Unveiling Memorization in Code Models

Zhou Yang[♦], Zhipeng Zhao[♥], Chenyu Wang[♦], Jieke Shi[♦],
Dongsun Kim[♠], Donggyun Han[♣], and David Lo[♦]

[♦]School of Computing and Information Systems, Singapore Management University, Singapore
[♥]Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
[♠]School of Computer Science and Engineering, Kyungpook National University, Daegu, South Korea
[♣]Department of Computer Science, Royal Holloway, University of London, London, UK
{zyang, chenyuwang, jiekeshi, davidlo}@smu.edu.sg, zpzhao.zzp@gmail.com, darkrsw@knu.ac.kr,
DongGyun.Han@rhul.ac.uk

ABSTRACT

The availability of large-scale datasets, advanced architectures, and powerful computational resources have led to effective code models that automate diverse software engineering activities. The datasets usually consist of billions of lines of code from both open-source and private repositories. A code model memorizes and produces source code verbatim, which potentially contains vulnerabilities, sensitive information, or code with strict licenses, leading to potential security and privacy issues.

This paper investigates an important problem: *to what extent do code models* memorize *their training data?* We conduct an empirical study to explore memorization in large pre-trained code models. Our study highlights that simply extracting 20,000 outputs (each having 512 tokens) from a code model can produce over 40,125 code snippets that are memorized from the training data. To provide a better understanding, we build a taxonomy of memorized contents with 3 categories and 14 subcategories. The results show that the prompts sent to the code models affect the distribution of memorized contents. We identify several key factors of memorization. Specifically, given the same architecture, larger models suffer more from memorization problem. A code model produces more memorization when it is allowed to generate longer outputs. We also find a strong positive correlation between the number of an output's occurrences in the training data and that in the generated outputs, which indicates that a potential way to reduce memorization is to remove duplicates in the training data. We then identify effective metrics that infer whether an output contains memorization accurately. We also make suggestions to deal with memorization.

CCS CONCEPTS

• **Software and its engineering** → **Software development techniques**; • **Computing methodologies** → **Artificial intelligence**; • **Security and privacy**;

KEYWORDS

Open-Source Software, Memorization, Code Generation

1 INTRODUCTION

As more large open-source datasets become publicly available [30, 38], code models [10, 21, 29, 50, 66], trained on billion lines of code, are now an important part of software engineering. The models automate a series of critical tasks such as defect prediction [60], code review [42], code generation [7] and software questions analysis [33, 34, 68]. These models have gone beyond academic exploration and have been widely deployed and used by a large number of users. For example, GitHub CoPilot [2], powered by the OpenAI Codex model [21], obtains over 400,000 subscribers in the first month of its release [63] and has already been used by over 1.2 million developers. In addition, the recently released models [3, 29, 41] demonstrate outstanding performance in software engineering tasks [58, 67] and has powered many tools, e.g., IDE plugins.

The impressive performance of these models can be attributed to the combination of advanced model architectures (e.g., state-of-the-art Transformer models with Despite the remarkable advancements, they are confronted with a set of privacy and legal challenges. For example, CoPilot was found to produce real people's names and physical addresses in its outputs [25]. It is also reported that Samsung employees accidentally leaked company secrets via ChatGPT [6], which raises concerns as ChatGPT retains records of these conversations and could potentially use this data for training its system [5].

The above concerns emphasize the importance of *exploring the capacity of large code models to memorize their training data*. A code model memorizes a string from its training data if there exists a prompt to make the model generate this string. This exploration is critical, especially given the fact that the training data for these models may come from a variety of sources. For example, large public datasets, such as the repositories hosted on GitHub, are made publicly available for training code models. These datasets may contain licensed code. If code models memorize and generate licensed code, and model users are not aware of this and utilize the code, it potentially results in a breach of the license agreements.

The training data includes 'software secrets' [13] such as passwords and API keys, which can be outputted to users and leveraged by attackers directly. For instance, as shown in our evaluation, a

code model can memorize the username and password to access a database in a public IP address. Furthermore, the training datasets may contain vulnerable or even malicious code. Such code could potentially be memorized and displayed to users, causing a serious risk to the security and integrity of software systems that adopt outputs of the code models. Therefore, it is crucial to understand the memorization in code models.

To investigate the memorization of code models, we first explore two open-source models: CodeParrot and CodeParrot-small [1]. CodeParrot is a GPT-2 model with 1.5 billion parameters trained to generate Python code. CodeParrot-small is a lightweight version of CodeParrot with 110 million parameters. We choose to analyze the two models because (1) their training data is available and is of a size feasible to conduct memorization analysis; (2) the models share the same architecture and dataset but come in two sizes, allowing us to investigate the impact of model size on memorization;

Our investigation begins by extracting a large number of outputs using different methods. We extract 20,000 outputs by feeding a special token called the 'start token' as a prompt into CodeParrot; this method provides no information to guide the generation. After identifying over 40,125 unique memorized code snippets using clone detection, we conduct an open card sorting study on a statistically representative number (381) of memorized contents to build a taxonomy with 3 categories and 14 subcategories. We find that 277 out of 381 memorized code snippets contain documentation (e.g., license information and docstring), and 239 out of 381 memorized code snippets contain code logic.

Additionally, we investigate factors that affect the memorization. The results show that CodeParrot memorizes more contents than CodeParrot-small, indicating that larger models have stronger memorization ability. Allowing a model to generate longer outputs can reveal more memorization. When the maximal length of outputs increases, the number of memorized contents increases as well. We also find a strong positive correlation between the number of a memorized content's occurrences in the training data and that in the outputs, which indicates that a potential way to reduce memorization is to remove duplicates in the training data.

Inspired by a study on memorization in language models [19], we investigate four metrics to infer whether an output contains memorization: (1) perplexity [22], (2) ratio of perplexity of two models, (3) ratio of perplexity to zlib [4], and (4) average perplexity of sliding windows. We compute these metrics to rank the outputs and analyze the top-100 ranking outputs. We find that the first 3 metrics are very effective in identifying memorized contents: over 97% of the top-100 outputs contain memorization. However, metrics (1) and (3) tend to rank outputs with license information higher, while metric (2) tends to rank outputs with code logic higher. Additionally, to demonstrate that memorization also exists in other code models, we analyze outputs from two popular code models that have already been deployed in practice: Incoder [29] and StarCoder [41]. We manually analyze the top 100 outputs from the two models ranked using metric (2). We search code snippets in the top 100 outputs on GitHub to see whether GitHub repositories contain the same code. If yes, it is an evidence that the model potentially memorize the code from its training data collected from GitHub. We find that 81% and 75% of the top-100 outputs from Incoder and StarCoder contains code snippets that can be found on GitHub, respectively.

Besides, 90.12% and 65.33% of memorization are related to code logic.

To summarize, this paper makes the following contributions:

- We systematically investigate the phenomena of memorization in large code models, highlighting the potential risks of memorization in code models.
- We empirically show the feasibility to extract a large number of memorized contents from a code model using simple methods. We categorize the memorized contents into 14 categories and analyze their distribution.
- We analyze the factors affecting memorization in code models. Also, we evaluate metrics to infer memorization.
- We show that deployed code models memorize training data as well. We also provide some suggestions on how to deal with potential risks brought by such memorization.

Paper structure. Section 2 provides the background and motivation. In Section 3, we detail our methodology for extracting, identifying and inferring memorization. Section 4 describes the experiment settings. We analyze the memorization by answering three research questions in Section 5. In Section 6, we provide some suggestions and discussion Next, Section 7 highlights relevant studies. We conclude our paper and present future work in Section 8.

2 BACKGROUND AND MOTIVATION

This section describes the code generation process and some motivating examples to study such memorization.

2.1 Code Generation Process

Modern code models typically utilize the Transformer architecture [64]. Many well-known models (such as InCoder [29] and codeparrot [1]) are built on the GPT-2 model [54], which is referred to as the *generative pre-trained transformer* model. The primary objective of such models is to generate code based on a given context (i.e., the *prompt* sent into the model's input layer). This is accomplished by training the model on a large corpus of datasets and letting the model learn a probability distribution that predicts the likelihood of the next token, given the context.

Let us assume that the context consists of a sequence of tokens, denoted by $p = \langle x_1, x_2, \cdots, x_n \rangle$. The model applies a softmax layer to compute the probability distribution of the next token, denoted by $P(x_{n+1}|p)$. Formally, this distribution is defined:

$$P(x_{n+1}|p) = \frac{e^{z_{n+1}}}{\sum_{i=1}^{k} e^{z_i}} \quad (1)$$

In the equation, z_i is the logit of the i-th token computed by the model. Then, the model uses a *decoding strategy* to decide the next token. A commonly adopted strategy is the top-k sampling [15]. The model selects the k most probable tokens from the probability distribution, and then chooses the next token from this new set of tokens. The parameter k plays an important role in balancing the trade-off between diversity and coherence in the generated output. Intuitively, a small k encourages the model to generate more fixed and coherent outputs, while a large k introduces randomness and diversity. This paper assumes users can control the code generation by modifying the k parameter, which is feasible even with only access to the API (e.g., OpenAI API allows k to be changed).

2.2 Memorization Definition

Memorization in the context of generation tasks refers to a model's ability to store and recall specific information or patterns it has encountered during its training. We define a string s as a piece of memorization as follows. Let $f : P \rightarrow S$ be a code model, where P is the space of prompts and S is the space of strings. Let $D \subseteq S$ be the training dataset. A string $s \in S$ is considered memorized if:

$$\exists p \in P; s \in f(p) \land s \in D \tag{2}$$

In other word, a string s is memorized if there exists a prompt p such that the model generates $f(p)$ when given p as input, and s appears both in the training dataset D and the generated output $f(p)$. In practice, the length of s should be substantially long. Otherwise, there might be many less useful memorized code such as variable names or boilerplate statements. We explain the threshold to decide memorization from code models in Section 3.2.

2.3 Memorization in Code Models

The issue of memorization is especially concerning and should warrant careful attention from the research community. We explain three scenarios where memorization may negatively impact the code models. Each scenario comes with an example of memorization we encountered during our experiments. We obfuscate the examples due to ethical considerations (e.g., protecting sensitive information and the identity of contributors of vulnerable code).

Scenario 1: Intellectual property issues. Some code models are trained on public repositories on GitHub, without paying much attention to the licensing terms of the code. Consequently, memorization in code models may cause intellectual property issues, such as violation of open-source licenses. When the model learns to reproduce specific code snippets from the training data instead of generalizing the underlying concepts, this leads to the generation of code that closely resembles copyrighted or licensed code, potentially infringing on the intellectual property rights of the original authors. Listing 1 shows an output from CodeParrot, which memorizes the original read function verbatim from a public repository that is licensed under Apache License 2.0. If a developer uses the generated code without proper compliance with the Apache License 2.0 terms, it may violate open-source licenses.

```
1  def read(self, iprot):
2    iprot.readStructBegin()
3    while True:
4      (fname, ftype, fid) = iprot.readFieldBegin()
5      if ftype == TType.STOP:
6        break
7      if fid == 1:
8        if ftype == TType.STRING:
9          .....
```

Listing 1. An example of code snippets generated by CodeParrot, which is protected by an Apache license.[1]

Scenario 2: Security vulnerabilities. The training data may contain vulnerable and even malicious code. After being trained on such data, code models might generate malicious code and exploitable bugs, or adopt poor security practices. If a model memorizes and outputs insecure code to users, it puts users and systems at risk

---

[1]https://github.com/Workiva/frugal

by opening them up to security threats. Listing 2 demonstrates potential security risks related to code model memorization. The memorized code is susceptible to SQL injection attacks because it directly uses the 'user_id' input in the SQL query without sanitizing or parameterizing it. An attacker could potentially manipulate the 'user_id' input to execute malicious SQL queries if this generated code is adopted by users.

```
1  def fetch_user_data(user_id):
2    connection = sqlite3.connect(<masked value>)
3    cursor = connection.cursor()
4    query = f'SELECT * FROM users WHERE id = {user_id}'
5          + <masked value>
6    cursor.execute(query)
```

Listing 2. An example of insecure code generated by CodeParrot. The identifiers are substituted and strings are masked due to ethical consideration.

Scenario 3: Leakage of sensitive information. Recent studies show that there exist a vast amount of software secrets in public repositories [13, 27], including API keys, passwords, etc. A code model that tends to memorize data can cause the leakage of sensitive information if it has been exposed to such information during training. Listing 3 provides an example that a code model memorizes a hardcoded public IP address, username, and password from its training data. On the one side, the code model exposes sensitive information, leading to potential security breaches and privacy violations. On the other side, even the end users do not use the generated information, users may use the program sketch and replace the password with their own, which is still an insecure development practice.

```
1  netowrk_config = {
2    'device_type': <masked value>,
3    'ip':    <masked value>,
4    'username': <masked value>,
5    'password': <masked value>
6  }
7  network_con = ConnectHandler(**netowrk_config)
8  print network_con.find_prompt()
```

Listing 3. An example of source code with sensitive information generated by CodeParrot, including sensitive information like public IP address, username, etc. Identifiers are substituted and strings are masked for ethical consideration.

2.4 Ethical Consideration

We emphasize our ethical responsibility by explicitly stating that our goal is to comprehend the memorization phenomenon rather than actively exploiting sensitive information within memorized contents. Consequently, we avoid utilizing 'targeted attacks' [9] (i.e., aiming to extract specific types of memorization from the training data) to identify or extract sensitive information deliberately. To further minimize the ethical concerns, we choose two open-source code models that are trained on publicly available datasets rather than models that are commercial. Throughout the research process, we ensure that we handle data and results in an ethical manner. For example, in Listing 2 and 3, we obfuscate identifiers and mask the sensitive information so that the contributors of the vulnerable code cannot be disclosed.

3 METHODOLOGY

This section explains the methodology of our study, including how to sample outputs from code models and how to detect memorization in code models.

3.1 Generating Outputs from Code Models

We consider four different strategies to generate outputs from code models: non-prompt generation, temperature-decaying generation, prompt-conditional generation, and two-step generation.

Non-Prompt Generation (NPG). We leverage the autoregressive nature of language models to generate outputs without an initial prompt. The generation process is described as:

(1) Language models usually have a special token to represent the beginning of a sentence (e.g., `<s>` in GPT-2). We initialize a token sequence with only this start-of-sentence token, denoted by $p = \langle x_1 \rangle$, where $x_1 = $ `<s>`.

(2) For each step $t = 2, 3, \cdots$, we compute the probability distribution of the next token given the current token sequence as input, $P(x_t|p)$ and decide the next token x_t. We then append the new token to the token sequence, updating the context for the next iteration: $p \leftarrow \langle p, x_t \rangle$.

(3) The above process repeats until a termination criterion is met. In this paper, the termination criterion is specified by the maximal length of output a model generates. Once the termination criterion is met, the generated token sequence is then decoded into human-readable code.

Temperature-Decaying Generation (TDG). The outputs generated by NPG can often be repetitive and lack of diversity. To increase the diversity, we can use a larger value for the *temperature* τ. The temperature value controls the level of diversity in outputs. Recalling the Equation 1, the temperature τ is used to scale the logits before applying the softmax function. Specially, the new distribution is computed as follows:

$$P(x_{n+1}|p) = \frac{e^{\frac{z_{n+1}}{\tau}}}{\sum_{i=1}^{k} e^{\frac{z_i}{\tau}}} \tag{3}$$

When τ is small, the probability distribution is compressed, which means that the model will be more likely to select the originally most probable tokens. Otherwise, the model will give less preference to the originally most probable tokens, producing more diverse outputs. However, maintaining a high temperature increases the probability of generating incoherent code and, as a result, produces less memorization. Carlini et al. [19] use the *temperature-decaying* strategy. Specifically, during the generation process, the temperature gradually decreases from a high value to a low value. It encourages the model to generate diverse outputs for the first several tokens and then gradually focus on generating coherent outputs.

Prompt-Conditioned Generation (PCG). This strategy requires designing a prompt for the code model. Generation from the start token tends to produce many contents that usually appear at the beginning of code files, e.g., license information. Besides, the users usually send prompts to the code model for completion rather than generating code from the start token. Therefore, we randomly choose a list of files from unseen testing data and parse the files

to extract the function definition statements. We feed the function definition statements as the prompt to the code model for completion.

Two-Step Generation (TSG). In NPG, a code model generates outputs from the start token, which produces a large number of license information and less amount of outputs relevant to the prompt. Thus, after we identify memorization using NPG, we send the most frequently appearing memorization as the prompt to the code model for completion to explore whether this can drive the model to generate more memorization.

3.2 Memorization Detection

After generating outputs from the code models, we detect whether the outputs contain memorization. Recalling the definition of memorization in Section 2.2, a string from model outputs is memorized if this string is also in the training data. In a previous study analyzing memorization in natural language models [19], the memorization is detected by performing fuzzy match (e.g., 3-gram fuzzy match) between the generated text and the training data. However, this strategy is not suitable to code models for the following reasons. Unlike natural language, source code has a more structured syntax with specific rules and conventions, which usually have unique constructs, keywords, and idioms that often occur together in specific sequences. A fuzzy match approach might flag these commonly occurring sequences as memorization despite the fact that they are simply inherent to the language.

We use the concept of code clone to determine whether an output contains memorization. A *Type-1 clone*, also known as an exact clone or an identical clone, refers to a specific type of code duplication in which two or more code fragments are exactly the same [56]. This means that the code fragments have the same sequence of tokens, including comments, whitespace, and formatting. Comparing to other types of clone (e.g., Type-2 clone), Type-1 clone means that a model produces output verbatim from the training data, which is a stronger evidence of memorization. However, very short Type-1 clones may not be considered memorization and they may not reveal useful information about the model. For example, the code snippet 'a = 1' may appear in many places in the training data. As a result, we only consider Type-1 clones that are longer than a threshold number of lines L.

Following a study [23] on analyzing the clones of code generated by deep learning-based code recommenders, we employ SIMIAN [32] for identifying Type-1 clones between the generated code and the training data. Although there are other clone detection tools, some of these tools are not publicly accessible, and others are unable to process incomplete code snippets, which are often produced by language models. By default, SIMIAN identifies clones that span at least six lines, which is chosen as the threshold L in our study.

3.3 Memorization Prediction

While *memorization detection* directly compares the output of a code model with source code in the training data, *memorization prediction* infers whether an output contains memorized content without accessing the training data. This task assumes that the training data is not accessible; this may simulate *adversarial* settings. For example, a malicious user may generate a large amount of outputs

and then infer which parts are more likely to be memorization. If such a malicious user can infer memorization correctly, information regarding the training data (e.g., private code or software secrets) can be leaked, putting a potential security risk. According to the study on analyzing memorization in natural language models [19], this paper adopts the following metrics to infer memorization.

Perplexity. Perplexity [22] measures how well a language model predicts a sample. A lower perplexity value indicates that the model is more confident in its predictions. Intuitively, a model is more confident on the example that it has seen during training. Therefore, a lower perplexity value may indicate that a model retrieves memorization.

PPL-PPL ratio. Language models tend to have low perplexity trivial memorization, e.g., repeated contents such as license information. From the perspective of risk assessment, exposing code logic leads to higher risks than exposing trivial memorization. Imagining that there are two code models: a small model and a large model, both trained on the same dataset. As shown in Section 5.2, a large model better memorizes more non-trivial contents than the small model. It means that on trivial memorization, both two models have small perplexity, but on non-trivial memorization, the large model has smaller perplexity than the small model. Therefore, we use the ratio of perplexity between the large model and the small model to infer memorization. We name this metric as PPL-PPL ratio (PPL for perplexity). The metric is computed as $\frac{log(P_l)}{log(P_s)}$, where P_s is the perplexity computed using the small model and P_l is the perplexity of the large model. A small ratio indicates that an output is more likely to contain non-trivial memorization.

PPL-zlib ratio. zlib [4] is a data compression library. The zlib entropy of a text is computed as the number of bits when the text is compressed with zlib library. A repetitive input has a small zlib entropy. We compute the PPL-zlib ratio: $\frac{log(P_v)}{zlib}$ (the ratio of the model perplexity and the zlib entropy). A small ratio indicates that the output content is more likely to be memorized but less likely to be repetitive [19].

Average PPL. Memorized contents may be surrounded by non-memorized outputs. Computing the perplexity of an output mixed of memorized and non-memorized content may not reflect the model's confidence. As stated in Section 3.2, clones spanning over 6 lines are considered as memorization in this paper. So we apply a sliding window of 6 lines to each output (moving by one line each time). We compute the perplexity of each window and then compute the Average PPL of all windows. We rank model outputs by the Average PPL in ascending order.

4 EXPERIMENT SETUP

4.1 Subject Code Models and Dataset

4.1.1 Models for Memorization Analysis. This paper first analyzes the contents that are exactly memorized from the training data (i.e., Type-1 clones) and the factors that affect such memorization. To achieve this goal, we investigate two models: CodeParrot and CodeParrot-small. The former is based on the GPT-2 model with 1.5 billion parameters, and the latter is a smaller version

Table 1. Model performance on the OpenAI's HumanEval benchmark [21]. Pass@n means the chance that a model provides a correct answer within n attempts.

Model	Size	Pass@n		
		n=1	n=10	n=100
CodeParrot	1.5B	3.80%	6.57%	12.78%
CodeParrot-small	110M	3.58%	8.03%	14.96%
PolyCoder	160M	2.13%	3.35%	4.88%
PolyCoder	400M	2.96%	5.29%	4.88%
GPT-Neo	125M	0.75%	1.88%	2.97%
GPT-Neo	1.3B	4.79%	7.47%	16.30%

of CodeParrot, which also uses the GPT-2 architecture but has fewer (110 million) parameters. Both models are trained on a collection of Python files from scratch. The two models are available on the HuggingFace model hub. Table 1 shows the performances of two investigated models and other code models of similar sizes on the OpenAI's HumanEval benchmark [21]. Although there are other available code models, this study selects CodeParrot and CodeParrot-small as the main research subjects for the following reasons.

First, the training data of some models can be either inaccessible or lack sufficient details, preventing a thorough analysis of their memorization capabilities. For example, InCoder [29] states the source of its training data but does not provide the exact dataset used for training. Second, while some models have available data, its extensive dataset, which includes multiple programming languages, complicates clone detection-based memorization analysis. For example, SantaCoder [10] uses a training set of 3TB and GPT-Neo [16] is trained on a diverse text dataset of 800GB. Additionally, SantaCoder only comes in one size, making it unsuitable for analyzing the impact of model size. Third, commercial models like CHATGPT exhibit superior results but cannot be analyzed due to the unavailability of their training data. Attempting to extract training data from these models can be potentially considered violating their terms of use as well. Lastly, CodeParrot is a popular model. Querying "CodeParrot" as the keyword on HuggingFace returns 419 models. In contrast, searching HuggingFace using "codebert" and "codet5", which are both widely evaluated models in the literature, returns only 126 and 124 models, respectively.[2]

The selected models are trained on the CodeParrot dataset, which was created with the GitHub dataset available via Google's BigQuery. This dataset contains approximately 22 million Python files and is 180 GB in size. Some preprocessing steps are conducted to clean the dataset. After removing whitespace in each file, the developers of CodeParrot find and remove exact duplicates by computing the hash of each file. The files whose first 5 lines contain the word 'auto-generated' are removed to remove the potentially generated files. Then, we follow the preprocessing tasks detailed in the original dataset repository. After data cleaning, the processed dataset is split into the training set and the validation set, which contain approximately 5 million files that are 50GB.

---

[2]The result is obtained on 25 July 2023.

4.1.2 Analyzing Memorization in Deployed Models. We also conduct a case study on more code models to analyze whether they can potentially memorize their training data as well. We select two additional models: Incoder [29] and `StarCoder` [41]. We consider these models for the following reasons. (1) The two models are of larger sizes (6B and 15.5B, respectively), popular, and demonstrate strong performance [29, 41]; and (2) These models have been deployed for practical usage. `StarCoder` has been deployed as an extension that can be used in IDEs like IntelliJ IDE[3]. Incoder is deployed and used within Meta [47]. Evaluating these models can help us understand the risks of using code models in practice.

4.2 Research Questions

In this study, we answer three research questions to investigate the risk of code models memorizing training data:

- **RQ1.** *What do code models memorize?*
- **RQ2.** *What factors affect memorization in code models?*
- **RQ3.** *How to infer whether an output contains memorized information?*

The first question aims to understand the extent and nature of information that code models tend to memorize, trying to categorize the memorized contents into different types. The second question analyze the factors that influence the memorization, which helps us understand the risks associated with code models leaking their training data. The third question focuses on infering whether an output contains memorized information from its training data. By addressing these research questions, we seek to get a deeper understanding of the risks associated with code models leaking their training data, along with practical techniques to identify and mitigate such risks. The following paragraphs describe the experiment design for each research question.

4.2.1 RQ1. What do code models memorize?
Motivation. Code models are trained on a large amount of data collected from various sources, including both public open-source repositories and private code bases; both of which are prone to contain sensitive information. Understanding what code models memorize helps avoid the risks associated with code models leaking their training data. For example, if code models can memorize and output information such as function implementation and even private data, then the risk of code models leaking their training data is high.

Experiment Design. We make a code model generate a large amount of source code according to different generation strategies described in Section 3.1. For this experiment, we choose the `CodeParrot` model as the experiment subject. For each strategy, we generate 20,000 outputs; each output has a maximal length of 512 tokens. Then we analyze the Type-1 clones spanning more than 6 lines that appear in both the training data and the outputs, i.e., memorized contents.

In order to gain a deeper understanding of the memorization, we design an annotation study to classify the memorization into different categories. We identify 40,125 unique clones (ranging from 6 to 53 lines) from 20,000 outputs produced by Non-Prompt

Generation (NPG). We use a widely adopted sample size calculator[4] with a confidence level of 95% and a confidence interval of 5 to obtain a statistically representative sample size of 381. We conduct an open-card sorting study [14], a well-established technique for generating meaningful groupings of data. Two authors of this paper discuss with each other to categorize the cards and a senior author resolves the disagreement and merge low-granularity categories into higher-level ones. We then classify the memorized contents into three main categories: *Documentation*, *Code Logic*, and *Others*, each having 2 to 8 sub-categories. Each memorized content can contain multiple categories.

4.2.2 RQ2. What factors affect memorization in code models? **Motivation.** RQ1 shows that code models memorize different types of contents and that the content of the prompts affects the model outputs, motivating us to further investigate the other factors that impact the memorization of code models. Knowing the relevant factors helps us better understand code models and highlight potential directions to mitigate memorization.

Experiment Design. We consider the following factors that may affect the memorization of code models.

- **Model size**: Previous research [50] shows that if two models share the same architecture and the same dataset, the larger model has stronger capacity. In this experiment, we compare the memorization power of `CodeParrot` and its smaller version `CodeParrot-small`.
- **Top k sampling**: Code models generate the next token by choosing from the top-k most likely tokens. A small (large) k value can generate more fixed (diverse) outputs. We try 4 settings: 5, 10, 20, and 40 to investigate the effect of k.
- **Output length**: The length of model outputs (i.e., number of tokens generated) may also impact the memorization of code models. We try 4 settings: 256, 512, 768, and 1024.
- **Number of generated outputs**: If we can query the model for many times and obtain many outputs, the model may expose more memorized information.
- **Occurrences in the training data**: The number of occurrences of a code snippet in the training data can affect the memorization. Intuitively, if the model sees a training example frequently, it may overfit this example and produce the frequently occurring patterns at higher probability.

4.2.3 RQ3. How to infer whether an output contains memorized information?
Motivation: For the aforementioned research questions, we assume to have complete access to the training data in order to analyze memorization. This assumption is practical when conducting analysis as model developers. However, in certain adversarial situations, such as when malicious users aim to extract training data from code models, the attacker typically does not have knowledge of the training data. Therefore, we investigate how to infer whether an output from code models contains memorization without querying the training data.
Experiment Design: We use the methods described in Section 3.3 to rank the outputs from code models by using four different metrics.

Following the setting in the paper that propose these metrics [19], we look into the 100 top-ranking outputs from the ranked lists and compute the ratio of the outputs containing memorized contents.

4.3 Implementation Details

We fetch the `CodeParrot` and `CodeParrot-small` models from the HuggingFace model hub and run them an NVIDIA GeForce A5000 GPU with 24 GB of memory. To implement the temperature-decaying generation, we use an initial high temperature of 20.0. Each time the model generates a new token, we decrease the temperature by 1.0 until it reaches 1.0 (i.e., 20 tokens are generated), after which we keep the temperature at 1.0. We download the datasets of the two models released by the authors of `CodeParrot`.[5] As the clone detection tool we use consumes a large amount of time when analyzing many files, we merge all the files in the training data and split the merged file into 53 chunks and run clone detection in parallel. On a machine having AMD EPYC 7643 CPU with 48 cores and 512GB memory, analyzing memorization for each 20,000 examples in parallel takes one hour. We encourage researchers with more computational resources to replicate our study at a larger scale.

5 EXPERIMENT RESULTS

5.1 RQ1. What do code models memorize?

Memorization Detection and Categorization. Model outputs by different generation strategies include a significant number of code fragments memorized from the training data. Note that, on average, approximately 43% and 57% of 20,000 outputs from `CodeParrot-small` and `CodeParrot` contain memorized information, respectively. Following the procedure described in Section 4.2.1, we randomly sample 381 memorized outputs for each generation strategy and categorize them. Table 2 shows the total number of occurrences of each category obtained using different generation strategies. We observe that the *license and copyright*, as well as the *import statements*, are the most frequent categories. More specifically, the *license and copyright* category appears 216, 150, 288, and 222 times in the outputs generated by NPG, TSG, TDG, and PCG, respectively. The *import statements* sub-category appears 129, 211, 80, and 112 times, respectively. The reasons for their higher frequency than other categories are two-fold. On the one side, such information usually appears at the beginning of a file, which is easy to been seen and memorized. On the other side, there exist many duplicates of such information (e.g., developers declare the same license in many files).

Each generation strategy shows different distributions of outputs. For example, we find that TSG tends to generate less license and copyright information (decrease from 216 to 150) and more code logic (increase 239 from 330) than NPG. The reason is that TSG selects the most frequent memorization found by NPG (which is the license information) as the prompt into code models. The model will complete the content after the prompt and thus skips the license information to generate more code logic related contents, such as import statements and method definitions.

Table 2. Occurrences of memorization extracted with Non-Prompt Generation (NPG), Temperature Decaying Generation (TDG), Prompt-Condition Generation (PCG), and Two-Step Generation (TSG). The number of annotated outputs for each sampling method is 381. Note that one output may map to multiple categories of memorization.

Category	NPS	TSS	TDS	PCS
Documentation	277	247	333	315
License and Copyright	216	150	288	222
Docstring	30	55	10	50
Usage Intruction	31	42	35	43
Code Logic	239	330	100	304
Import Statements	129	211	80	112
Method Definition	39	41	2	41
Method Calls	25	30	4	60
Class Definition	17	19	4	18
Conditional Statements	14	10	4	33
Exception Handling	9	11	4	32
Testing	4	7	2	5
Print Statement and Log	2	1	0	3
Others	60	36	45	57
Configuration	38	23	25	52
Unable to Classify	22	13	20	5

Temperature has an impact on the diversity of sampled outputs. For example, TDG drives the models to produce higher portion of license information. The potential reason is that using a higher temperature diversifies model outputs. For those contents that are harder to memorize (e.g., code logic), adding more diversity makes the model fail to generate them. Consequently, most of the outputs generated by TDG are license information while it generates the least code logic among all generation strategies. Although the prompts used in PCG already include the license and import statements, PCG produces a similar number of license and import statements as NPG. This is because PCG generates code logic to complete the function first, after which the model generates license and import statements. This observation contrasts with the conclusion drawn in NLP models that higher temperature values helps the model produce more non-trivial information [19], which is the code logic in our case.

Interestingly, we find that PCG produces more memorization related to code logic than other generation strategies. For example, the occurrences of *exception handling* increase from 9 in NPG to 32 in PCG, and that of *method calls* increase from 25 to 60. PCG to some extent simulates the behavior of users when interacting with code models: writing some code like function definition and letting the model complete. Our results suggest that code models may produce much memorization when interacting with users, posing data leakage risk.

Sensitive Information Detection. Our experiment also reveals sensitive information memorized by the code models. One of the examples is shown in Listing 4 in which we can identify a private key that might expose financial accounts. As discussed in Section 2, there might be other types of sensitive information, but we focus on counting IP addresses, email addresses, and hash keys as they

```
1  class SignMessagesTest(BitcoinTestFramework):
2    def set_test_params(self):
3      self.setup_clean_chain = True
4      self.num_nodes = 1
5    def run_test(self):
6      message = <masked value>
7      self.log.info(<masked value>)
8      priv_key = <masked value>
```

Listing 4. An example of source code with private keys. The identifiers are substituted and strings are masked because of ethical consideration.

are explicitly identifiable compared with other types of information such as licensed code or vulnerabilities. We use detect-secrets,[6] a popular open-source tool to scan three types of sensitive information in the 20,000 outputs generated by NPG. After removing local IPs and emails containing "example," we find 25 IP addresses, 914 emails, and 25 keys that are exactly same with the information from the training data as well (i.e., they are memorized).[7] Our findings warn that such sensitive information requires to be properly handled (e.g., removed) before training code models.

> **Answers to RQ1**: Code models successfully memorize different types of content, including both documentation and code logic. The license and copyright, as well as import statements, are the most frequently appearing memorization. With proper strategies, one can drive the model to produce more code logic-related memorization. Code models memorize sensitive information such as private keys and emails.

5.2 RQ2. What factors affect memorization in code models?

Model size and top k sampling. The model size has a substantial impact on the frequency of memorized outputs and k parameters affect the results of memorization as well. Following the procedure described in Section 5.2, we use the Non-Prompt Generation (NPG) to generate 20,000 outputs from CodeParrot and CodeParrot-small, respectively. Figure 1a illustrates the numbers of unique memorization in the outputs of the two models with different k values. The orange and blue curves represent the results of CodeParrot and CodeParrot-small, respectively. We observe that curve of the larger model CodeParrot is consistently above the other curve, suggesting that it memorizes more contents. Besides, the larger CodeParrot also memorizes longer contents. Given the same setting (k=20, output length is 512), CodeParrot produces memorization with a maximal of 60 lines, while CodeParrot-small only produces memorization with a maximal of 45 lines.

We analyze how the k value affects the memorization of code models. We try 4 settings of k values: 5, 10, 20, and 40. The length of each output is set as 256 tokens. As shown in Figure 1a, when k is low (e.g., 5), both models produce less memorization. As a small k means that a model chooses from a small and relatively fixed set of tokens, which may lead to the generation of memorization with more duplicates (i.e., the number of unique memorization is smaller). When k increases from 5 to 10, the outputs become

---

[6]https://github.com/Yelp/detect-secrets
[7]The detection results of other generation strategies are included in '`./PPI`' folder in the replication package.

Table 3. Number of unique memorized-code for a given set of parameters: (1) number of tokens and (2) number of outputs.

Model	# outputs	256	512	768	1024
CodeParrot-small	5,000	6,666	9,080	11,041	14,031
	10,000	10,627	14,655	17,664	22,243
	15,000	14,015	19,444	23,863	29,133
	20,000	16,966	23,574	29,204	35,363
CodeParrot	5,000	9,785	14,645	18,325	22,570
	10,000	16,062	24,345	32,519	37,448
	15,000	21,560	32,666	42,853	50,127
	20,000	26,420	40,125	51,059	61,787

more diverse and consequently produce more unique memorized outputs, which can be observed from the figure that the number of unique memorization increases, reaching a peak at k=10. However, when k continues to increase, the outputs become more diverse and differ more from the training data, which leads to a decrease in the number of unique memorization. As a result, models memorize less alongside k increases from 10 to 40.

Output length and the number of outputs. Longer outputs can expose more memorized contents according to our experiments. Given a model, we keep other factors unchanged and vary the maximal number of tokens of the outputs: 256, 512, 768, and 1024. The results are shown in Table 3. When the length of outputs increases, the number of unique memorization also increases. This trend is consistent for both models on different numbers of outputs. The impact of output length is larger on the larger model. Specifically, given the same setting (5,000 examples and increase the output length from 256 to 1024), the number of unique memorization increases by 7,365 (from 6,666 to 14,031 for CodeParrot-small, while increases by 12,785 (from 9,785 to 22,570) for CodeParrot.

Table 3 also shows that when we extract more outputs from the models, the number of unique memorization increases as well. We let the CodeParrot model to generate 10 million outputs (512 tokens each) and plot the trend of the number of unique memorization with the number of outputs in Figure 1b. The blue curve shows the total number of unique memorization, while the orange curve shows the number of newly identified unique memorization by generation additional 200,000 outputs each time. We see a 'diminishing return' pattern: although more memorization can be identified when more outputs are generated, the number of newly identified memorization decreases. The 10 million outputs contain over 8 million unique memorization that is longer than 6 lines.

Occurrences in the training data. Code fragments more frequently appearing in the training data are more likely to be generated by the code models. We count the number of occurrences of each memorized content in the training data and the model outputs and visualize the distribution in Figure 1c. Spearman's rank correlation test obtains a correlation coefficient of 0.804 ($p < 0.01$), indicating a strong positive correlation between the two variables. We conduct the Pearson correlation test to measure the linear relationship. We obtain a correlation coefficient of 0.752 ($p < 0.01$), which indicates a strong positive linear correlation between the two variables. The finding implies that when the model is exposed to a higher frequency of certain contents in its training data, it tends to
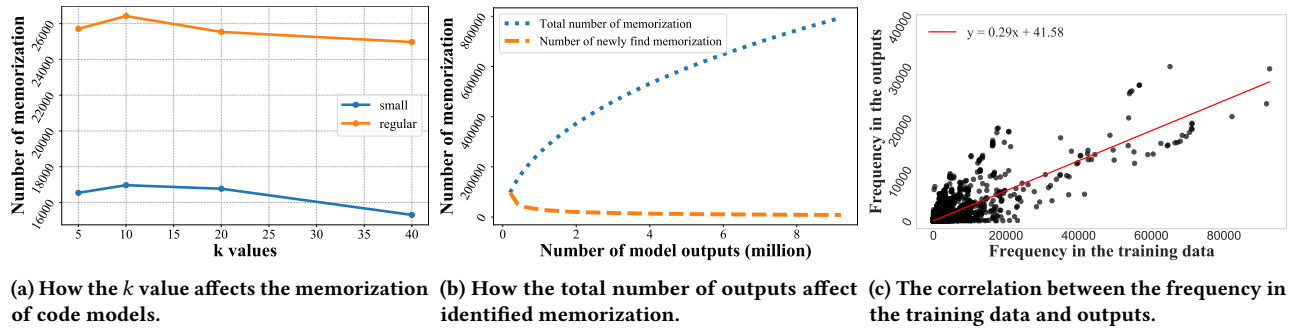
(a) How the k value affects the memorization of code models.

(b) How the total number of outputs affect identified memorization.

(c) The correlation between the frequency in the training data and outputs.

Figure 1. Three factors that affect memorization in code models.

memorize it and produces that content more frequently in its outputs, highlighting to importance of conducting data de-duplication before training code models.

Answers to RQ2: We identify five factors affecting memorization: (1) the model size given the same architecture; (2) the hyperparameter k in top-k sampling; (3) the length of generated outputs; (4) the number of generated outputs; (5) the frequency of the code fragment in the training data.

5.3 RQ3. How to infer whether an output contains memorized information?

Prediction performance. We use the four metrics described in Section 3.3 to rank the outputs generated by the code models and analyze the 100 top-ranking outputs. Table 4 shows the ratio of the top 100 outputs that contain memorization. We find that generally three out of four metrics can accurately rank the memorized outputs at the top. Note that, on average, approximately 43% and 57% of 20,000 outputs from `CodeParrot-small` and `CodeParrot` contain memorized information. When the output length is set as 256 tokens, all the top 100 outputs ranked using perplexity, PPL-PPL ratio, and PPL-zlib ratio, contain memorization. Ranking the outputs using PPL-zlib ratio achieves the best performance, with all the top 100 outputs containing memorization. The method of PPL-PPL ratio is slightly less effective. When the output length is 512, its detection accuracy is 79%. But in other settings, its accuracy is close to 100%. However, the Average PPL metric is less effective in detecting memorization, especially on long outputs.

We further analyze what types of memorization are inferred by these methods. We conduct another annotation study to count the occurrences of different categories of memorization in the 100 top ranked outputs for `CodeParrot-small` model (output length is 256). We find that all the memorized outputs ranked by the perplexity and PPL-zlib Ratio are license information. However, using PPL-PPL ratio can rank more diverse memorization in the top; 12% of memorized contents contain code logic.

Answers to RQ3: Three out of four metrics can infer memorization accurately. The PPL-PPL ratio ranks more diverse memorization at the top positions, while memorization highly ranked by the other metrics is mainly license information.

```
1   # Bitcoin Cash (BCH)    qpz3<masked value>5nuk
2   # Ether (ETH) -         0x84<masked value>c9FB
3   # Litecoin (LTC) -      Lfk5<masked value>qPvu
4   # Bitcoin (BTC) -       34L8<masked value>BtTd
5
6   # contact :- nnheo@example.com
```

Listing 5. An example of output from `StarCoder` that contains a substituted email address: 'nnheo@example.com'. The addresses of cryptocurrency are still memorized. We only show the first and last 4 digits to protect privacy.

6 DISCUSSION

6.1 Memorization in Deployed Code Models

We analyze the memorization of two models that have already been deployed in Meta and IntelliJ IDE: `Incoder` and `StarCoder`. Both `Incoder` and `StarCoder` are trained on large-scale datasets (159GB and 3TB) that cover multiple programming languages. However, the training data of `Incoder` is not directly available and the magnitude of the training data for `StarCoder` presents significant challenges when attempting to analyze Type-1 clones. Note that the authors of the models [29, 41] mention that the training data is curated from GitHub. Thus, we use the search function provided by GitHub as a proxy and manually confirm the memorization of these models. If GitHub returns the exact code snippets (spanning over 6 lines), we consider an output contains memorization.

For each model, we extract 20,000 outputs using NPG. According to the finding from RQ3 that *the PPL-PPL ratio ranks more diverse memorization at the top positions*, we rank the outputs using the PPL-PPL ratio and analyze the top 100 outputs. Then, we manually search GitHub for each line of code in the top 100 outputs. We find that 81% and 75% of the selected outputs from `Incoder` and `StarCoder` are confirmed to be memorized. Besides, 90.12% and 65.33% of memorization from the two models is related to code logic. The results reveal that there is risk that the two models can memorize and expose their training data. This fact can be worrying especially when models that trained on private repositories (e.g., internal code in a company) are deployed for public usage.

Both `StarCoder` and `InCoder` adopt preprocessing methods to remove personal identifiable information and other software secrets in the training data. They use regular expressions to detect and substitute emails. The benefit of doing so is reflected in the model outputs. For example, Listing 5 shows an output from `StarCoder`. We search GitHub for this output; GitHub returns the exact code

except a different email address. We believe this output is memorized from the training data, and it substitutes the email address in the training data. However, we find that the addresses of cryptocurrency (i.e., the masked values in Listing 5) are still memorized. This highlights the need of appropriately dealing with more types of sensitive information in the training data before training code models and releasing them to the public.

6.2 Suggestions on Memorization in Code Models

Based on our findings, we provide the following suggestions to deal with memorization in code models.

1. Data Sources: Users from various platforms such as GitHub and Stack Overflow should have the right to explicitly indicate whether their data can be utilized for AI model training. Otherwise, the code models may memorize the data and expose the data to the other users. Platforms should support such declarations, allowing users to specify their preferences at different levels of granularity. For example, a user may allow the main branch of a repository to be used for training, but not the development branch as it may include experimental and unfinished code. To support the declarations, GitHub can allow users to include a separate file that outlines the rights associated with a repository or provide an option to specify the rights in the account settings.

2. Data Collection and Processing: Data collectors should pay attention to data license information and avoid using data with strict or unclear licenses for training code models. Appropriate preprocessing of the dataset is necessary, which may include but not limited to detecting and removing personal identifiable information and other software secrets reasonably. Our study shows that duplicates in the training data are more likely to be memorized. Allamanis [11] suggests that duplicate code may cause adverse effects on code models. Removing duplicates in the training data can help reduce memorization. Additionally, collectors can employ defensive methods to prevent privacy attacks (e.g., data poisoning).

3. Additional Information for Outputs: Code model developers should also offer users sufficient information when they use the model. For instance, model developers should detect whether an output is likely to be memorized from the training data. If so, users should be informed of the output's origin and its copyright information. This not only prevents users from inadvertently violating open-source regulations but also empowers them to make informed decisions about the quality of the output. For example, users may avoid using an output if it is from a poorly-maintained repository. Furthermore, our study shows that larger models memorize more contents. Therefore, the service providers of large code models (e.g., ChatGPT) should take measures to limit the queries to the model to prevent potential privacy breaches.

4. Opt-out Mechanisms: Dataset providers should allow users to determine if their data has been included in a dataset. We also suggest building a tracking system to build a connection between a model and its training data. Such a tracking system helps identify what models include a specific user's data for training. As mentioned in Suggestion 1, users should have the right to prevent

their code from being used in training and be able to provide their consent. For data that has already been used, users should have the right to declare or withdraw their consent. Their data should not only be removed from the dataset but also from the models. Moreover, it is necessary to develop a certification mechanism to evaluate whether a model has removed the data from its knowledge properly.

5. Multidisciplinary Collaborations: We believe that a multidisciplinary approach is essential to effectively address the outcomes of data memorization in code models. AI researchers can contribute developing models that minimize memorization while maintaining the model performance. Software engineering researchers can focus on open-source data management and understand user requirements. Legal professionals provide guidance on copyright and other regulatory requirements to ensure compliance and the protection of users' rights.

6.3 Threats to Validity

Threats to Internal Validity. Internal validity refers to the extent to which a study is free from errors or biases that could invalidate the results. Our study leverages Type-1 clone detection to identify memorization. However, the accuracy of clone detection tool may affect our results. To mitigate this threat, we choose a state-of-the-art clone detection tool, SIMIAN [32], which has been used to analyze code clones between the model output and training data. Another threat is that using Type-1 clone detection may lead to under estimation of memorization. For example, a code model may not fully memorize a code snippet and produce a slightly modified version of the code snippet. In this case, this output is not treated as memorization.

Threats to External Validity. External validity refers to the extent to which the results of a study can be generalized to other settings. This paper evaluates two code models based on the GPT-2 architecture, which is widely used in open-source code models [10, 29]. However, the results obtained in this paper may not generalize to other models. To alleviate this threat, we analyze the memorization in two popular code models that are deployed in practice and demonstrate that they also suffer from memorization issues. We also plan to conduct experiments on more models and datasets in the future.

Table 4. The performance of different methods in inferring whether an output contains memorized information. The numbers represent the ratio of top 100 outputs ranked by each method; i.e., 1.0 indicates that the method successfully computes rankings top 100 as memorized source code.

Methods	Models			
	CodeParrot		CodeParrot-S	
	t=256	t=512	t=256	t=512
Perplexity	1.00	0.92	1.00	1.00
PPL-PPL Ratio	1.00	0.79	0.97	0.98
PPL-zlib Ratio	1.00	1.00	1.00	1.00
Average PPL	0.74	0.24	0.86	0.36

7 RELATED WORK

In this section, we present an overview of studies relevant to this paper, including (1) pre-trained models of code and (2) privacy concerns in AI models.

7.1 Pre-trained Code Models and Analysis

Large language models like BERT [24, 43] and GPT [17, 54] have excelled in NLP tasks, inspiring pre-trained models for code. Code-BERT [28] and a list of similar models (including GraphCode-BERT [31], CuBERT [39], etc) are developed to produce code embeddings to support downstream tasks like defect prediction. Many code models leverage the GPT architecture [17, 54] to conduct generation tasks. CodeGPT, which trains the GPT-2 architecture on CodeSearchNet [38], is proposed as a baseline model in the CodeXGLUE benchmark [45]. Code models with larger sizes and better performance are also proposed, like InCoder [29], Code-Gen [50]. A list of studies [51, 72, 73] have empirically shown the outstanding performance of these models on various tasks.

Researchers have also studied the limitations of code models and threats, highlighting the need to study their trustworthiness [44]. One of the limits is that a malicious user can generate adversarial examples [36, 59, 69, 71] by adding semantic preserving transformations [12, 52] to fool code models. Another threat to the training data is data poisoning. An attacker can simply inject a small portion of malicious code into the training data to poison the datasets [40, 55, 65, 70]; consequently, the obtained code models have backdoors that can be triggered by specific inputs. Data poison attacks can also harm the performance of code models [49, 61].

Software engineering researchers have noticed the potential issues brought by memorization in code models. Al-Kaswan et al. [8] discuss the security, privacy, and license implications of memorization in code models. Our paper conducts the first systematic study of memorization in code models, by categorizing memorized content, analyzing factors that affect memorization, etc. Rabin et al. [53] also evaluate memorization in code models, however, their concept of memorization diverges from ours. In their paper, a code model learns from noisy dataset by memorizing the noise and fails to generalize to test data, which is more similar to the notion of overfitting. Our paper investigates to what extent code models memorize and output the training data verbatim.

7.2 Privacy Concerns in AI Models

Henderson et al. [35] discuss some ethical challenges in data-driven dialogue systems, one of which is the potential privacy violation. Carlini et al. [18] evaluate unintended memorization in Google's Smart Compose that can complete emails. Feldman [26] uses the long-tail theory to explain the memorization behavior of deep learning models. Zhu et al. [74] analyzes memorization behavior for a LSTM-based neural language model. One important privacy concern related to memorization is the feasibility of data extraction attack, which aims to extract the training data from the model. An important privacy threat related to memorization is the data extraction attack, which aims to extract the training data from the model. Carlini et al. [19] extract around 600 examples of memorization from the GPT-2 model like URLs, phone numbers, etc. Al-Kaswan et al. [9] propose a targeted attack on extracting data from the

GPT-Neo model. To the best of our knowledge, our paper presents the first empirical study of memorization in large pre-trained code models. Another important privacy concern is the membership inference attack (MIA), which aims to infer whether a specific data sample is used to train a model. Shokri et al. [57] propose a black-box MIA method on machine learning-based classification models. Hisamoto et al [37] operate MIA on machine translation systems. Chen et al. [20] produces a taxonomy of membership inference attacks against various generative models. Mireshghallah et al. [46] use MIA to quantify the privacy risk of masked language models like BERT. Researchers also propose defensive method to mitigate the risks by MIA. For example, Tang et al. [62] propose to use model ensemble to mitigate the MIA risk. Nasr et al. [48] leverage adversarial regularization to enhance membership privacy.

8 CONCLUSION AND FUTURE WORK

This paper conducts a comprehensive study to examine the memorization in large pre-trained code models. We develop a taxonomy for memorized contents, consisting of 3 primary categories and 14 subcategories. The study uncovers several key factors affecting memorization: the model size, the length of outputs, etc. We also discover a strong positive correlation between the frequency of an output's appearance in the training data and that in model outputs. This suggests that eliminating duplicates in the training data could potentially reduce memorization. Furthermore, we identify effective metrics that accurately determine whether an output contains memorized contents and offer recommendations on how to address the issue of memorization in code models. Additionally, we evaluate memorization in two popular models that are already deployed.

In future work, we plan to study larger code models and more programming languages. We also plan to explore different strategies to mitigate memorization in code models.

> The replication package is provided for replication at **https://github.com/yangzhou6666/Privacy-in-Code-Models**, which should not be used for malicious purposes like conducting data extraction attacks.

REFERENCES

[1] [n. d.]. codeparrot (CodeParrot). https://huggingface.co/codeparrot
[2] [n. d.]. GitHub copilot · your AI pair programmer. https://github.com/features/copilot
[3] [n. d.]. Introducing chatgpt. https://openai.com/blog/chatgpt
[4] [n. d.]. A Massively Spiffy Yet Delicately Unobtrusive Compression Library. https://zlib.net/. Accessed on March 27, 2023.

[5] [n. d.]. What is chatgpt? https://help.openai.com/en/articles/6783457-what-is-chatgpt

[6] 2023. Samsung employees accidentally leaked company secrets via chatgpt: Here's what happened. https://gizmodo.com/chatgpt-ai-samsung-employees-leak-data-1850307376

[7] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2655–2668.

[8] Ali Al-Kaswan and Maliheh Izadi. 2023. The (ab)use of Open Source Code to Train Large Language Models. arXiv:2302.13681 [cs.SE]

[9] Ali Al-Kaswan, Maliheh Izadi, and Arie van Deursen. 2023. Targeted Attack on GPT-Neo for the SATML Language Model Data Extraction Challenge. arXiv:2302.07735 [cs.CL]

[10] Loubna Ben Allal, Raymond Li, and Denis Kocetkov et al. 2023. SantaCoder: don't reach for the stars! arXiv:2301.03988 [cs.SE]

[11] Miltiadis Allamanis. 2019. The Adverse Effects of Code Duplication in Machine Learning Models of Code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) (*Onward! 2019*). Association for Computing Machinery, New York, NY, USA, 143–153. https://doi.org/10.1145/3359591.3359735

[12] Leonhard Applis, Annibale Panichella, and Arie van Deursen. 2021. Assessing Robustness of ML-Based Program Analysis Tools using Metamorphic Program Transformations. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1377–1381. https://doi.org/10.1109/ASE51524.2021.9678706

[13] Setu Kumar Basak, Lorenzo Neil, Bradley Reaves, and Laurie Williams. 2023. SecretBench: A Dataset of Software Secrets. In *Proceedings of the 20th International Conference on Mining Software Repositories (MSR '23)*. 5 pages.

[14] Andrew Begel and Thomas Zimmermann. 2014. Analyze This! 145 Questions for Data Scientists in Software Engineering. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) (*ICSE 2014*). Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/2568225.2568233

[15] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. *Advances in neural information processing systems* 28 (2015).

[16] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow*. https://doi.org/10.5281/zenodo.5297715 If you use this software, please cite it using these metadata..

[17] Tom Brown, Benjamin Mann, and Nick et al. Ryder. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901.

[18] Nicholas Carlini, Chang Liu, 'Ulfar 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, USENIX Association, Santa Clara, CA, USA, 267–284.

[19] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting Training Data from Large Language Models. In *USENIX Security Symposium*.

[20] Dingfan Chen, Ning Yu, Yang Zhang, and Mario Fritz. 2020. GAN-Leaks: A Taxonomy of Membership Inference Attacks against Generative Models. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) (*CCS '20*). Association for Computing Machinery, New York, NY, USA, 343–362. https://doi.org/10.1145/3372297.3417238

[21] Mark Chen, Jerry Tworek, and Heewoo Jun et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[22] Stanley F. Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language* 13, 4 (1999), 359–394. https://doi.org/10.1006/csla.1999.0128

[23] Matteo Ciniselli, Luca Pascarella, and Gabriele Bavota. 2022. To What Extent Do Deep Learning-Based Code Recommenders Generate Predictions by Cloning Code from the Training Set?. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) (*MSR '22*). Association for Computing Machinery, New York, NY, USA, 167–178. https://doi.org/10.1145/3524842.3528440

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[25] Delton Ding. 2021. GitHub Copilot provided me with a picture of someone's ID card? PIC.TWITTER.COM/RRLE1UXF6U. https://twitter.com/DeltonDing/status/1423651446340259840

[26] 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* (Chicago, IL, USA) (*STOC 2020*). Association for Computing Machinery, Association for Computing Machinery, New York, NY, USA, 954–959. https://doi.org/10.1145/3357713.3384290

[27] Runhan Feng, Ziyang Yan, Shiyan Peng, and Yuanyuan Zhang. 2022. Automated Detection of Password Leakage from Public GitHub Repositories. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 175–186. https://doi.org/10.1145/3510003.3510150

[28] 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, 1536–1547.

[29] 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 *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=hQwb-lbM6EL

[30] 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).

[31] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu andz Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

[32] Simon Harris. n.d.. Simian. https://www.harukizaemon.com/simian.

[33] Junda He, Zhou Xin, Bowen Xu, Ting Zhang, Kisub Kim, Zhou Yang, Ferdian Thung, Ivana Irsan, and David Lo. 2023. Representation Learning for Stack Overflow Posts: How Far are We? arXiv:2303.06853 [cs.SE]

[34] Junda He, Bowen Xu, Zhou Yang, DongGyun Han, Chengran Yang, and David Lo. 2022. PTM4Tag: Sharpening Tag Recommendation of Stack Overflow Posts with Pre-Trained Models (*ICPC '22*). Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3524610.3527897

[35] Peter Henderson, Koustuv Sinha, Nicolas Angelard-Gontier, Nan Rosemary Ke, Genevieve Fried, Ryan Lowe, and Joelle Pineau. 2018. Ethical Challenges in Data-Driven Dialogue Systems. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society* (New Orleans, LA, USA) (*AIES '18*). Association for Computing Machinery, Association for Computing Machinery, New York, NY, USA, 123–129. https://doi.org/10.1145/3278721.3278777

[36] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2022. Semantic Robustness of Models of Source Code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 526–537. https://doi.org/10.1109/SANER53432.2022.00070

[37] 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. https://doi.org/10.1162/tacl_a_00299

[38] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[39] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*. PMLR, 5110–5121.

[40] Jia Li, Zhuo Li, Huangzhao Zhang, Ge Li, Zhi Jin, Xing Hu, and Xin Xia. 2022. Poison Attack and Defense on Deep Source Code Processing Models. https://doi.org/10.48550/ARXIV.2210.17029

[41] Raymond Li, Loubna Ben Allal, and Yangtian Zi et al. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL]

[42] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating Code Review Activities by Large-Scale Pre-Training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) (*ESEC/FSE 2022*). Association for Computing Machinery, New York, NY, USA, 1035–1047. https://doi.org/10.1145/3540250.3549081

[43] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[44] David Lo. 2023. Trustworthy and Synergistic Artificial Intelligence for Software Engineering: Vision and Roadmaps. arXiv:2309.04142 [cs.SE]

[45] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021). arXiv:2102.04664

[46] 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

[47] Vijayaraghavan Murali, Chandra Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, and Nachiappan Nagappan. 2023. CodeCompose: A Large-Scale Industrial Deployment of AI-assisted Code Authoring. *arXiv preprint arXiv:2305.12050* (2023).

[48] Milad Nasr, Reza Shokri, and Amir Houmansadr. 2018. Machine Learning with Membership Privacy Using Adversarial Regularization. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 634–646. https://doi.org/10.1145/3243734.3243855

[49] Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco, Massimiliano Di Penta, and Davide Di Ruscio. 2021. Adversarial Attacks to API Recommender Systems: Time to Wake Up and Smell the Coffee?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 253–265. https://doi.org/10.1109/ASE51524.2021.9678946

[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 *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=iaYcJKpY2B_

[51] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An Empirical Comparison of Pre-Trained Models of Source Code. arXiv:2302.04026 [cs.SE]

[52] Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (2021), 106552.

[53] Md Rafiqul Islam Rabin, Aftab Hussain, Mohammad Amin Alipour, and Vincent J. Hellendoorn. 2023. Memorization and generalization in neural code intelligence models. *Information and Software Technology* 153 (2023), 107066. https://doi.org/10.1016/j.infsof.2022.107066

[54] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[55] G. Ramakrishnan and A. Albarghouthi. 2022. Backdoors in Neural Models of Source Code. In *2022 26th International Conference on Pattern Recognition (ICPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 2892–2899. https://doi.org/10.1109/ICPR56361.2022.9956690

[56] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[57] 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)*. 3–18. https://doi.org/10.1109/SP.2017.41

[58] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. arXiv:2301.08653 [cs.SE]

[59] Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. 2021. Generating Adversarial Computer Programs using Optimized Obfuscations. *ICLR* 16 (2021), 209–226.

[60] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An Empirical Study of Deep Learning Models for Vulnerability Detection. arXiv:2212.08109 [cs.SE]

[61] 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* (Virtual Event, Lyon, France) *(WWW '22)*. Association for Computing Machinery, New York, NY, USA, 652–660. https://doi.org/10.1145/3485447.3512225

[62] Xinyu Tang, Saeed Mahloujifar, Liwei Song, Virat Shejwalkar, Milad Nasr, Amir Houmansadr, and Prateek Mittal. 2022. Mitigating membership inference attacks by {Self-Distillation} through a novel ensemble architecture. In *31st USENIX Security Symposium (USENIX Security 22)*. 1433–1450.

[63] Roberto Torres. 2022. GitHub copilot adds 400K subscribers in first month. https://www.ciodive.com/news/github-copilot-microsoft-software-developer/628587/

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[65] Yao Wan, Shijie Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Dezhong Yao, Hai Jin, and Lichao Sun. 2022. You See What I Want You to See: Poisoning Vulnerabilities in Neural Code Search. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1233–1245. https://doi.org/10.1145/3540250.3549153

[66] 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, EMNLP 2021*.

[67] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for 0.42 each using ChatGPT. arXiv:2304.00385 [cs.SE]

[68] Chengran Yang, Bowen Xu, Ferdian Thung, Yucen Shi, Ting Zhang, Zhou Yang, Xin Zhou, Jieke Shi, Junda He, Donggyun Han, and David Lo. 2023. *Answer Summarization for Technical Queries: Benchmark and New Approach*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3551349.3560421

[69] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural Attack for Pre-Trained Models of Code. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1482–1493. https://doi.org/10.1145/3510003.3510146

[70] Zhou Yang, Bowen Xu, Jie M. Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo. 2023. Stealthy Backdoor Attack for Code Models. https://doi.org/10.48550/ARXIV.2301.02496

[71] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020).

[72] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An Extensive Study on Pre-Trained Models for Program Understanding and Generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 39–51. https://doi.org/10.1145/3533767.3534390

[73] Xin Zhou, DongGyun Han, and David Lo. 2021. Assessing Generalizability of CodeBERT. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 425–436. https://doi.org/10.1109/ICSME52107.2021.00044

[74] Derui Zhu, Jinfu Chen, Weiyi Shang, Xuebing Zhou, Jens Grossklags, and Ahmed E. Hassan. 2021. DeepMemory: Model-based Memorization Analysis of Deep Neural Language Models. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1003–1015. https://doi.org/10.1109/ASE51524.2021.9678871