2023. Membership Inference Attacks With Token-Level Deduplication on Korean Language Models. *IEEE Access* 11 (2023), 10207–10217.

[41] Alina Oprea and Apostol Vassilev. 2023. *Adversarial machine learning: A taxonomy and terminology of attacks and mitigations*. Technical Report. National Institute of Standards and Technology.

[42] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[43] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. [n. d.]. Language Models are Unsupervised Multitask Learners. ([n. d.]).

[44] Swapnil Sharma, Nikita Anand, and Kranthi Kiran G. V. 2023. Stochastic Code Generation. arXiv:2304.08243 [cs.CL]

[45] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 3–18.

[46] Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. 2022. CoProtector: Protect Open-Source Code against Unauthorized Training Usage with Data Poisoning. In *Proceedings of the ACM Web Conference 2022*. 652–660.

[47] Jasper Tan, Daniel LeJeune, Blake Mason, Hamid Javadi, and Richard G. Baraniuk. 2023. A Blessing of Dimensionality in Membership Inference through Regularization. In *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 206)*, Francisco Ruiz, Jennifer Dy, and Jan-Willem van de Meent (Eds.). PMLR, 10968–10993. https://proceedings.mlr.press/v206/tan23b.html

[48] Kushal Tirumala, Aram Markosyan, Luke Zettlemoyer, and Armen Aghajanyan. 2022. Memorization without overfitting: Analyzing the training dynamics of large language models. *Advances in Neural Information Processing Systems* 35 (2022), 38274–38290.

[49] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[50] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).

[51] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[52] 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*. 1–10.

[53] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. CERT: Continual Pre-training on Sketches for Library-oriented Code Generation. In *The 2022 International Joint Conference on Artificial Intelligence*.