Decoding Secret Memorization in Code LLMs Through Token-Level Characterization

Yuqing Nie[†]
Beijing University of Posts and Telecommunications
Beijing, China
jiangsha@bupt.edu.cn

Chong Wang[†]
Nanyang Technological University
Singapore, Singapore
chong.wang@ntu.edu.sg

Kailong Wang[*]
Huazhong University of Science and Technology
Wuhan, China
wangkl@hust.edu.cn

Guoai Xu[*]
Harbin Institute of Technology
Shenzhen, China
xga@hit.edu.cn

Guosheng Xu
Beijing University of Posts and Telecommunications
Beijing, China
guoshengxu@bupt.edu.cn

Haoyu Wang
Huazhong University of Science and Technology
Wuhan, China
haoyuwang@hust.edu.cn

Abstract—Code Large Language Models (LLMs) have demonstrated remarkable capabilities in generating, understanding, and manipulating programming code. However, their training process inadvertently leads to the memorization of sensitive information, posing severe privacy risks. Existing studies on memorization in LLMs primarily rely on prompt engineering techniques, which suffer from limitations such as widespread hallucination and inefficient extraction of the target sensitive information. In this paper, we present a novel approach to characterize real and fake secrets generated by Code LLMs based on token probabilities. We identify four key characteristics that differentiate genuine secrets from hallucinated ones, providing insights into distinguishing real and fake secrets. To overcome the limitations of existing works, we propose DESEC, a two-stage method that leverages token-level features derived from the identified characteristics to guide the token decoding process. DESEC consists of constructing an offline token scoring model using a proxy Code LLM and employing the scoring model to guide the decoding process by reassigning token likelihoods. Through extensive experiments on four state-of-the-art Code LLMs using a diverse dataset, we demonstrate the superior performance of DESEC in achieving a higher plausible rate and extracting more real secrets compared to existing baselines. Our findings highlight the effectiveness of our token-level approach in enabling an extensive assessment of the privacy leakage risks associated with Code LLMs.

I. INTRODUCTION

Large Language Models (LLMs) have revolutionized the field of natural language processing, enabling groundbreaking advancements in various domains. A notable sub-domain of LLMs is Code LLMs, which specialize in generating, understanding, and manipulating programming code. Code LLMs have found extensive applications in code completion, code generation, bug fixing, and code summarization, demonstrating their significant impact on software development practices and productivity [1]–[5]. Despite the remarkable capabilities of LLMs, they raise a significant privacy concern. Specifically, LLMs are trained on vast amounts of data sourced from a wide range of locations, including public repositories, forums, and websites. This comprehensive training process inadvertently leads to the memorization of sensitive and private information present in the training data [6]. Consequently, Code LLMs become prone to leaking sensitive and critical information (e.g., user credentials and secrets) during the code generation process, posing severe privacy risks and breaches.

Existing studies on memorization in LLMs have primarily employed prompt engineering techniques to elicit the memorized information from the models [7], [8]. While these approaches have provided valuable insights, they suffer from two main limitations. First, forcing or eliciting the LLM to output secret information might lead to widespread hallucination [9], where the LLM generates false secrets, resulting in a high number of false positives. This can significantly hinder the accurate assessment of the true extent of memorization and privacy leakage risks. Second, the prompt engineering based approaches generat output randomly, which can be inefficient in extracting the target sensitive information memorized by LLMs. This randomness may lead to repeated extraction of similar secrets across multiple queries while missing other unique secrets, potentially underestimating the true extent and variety of sensitive information memorized by LLMs and hindering an effective assessment of the associated privacy risks. To fully grasp the intricacies of memorization in LLMs and overcome these limitations, it is imperative to delve deeper into the internal workings of these models and examine the issue at a more granular level.

Our Work. In this paper, we present a novel approach to characterize real and fake secrets generated by Code LLMs based on token probabilities. Through extensive observation and analysis, we have identified several key characteristics that distinguish genuine secrets from hallucinated ones, such as the stabilization of real secret tokens at high probabilities (C1), higher overall probabilities of real secret tokens compared to fake ones (C2), more pronounced probability advantages of real secret tokens (C3), and the ability to identify certain fake secrets early in the decoding process using secret strength metrics like Shannon entropy (C4). While we illustrate these characteristics using Google API Keys, it is important to note that they are generalizable to various types of secrets.

To overcome the aforementioned limitations from existing

[†]These authors contributed equally to this work (co-first authors).
[*]Corresponding author.

works in the literature, we propose a method named DESEC that leverages token-level features derived from the identified characteristics to guide the token decoding process. DESEC consists of two stages: (1) Offline Token Scoring Model Construction, which utilizes a proxy Code LLM to generate training data and train a scoring model to predict the likelihood score that a token belongs to a real secret, and (2) Online Scoring Model Guided Decoding, which leverages the scoring model to predict a score for the tokens at each decoding step, combining it with the original LLM-predicted probability to reassign token likelihoods and guide the selection of tokens.

Through extensive experiments on five state-of-the-art Code LLMs (StableCode, CodeGen2.5, DeepSeek-Coder, CodeLlama, and StarCoder2) and a diverse dataset of 1200 code files containing various types of secrets across multiple programming languages, we demonstrate the superior performance of DESEC compared to existing baselines. DESEC achieves an average Plausible Rate (PR) of 44.74% across the five Code LLMs, significantly outperforming HCR (10.98%) and BS-5 (23.60%). Moreover, DESEC successfully extracts a total of 1076 real secrets from the victim models, surpassing the numbers obtained by HCR (230 real secrets) and BS-5 (845 real secrets). These results highlight the effectiveness of our token-level approach in guiding the decoding process to generate more diverse and accurate secrets, enabling a more comprehensive assessment of the privacy leakage risks associated with Code LLMs.

Contributions. We summarize the contributions as follows:

- **Identification of key characteristics distinguishing real and fake secrets.** We identify four generalizable key characteristics that differentiate genuine secrets from hallucinated ones generated by Code LLMs, based on token probabilities and secret strength metrics.
- **Development of a token-level approach for secret extraction.** We propose DESEC, a two-stage method that constructs an offline token scoring model using a proxy Code LLM and employs it to guide the decoding process by reassigning token likelihoods based on token-level features derived from the identified characteristics.
- **Extensive evaluation on state-of-the-art Code LLMs.** We evaluate DESEC on four state-of-the-art Code LLMs using a diverse dataset, demonstrating its superior performance in achieving a higher plausible rate and extracting more real secrets compared to existing baselines.

Ethical Considerations. In conducting this study, we prioritized ethical considerations to ensure responsible research practices. We strictly limited our use of the derived secrets to validating their authenticity without exploiting or misusing them in any way. Furthermore, we have taken great care to mask all the information presented in this paper, including any personal or sensitive data, to prevent potential leakage and real damage. By adhering to these ethical principles, we aim to contribute to the understanding of memorization in Code LLMs without causing any unintended consequences.

II. BACKGROUND AND RELATED WORK

A. Code LLMs

Code LLMs, based on large language model technology and trained on vast code datasets, are designed to understand and generate programming code [10]. They show significant potential in tasks such as code generation, autocompletion, and error detection [1]–[5]. Researchers are dedicated to enhancing their capabilities [3], [11]–[15], leading to the emergence of various Code LLMs, including Codex [16], CodeParrot [17], CodeBERT [18], CodeGPT [19], and Codellama [20]. These models have become indispensable tools for software developers worldwide, with applications such as Google's CodeSearchNet [21] using CodeBERT [18] for code search, Visual Studio Code integrating GitHub Copilot [22] and CodeGeeX [23] as extensions, and Kaggle's AI Code Competition using Codex to help beginner programmers learn to code.

B. Memorization in LLMs

LLMs can remember a large amount of training data during the training process. This memorization mechanism enables the model to reproduce information from the training data when generating text. For a large model f trained on a dataset D, given an input sequence $x = (x_1, x_2, \ldots, x_n)$, the model generates an output sequence $y = (y_1, y_2, \ldots, y_m)$. The parameters of the model f are denoted as θ.

If there exists a prompt p such that: $f(p, \theta) = s, s \in D$, then the model f is said to have memorized the string s. Since s strictly exists in the training set D, this phenomenon is summarized as Exact Matching in the study [24]. Additionally, Lee et al. [25] defined a more lenient phenomenon called Approximate Matching: for $f(p, \theta) = s'$, if there exists a corresponding string s in the training set D such that s and s' are within a specified edit distance, then the model f is also said to have memorized the string s.

Many existing works have explored memorization in LLMs. Feldman [26] proposed that memorization of labels is necessary for near-optimal generalization error in data following natural distributions. Studies [27]–[29] reveal that attackers can perform training data extraction attacks by querying LLMs. Carlini et al. [30] investigated factors influencing LLM memorization, finding that increases in model capacity, data repetition, and prompt length significantly enhance memorization. Memorization also exists in Code LLMs, with studies [31], [32] confirming that code models can memorize their training data. Yang et al. [33] conducted a systematic study on memorization in code models, discussing factors such as data repetition and model size. For Multimodal Large Language Models, Chen et al. [34] developed metrics to measure data leakage during multimodal training.

The memorization in Code LLMs leads to privacy issues such as the reproduction of sensitive information and the leakage of proprietary code. Al-Kaswan et al. [35] proposed a targeted attack to identify whether a given sample in the training data can be extracted from the model. Niu et al. [7] explored the impact of temperature values on the generation of private information by GitHub Copilot, finding that about

8% of the prompts resulted in privacy leaks. Huang et al. [8] proposed HCR, in which Copilot generated 2702 hard-coded credentials and CodeWhisperer produced 129 keys during code completion tasks. Yao et al. [36] introduced CodeMI, a membership inference method that determines whether a given code sample is present in the training set of a target black-box model by analyzing the output features of shadow models.

III. TOKEN-LEVEL SECRET CHARACTERIZATION

We initially conduct a study to investigate the characteristics of real secrets and fake secrets generated by Code LLMs at token level, aiming to answer the research question:

RQ1: What are the token-level characteristics of secret memorization in Code LLMs?

A. Motivation

While existing researches have shown that Code LLMs can memorize secrets or privacy from training datasets [37]–[39], the characteristics of these memorized secrets remain underexplored. Additionally, given that Code LLMs may generate fake secrets due to hallucination issues [8], it is necessary to explore the differences between the characteristics of real and fake secrets [40].

To this end, we conduct an in-depth analysis of secret memorization in Code LLMs at the token level. We aim to extract the 6 most common types of commercial secrets included in the prior study [8], which together constitute 85.8% of all secrets. For ease of management, these secrets follow structured formats that can be defined using regular expressions. Table I lists these 6 types of secrets along with their corresponding regular expressions.

B. Study Setup

We construct code completion prompts and input them into Code LLMs to predict the missing secrets based on these prompts. During this process, we collect the predicted token probabilities at each decoding step and analyze their characteristics.

1) Source File Collection: For the 6 types of secrets, we use their regular expressions to search for code files containing them from open-source repositories through SourceGraph [41]. We choose SourceGraph for two reasons: (i) It integrates a vast number of open-source code repositories;

(ii) It supports efficient code search using regular expressions, which perfectly meets our requirement for searching secrets.

In the search process, we only consider 5 languages, namely HTML, Java, JavaScript, PHP, and Python, which are commonly used in web applications that may require the 6 types of secrets for online API requests. After duplication, 31,978 code files are identified as of June 16, 2024. To ensure data quality, we apply the filtering rules adopted in the StarCoder project [42] . Files that meet the following criteria are excluded:

- Files with an average line length exceeding 100 characters or a maximum line length exceeding 1000 characters.
- Files with less than 25% alphabetic characters.
- Files containing the string "`<?xml version=`" within the first 100 characters.

```
# Create your views here.
def mapapp(request):
    if request.method == 'GET':
        message = request.session.get('message')
        request.session['algorithms'] = None
        context = {
            'message':message,
            'GMAPS_API_KEY': 'AIza
```

Fig. 1: An Example of Completion Prompt

- For HTML files, we filter out files with less than 20% visible text or fewer than 100 characters of visible text.

These filtering operations remove approximately 3.85% of the files. Subsequently, we perform an additional examination of the characters surrounding the matched content to filter out the mismatched code files. For example, the middle part (highlighted in blue) of "...kBdAIza...BVV..." matches the regular expression for Google API Key, but the surrounding characters indicate that it is not a valid Google API Key. Ultimately, 66.57% of the original search results are retained, resulting a set of 25,353 code files. Table II shows the language and secret distribution in the final results.

We randomly select 200 code files for each type of secret, totaling 1,200 files, for subsequent method evaluation (Section V). The proportion of each programming language within each type of secret remains consistent with its proportion in the raw data. As a result, 24,133 code files are retained for analysis in this study.

2) Completion Prompt Construction: From the remaining code files, we randomly sample 400 files for each type of secret to construct completion prompts. Note that the numbers of files for STSK, SIWU, TCSI, and ACAK are fewer than 400, so we additionally collect files in other languages like Go and C using SourceGraph to reach 400. In total, 2,400 code files are used to construct completion prompts.

For each code file, we locate the secret's position and remove all content after the fixed secret prefix (*e.g.* "`AIza`" for Google API Key). We then retain the line containing the secret and up to 7 preceding lines as the completion prompt, which provides sufficient context without overwhelming the model based on the preliminary observations and previous study [43]. For example, as illustrated in Fig. 1, the fixed prefix "`AIza`" of a found Google API Key is retained, while all subsequent content is removed. After processing all code files, we obtain 2,400 completion prompts.

3) LLM-based Secret Generation: We feed these 2,400 prompts into a **Proxy Code LLM** to perform the code completion and predict the missing Google API Keys. In this study, we select StarCoder2-15B as the proxy LLM, which is a 15B-parameter model trained on multiple programming languages from The Stack v2 [44], supporting various tasks such as code completion and code generation. StarCoder2-15B is chosen as the proxy LLM because it shares the same common decoder-only architecture as the other Code LLMs in our study. This ensures consistency in studying secret token generation.

StarCoder2-15B generates a token sequence step by step

TABLE I: Target Secret Types and Their Formats

Secret Type	Acronym	Format (Regular Expression)	Token Constraint
Google API Key	GAK	AIza[0-9a-zA-Z\-_]{35}	[0-9a-zA-Z\-_]+
Google OAuth Client ID	GOCI	[0-9]{12}-[0-9a-z]{32}\.apps\.googleusercontent\.com	[0-9a-z\-\.]+
Stripe Test Secret Key	STSK	sk_test_[0-9a-zA-Z]{24}	[0-9a-zA-Z] +
Slack Incoming Webhook URL	SIWU	https:\/\/hooks.slack.com\/services\/[0-9a-zA-Z+\/]{44,46}	[0-9a-zA-Z+\/]+
Tencent Cloud Secret ID	TCSI	AKID[0-9a-zA-Z]{32}	[0-9a-zA-Z]+
Alibaba Cloud Access Key ID	ACAK	LTAI[0-9a-zA-Z]{20}	[0-9a-zA-Z]+

TABLE II: Distribution of Collected Code Files

	HTML	Java	JavaScript	PHP	Python	Total
GAK	6,297	965	5,194	1,135	1,533	15,123
GOCI	6,402	237	1,571	201	366	8,777
STSK	4	30	214	127	194	569
SIWU	10	31	83	58	163	345
TCSI	7	93	89	35	53	277
ACAK	4	174	38	21	25	262
Total	12,724	1,530	7,189	1,577	2,334	25,353

based on an input prompt. At each step, it generates a token probability distribution based on the context provided by the prompt and the previously predicted tokens. To avoid generating erroneous secrets that violate the format constraints in Table I, we set the probabilities of invalid tokens to 0. We then apply beam search with a width of 5 to select the next token, keeping the token sequence with the highest likelihood score as the final generated secret. The generation stops once the target secret's required length (character number) is reached. During the generation process, we record the predicted token probability distribution of each step for further analysis.

4) Generated Secret Verification: For each generated secret, we validate it through a combination of online API request and GitHub code search as follows:

- **API Request:** We develop verification scripts that send connection requests to the relevant services using the secret and observe the returned status codes. If the services are connected successfully, the secret is real and still active.
- **GitHub Search:** If the secret cannot be validated through an API request, we use GitHub Code Search [45] to search for it. If it can be found through this search and is not merely a usage example (*e.g.* "AKIDz8...EXAMPLE"), it is also considered a real memorized secret.

The verified dataset consists of 622 secrets: 311 real and 311 fake. It includes 100 real and 100 fake secrets each for Google API Key, Stripe Test Secret Key, and Tencent Cloud Secret ID (totaling 600), plus 11 real and 11 fake secrets for Google OAuth Client ID (due to limited availability). Slack Incoming Webhook URLs and Alibaba Cloud Access Key IDs are excluded as the model does not generate real secrets for these types.

C. Characterization

Based on our observation and analysis of token probabilities, we characterize the real and fake secrets generated by the Code LLM as follows. Note that these characteristics are generalizable to different types of secrets, we simply use Google API Keys to illustrate (see our replication package [46] for STSK, TCSI, and GOCI).
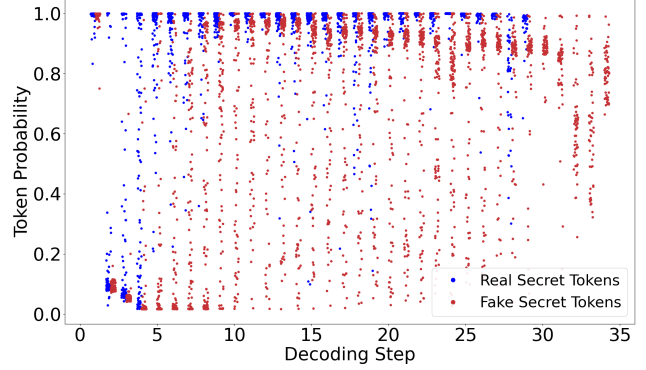
Fig. 2: Token Probability Scatter Plot for Google API Keys
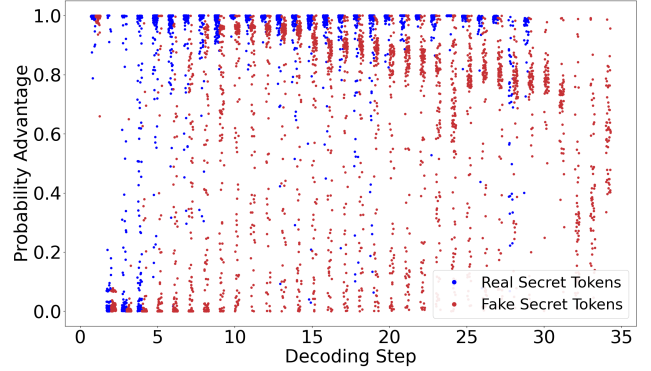
Fig. 3: Token Probability Advantages Scatter Plot for Google API Keys

C1: Tokens in real secrets tend to stabilize at relatively high probabilities after initial few decoding steps. Fig. 2 shows the scatter plot of token probabilities for both real and fake secrets (blue and red points respectively) along the decoding steps, where each point $(i, prob)$ represents a token at the i-th decoding step with a probability of $prob$. Typically, the probabilities of the initial few steps (e.g., the first five) for real secret tokens are relatively low. However, as the decoding progresses, the probability values rapidly increase and stabilize at a relatively high level. For instance, at the 10th step, tokens of the 100 real secrets have probabilities over 0.9. This observation can be attributed to the model's initial uncertainty in earlier steps; however, as it decodes more tokens and aligns with memorized sequences, its confidence increases quickly. Besides, an interesting discovery is that real secret tokens typically consist of multiple characters, while fake ones often have one single character. Consequently, real secrets decode in fewer steps and terminate earlier.

C2: Tokens in real secrets generally have higher probabilities than those in fake ones. According to Fig. 2, the overall probabilities of real secret tokens are generally higher

than those of fake secret tokens, with the latter often more scattered between 0 and 1. For example, at the 10-th step, the probability values of many fake secret tokens are evenly distributed between 0 and 0.9. This can be explained by the internal memory mechanism of LLMs. In addition, we conducted t-tests for this characteristic, and the results indicate a significant probability difference between real tokens and fake tokens for all secret types ($p \ll 0.01$).

C3: Probability advantage of tokens in real secrets is typically more pronounced than in fake ones. The probability advantage of a token at a given decoding step is defined as the difference between its probability and the probability of the next token in the distribution, ranked by descending probabilities. Fig. 3 shows a scatter plot of token probability advantages for both real and fake secrets along the decoding steps, where each point (i, adv) represents a token at the i-th decoding step with a probability advantage of adv. Similar to **C2**, the overall probability advantages of real secret tokens are generally higher than those of fake secret tokens. In addition, we conducted t-tests for this characteristic, and the results indicate a significant probability advantage difference between real tokens and fake tokens for all secret types ($p \ll 0.01$).

C4: Certain fake secrets can be identified early in decoding process using secret strength metrics like Shannon entropy. Code LLMs often generate weak secrets containing continuously repeated tokens due to the decoding objective of maximizing likelihood [47]. As the repetition of characters or tokens increases during the decoding process of such fake secrets, the Shannon entropy of the generated token sequence continuously decreases. This observation inspires us to continuously inspect the Shannon entropy during the decoding process and avoid certain fake secrets at an early stage to increase the likelihood of generating real secrets. Although entropy is commonly used to assess the strength of secrets, we are the first to integrate it directly into the LLMs' decoding process, rather than as a post-processing step. For example, if the current decoding step token sequence is "SyA2-", and "2" and "x" are the two tokens with the highest probabilities predicted by the Code LLM, we can discard "2" and choose "x" as the next token since appending "2" would decrease the sequence's entropy, thus preventing Code LLMs from falling into token repetition and producing fake secrets like "AIzaSyA2-2-2-2...".

> **Answer to RQ1**
>
> This study reveals that real and fake secrets generated by Code LLMs exhibit different token-level characteristics (**C1-C4**). This insight motivates us to leverage token-level features to improve the extraction of secrets memorized in Code LLMs.

IV. Methodology

Based on our characterization, we propose DESEC (see Fig. 4 for an overview), a method to extract secrets from Code LLMs by guiding the token decoding process with token-level features derived from **C1**, **C2**, **C3**, and **C4**. These features capture the characteristics of tokens that help determine whether they belong to a real secret. DESEC consists of two stages: (1) **Offline Token Scoring Model Construction**, which utilizes a proxy Code LLM to generate training data and train a scoring model to predict the likelihood score that a token belongs to a real secret; and (2) **Online Scoring Model Guided Decoding**, which leverages the scoring model to predict a score for the tokens at each decoding step. The score is then combined with the original LLM-predicted probability to reassign the token likelihoods, guiding the selection of tokens.

A. Token-Level Features

We first define four features for each token during the decoding process of Code LLMs, as follows:

Step Index ($step\_idx$): According to **C1**, real secret tokens exhibit distinct probability distributions at different decoding steps. To capture this, we use the step index of a token in the decoding process as a feature. For example, in Fig. 5, the token "2" generated at step 5 has a Step Index of 5.

Average Probability (avg_prob): According to **C2**, real secret tokens generally have higher probabilities than fake ones. To capture this while addressing the instability of using a single token's probability, we use the average probability of tokens selected in previous decoding steps as a feature. Compared with token's probability sequence, the average probability could reduce computational complexity while ensuring interpretability (see our replication package [46]). In Fig. 5, the probabilities of tokens up to "2" are 0.99, 0.18, 0.03, 0.02, and 0.03, with an average of 0.25, which is set as the Average Probability for token "2".

Probability Advantage ($prob\_adv$): According to **C3**, a token's probability advantage often indicates whether it belongs to a real secret. We capture this as a feature by calculating the difference between the token's probability and that of the next token in the probability distribution. In Fig. 5, at step 5, the probability of token "2" is 0.03, and the next token "DR" has a probability of 0.01. Thus, the Probability Advantage for token "2" is set to 0.02.

Entropy Ratio ($entp\_ratio$): According to **C4**, the changing trend of Shannon entropy in the predicted sequence during decoding can help filter out fake secrets early. We capture this by using the ratio of the current step's entropy to the previous step's entropy as a feature. In Fig. 5, for the token "2" generated at the current step, the corresponding string is "SAy2-2" with a Shannon entropy of 2.25. The previous step's string was "SAy2-" with a Shannon entropy of 2.32. Thus, the Entropy Ratio for token "2" is set to 0.97 (2.25/2.32).

Feature Vectorization. We form a feature vector for a token based on its four features as follows:

$$\mathbf{feat} = <step\_idx, avg\_prob, prob\_adv, entp\_ratio>$$

For example, the feature vector for the token "2" in Fig. 5 is:

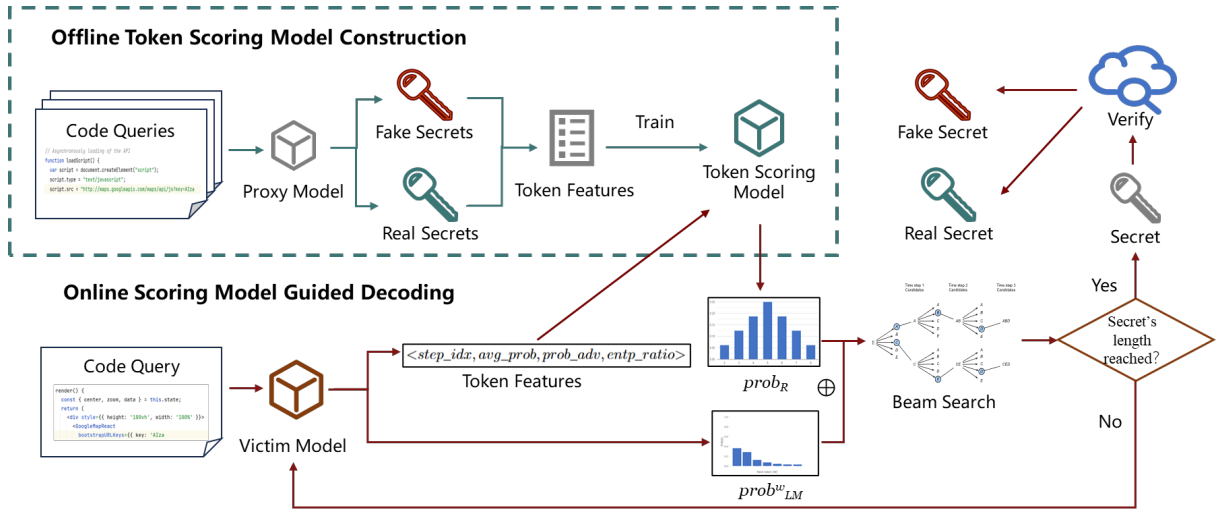$$\mathbf{feat} = <5, 0.25, 0.02, 0.97>$$
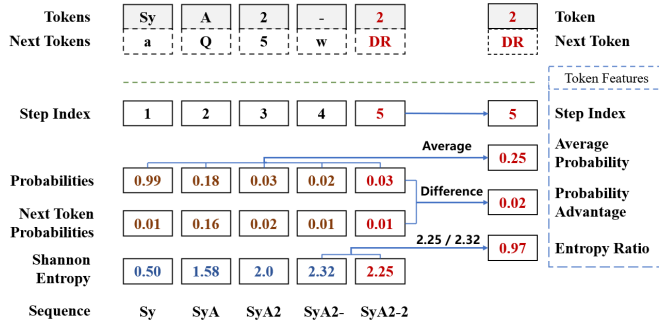
Fig. 4: Overall Workflow of DESEC

Fig. 5: Token Features Extraction Process

B. Token Scoring Model Construction

To combine the four independent features and assess the probabilities of tokens belonging to real secrets, we train a token scoring model.

Training Data Construction. We follow a process similar to our characterization study setup (Section III-B). We create completion prompts for each secret type based on searched code files and feed them into a ***Proxy Code LLM*** to generate candidate secrets. The generated candidates are validated and split into a real set $\mathcal{R}$ and a fake set $\mathcal{F}$. During the proxy Code LLM's token decoding process, we extract the feature vector **feat** for each token, resulting in two vector sets: $\textbf{Feat}_\mathcal{R}$ and $\textbf{Feat}_\mathcal{F}$, corresponding to $\mathcal{R}$ and $\mathcal{F}$, respectively.

Scoring Model Training. To effectively distinguish between tokens of real and fake secrets, we train a Linear Discriminant Analysis (LDA) [48] model to find a combination of the four features that maximizes the differentiation between $\textbf{Feat}_\mathcal{R}$ and $\textbf{Feat}_\mathcal{F}$. We choose this linear model over complex models like Multi-Layer Perceptron (MLP) [49] because our feature vector for each token contains only four features, and using non-linear models can lead to overfitting issues with low-dimensional feature vectors.

We prioritize the resource-efficient LDA rather than other complex models due to the immediacy required in decoding.

After training on $\textbf{Feat}_\mathcal{R}$ and $\textbf{Feat}_\mathcal{F}$, the resulting combination can be used by the scoring model to predict the probability that a feature vector **feat** belongs to real secrets:

$$prob_\mathcal{R} = \textbf{SM}(\textbf{feat}) \qquad (1)$$

C. Scoring Model Guided Decoding

Using the scoring model, we guide decoding process of the *Victim Code LLM*, as detailed in Algorithm 1, to enhance the likelihood of generating real secrets. The algorithm follows the traditional beam search decoding process, with additional enhancement steps (highlighted in red) including *Masking Invalid Tokens* (line 6), *Extracting Token Features* (line 9), *Calling Scoring Model* (line 10), and *Combining Probabilities* (line 11). The last three additional steps constitute the process referred to as *Scoring Model Guided Probability Re-weighting*. We first briefly introduce the overall process based on beam search and then describe the goals and the details of the enhancement steps.

Overall Beam Search Process. The overall decoding process follows the key steps of beam search:

- ***Hypothesis Pool Initialization*** (line 1). First, the process initializes a pool $Beam$ to maintain multiple hypotheses, where each hypothesis is a pair of $(seq, score)$, consisting of a token sequence seq and a likelihood score $score$. At the beginning, there is only one hypothesis $(pmpt, 0)$ in $Beam$, where $pmpt$ is the input prompt.

- ***Hypothesis Expansion*** (lines 5-16). For each hypothesis $(seq, score)$ in $Beam$, it is expanded to B new hypotheses which are added to a candidate list $Cands$. Specifically, the process first selects the top B tokens ($Toks$) from the LLM-predicted token probability distribution (lines 7-9) and then appends each selected token tok to seq (line 14), updating the likelihood score $score$ with tok's log probability $\log(prob_{LM})$ (line 15).

- ***Hypothesis Ranking and Pruning*** (lines 17-21). After expanding all hypotheses in $Beam$, there are B times new candidate hypotheses in the candidate list $Cands$. The hypotheses are ranked based on their likelihood scores (line 17), and the top B hypotheses are selected to update the hypothesis pool $Beam$ (line 21).

Algorithm 1: Scoring Model Guided Decoding

Data: Prompt $pmpt$, Beam Size B, Length Limit L
Result: Secret sec

1   $Beam \leftarrow [(pmpt, 0)]$ *// initialize hypothesis pool*
2   $step \leftarrow 0$
3   **while** *character # of sequences in Beam $< L$* **do**
4      $step \leftarrow step + 1$
5      $Cands \leftarrow []$ *// initialize candidate list*
6   **foreach** $(seq, score) \in Beam$ **do**
7        $\mathbf{p} \leftarrow \mathbf{LM}(seq)$ *// get probabilities for next token*
8        $\mathbf{p} \leftarrow$ MASKINVALIDTOKS$(\mathbf{p})$ *// mask invalid tokens*
9        $Toks \leftarrow$ TOPK$(\mathbf{p}, B)$ *// get top-B tokens by probability*
10   **foreach** $(tok, prob_{LM}) \in Toks$ **do**
11          $feat \leftarrow$ EXTRACTFEATS$(tok, step, seq, \mathbf{p})$
           // extract features
12          $prob_{\mathcal{R}} \leftarrow$ **SM**$(feat)$ *// call scoring model*
13          $prob \leftarrow prob_{LM}^{w} \times prob_{\mathcal{R}}$ *// combine probabilities*
14          $seq \leftarrow seq \oplus tok$
15          $score \leftarrow score + \log(prob)$
16          $Cands$.append$((seq, score))$
17      $Cands \leftarrow$ rank$(Cands)$ *// rank hypotheses by score*
18   **if** $step \leq K$ **then**
19        $Beam \leftarrow Cands$
20   **continue**
21      $Beam \leftarrow Cands[: B]$ *// select top-B hypotheses*
22   $sec \leftarrow$ ARGMAX$(Beam)$ *// select the best hypothesis*
23   **return** sec

- *Final Secret Selection* **(line 22).** These steps are iteratively performed until the length limit L of the target secret type is reached. After that, the hypothesis with the highest likelihood score in $Beam$ is selected, and the token sequence is returned as the final secret.

Our Enhancements. We outline the goals and details of the enhancement steps as follows.

- *Masking Invalid Tokens* **(line 8).** In traditional beam search, the LLM predicts the probability distribution $\mathbf{p}$ based on the existing token sequence seq (line 7), and the next step is to select the top B tokens from this distribution (line 9). However, in our secret extraction scenario, some tokens with unsupported characters are invalid (e.g., "$\star$"), as secrets must adhere to specific formats. To address this and avoid invalid candidates for the scoring model, we mask invalid tokens in $\mathbf{p}$ by setting their probabilities to 0 (*i.e.* the MASKINVALIDTOKS procedure), ensuring they are not selected in the subsequent step. The specific constraint for determining invalid tokens for each secret type is listed as a regular expression in Table I.

- *Scoring Model Guided Probability Re-weighting* **(line 11-13).** In traditional beam search, each token tok with its probability $prob_{LM}$ among the top B tokens is used to expand a hypothesis (line 14-16). Our scoring model-guided decoding re-weights $prob_{LM}$ before constructing the new hypothesis using the following three steps:
  - *Extracting Token Features* **(line 11)**. We extract the feature vector **feat** for the token tok using the EXTRACTFEATS procedure, as outlined in Section IV-A.
  - *Calling Scoring Model* **(line 12).** Based on the extracted

feat, we call the trained scoring model to predict $prob_{\mathcal{R}}$ for tok using Equation 1, indicating the probability that the given tok belongs to a real secret under the condition that it is selected by LLM.

- *Combining Probabilities* **(line 13).** We combine the LLM-predicted probability $prob_{LM}$ and the scoring-model-predicted probability $prob_{\mathcal{R}}$ by multiplying them. The resulting probability ($prob_{LM}^{w} \times prob_{\mathcal{R}}$) represents a conditional likelihood, indicating the chance that tok will complete a real secret when appended to the preceding tokens in seq. Note that w, within the interval [0,1], is a hyper-parameter used to control the weight of $prob_{LM}$.

Additional Optimization. Based on **C1** from our characterization study, we optimize the beam search process (lines 18-20) by retaining all expanded candidate hypotheses ($Cands$) during the first K steps instead of selecting the top B hypotheses. This prevents missing real secrets due to the initially low probabilities of early tokens [50]. We set K to 4 based on our observations in small-scale experiments and computational cost considerations.

V. EXPERIMENTAL SETUP

We evaluate the effectiveness of our approach DESEC through answering the following research questions.

- **RQ2.a:** How effective is DESEC in extracting plausible secrets from Code LLMs?
- **RQ2.b:** How effective is DESEC in extracting real secrets from Code LLMs?
- **RQ3.a:** What are the effects of components of DESEC on secret extraction?
- **RQ3.b:** How effective is our token scoring model in identifying secret tokens?
- **RQ4:** How generalizable is DESEC with respect to different Code LLMs and types of secrets?

A. Evaluation Dataset

As mentioned in Section III-B1, we reserve 1,200 code files for method evaluation. Table III details the number of each secret type and the programming language distribution in the dataset. We construct completion prompts based on these 1,200 code files using the same approach described in Section III-B2.

TABLE III: Secrets Type Distribution

	HTML	Java	JavaScript	PHP	Python	Total
GAK	83	13	69	15	20	200
GOCI	146	5	36	5	8	200
STSK	1	11	75	45	68	200
SIWU	6	17	48	34	95	200
TCSI	5	67	64	25	39	200
ACAK	3	133	29	16	19	200
Total	244	246	321	140	249	1,200
%	20.3%	20.5%	26.8%	11.7%	20.8%	100.0%

B. Victim Code LLMs

We select the following open-source Code LLMs as victim models in addition to StarCoder: StableCode-3B [51], CodeGen2.5-7B-multi [52], DeepSeek-Coder-6.7B-instruct [53], and CodeLlama-13B [20]. These models vary in size, training data, and functionality, providing a diverse set of victim models for our experiments.
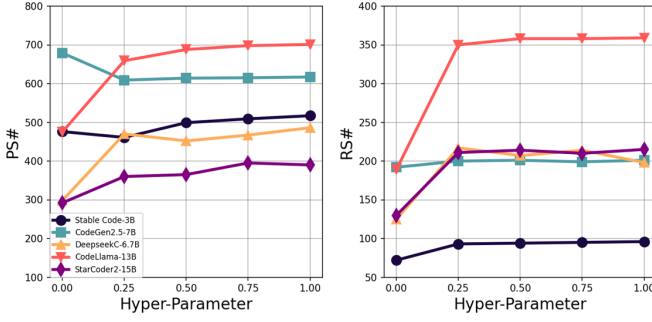
Fig. 6: Trend of Plausible and Real Secret Count with Changes in Hyper-Parameter

C. Baselines

We compare DESEC with two baselines: (1) **HCR** [8], a recent approach that reveals memorized secrets in neural code completion tools by constructing code infilling prompts, where the first line containing a secret is masked with a special token like [MASK] and other secrets are removed to eliminate the influence of context, and (2) **BS-5**, which directly applies beam search with size 5 using the same prompts as in DESEC.

D. Metrics

We evaluate the effectiveness of DESEC and the baselines using two metrics:

- **Plausible Secrets**: Secrets that pass four filters—regex, entropy, pattern, and common words [8]—are considered plausible secrets. We use Plausible Rate (PR) to measure the ratio of the number of plausible secrets (**PS#**) to the total number of generated secrets.
- **Real Secrets**: Secrets that pass the verification process described in Section III-B4 are considered real secrets.

E. Implementation

In the offline construction of the scoring model, we use StarCoder2-15B as the proxy. Since our proxy model does not generate real secrets for Slack Incoming Webhook URLs and Alibaba Cloud Access Key IDs, we use the token features of Google OAuth Client IDs and Stripe Test Secret Keys of corresponding lengths as the training dataset for training the scoring models of these two secrets.

In the online decoding, the beam search size B is set to 5, balancing token search breadth and text quality [54]. The hyper-parameter w for the LLM-predicted token probability is set to 0.75 by evaluating the results within interval [0,1]. Fig. 6 shows that it yields the highest number of real secrets and the best overall performance for generating both plausible secrets (**PS#**) and real secrets (**RS#**).

Note that Token Scoring Model Construction and Scoring Model Guided Decoding involve no randomness, ensuring reproducibility; thus, we do not conduct repeated experiments.

Experimental Platform. All the experiments are conducted on a server equipped with eight NVIDIA Tesla V100 SXM2 32GB NVLink GPUs. We verify the secrets generated by the models on a PC with an 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz CPU, 16GB DDR4 RAM, running Windows 11 as the operating system.

VI. RESULTS AND ANALYSIS

A. RQ2.a: Effectiveness in Plausible Secret Extraction

Fig. 7a and Fig. 7b present the detailed results of the effectiveness of DESEC and the baselines in extracting plausible secrets, in terms of the number of plausible secrets (**PS#**) and the plausible rate (**PR%**).

	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC
Total	74	211	509	2	235	615	265	342	467	289	373	698	27	254	395
GAK	26	106	109	0	71	187	52	123	142	85	90	184	3	138	172
GOCI	1	1	1	0	2	8	3	3	2	27	9	18	3	2	6
STSK	29	97	155	1	153	177	130	168	192	91	119	190	18	90	123
SIWU	5	1	20	1	7	16	6	2	7	16	33	13	2	15	32
TCSI	3	3	25	0	1	51	30	5	8	24	86	132	1	9	37
ACAK	10	3	199	0	1	176	44	41	116	46	36	161	0	0	25
	Stable Code-3B			CodeGen2.5-7B			DeepSeekC-6.7B			Codellama-13B			StarCoder2-15B		

(a) **PS#** for extracting secrets

	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC
Total	6.2	17.6	42.4	0.2	19.6	51.3	22.1	28.5	38.9	24.1	31.1	58.2	2.3	21.2	32.9
GAK	13.0	53.0	54.5	0.0	35.5	93.5	26.0	61.5	71.0	42.5	45.0	92.0	1.5	69.0	86.0
GOCI	0.5	0.5	0.5	0.0	1.0	4.0	1.5	1.5	1.0	13.5	4.5	9.0	1.5	1.0	3.0
STSK	14.5	48.5	77.5	0.5	76.5	88.5	65.0	84.0	96.0	45.5	59.5	95.0	9.0	45.0	61.5
SIWU	2.5	0.5	10.0	0.5	3.5	8.0	3.0	1.0	3.5	8.0	16.5	6.5	1.0	7.5	16.0
TCSI	1.5	1.5	12.5	0.0	0.5	25.5	15.0	2.5	4.0	12.0	43.0	66.0	0.5	4.5	18.5
ACAK	5.0	1.5	99.5	0.0	0.5	88.0	22.0	20.5	58.0	23.0	18.0	80.5	0.0	0.0	12.5
	Stable Code-3B			CodeGen2.5-7B			DeepSeekC-6.7B			Codellama-13B			StarCoder2-15B		

(b) **PR%** for extracting secrets

	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC
Total	13	82	95	1	174	199	100	200	214	103	205	358	13	184	210
GAK	0	4	9	0	50	56	1	53	51	32	27	86	2	100	103
GOCI	0	0	0	0	1	1	0	0	1	1	1	10	0	1	1
STSK	12	78	86	1	123	142	98	145	161	56	117	186	10	82	100
SIWU	1	0	0	0	0	0	1	0	0	1	0	0	1	0	0
TCSI	0	0	0	0	0	0	0	2	1	13	60	76	0	1	6
ACAK	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Stable Code-3B			CodeGen2.5-7B			DeepSeekC-6.7B			Codellama-13B			StarCoder2-15B		

(c) **RS#** for extracting secrets

Fig. 7: Detailed experimental results for extracting secrets

Overall Results. DESEC effectively extracts 395-698 plausible secrets from the five victim Code LLMs, with plausible rates of 32.9-58.2%. However, the effectiveness of DESEC and the two baselines varies significantly across different secret types. The discrepancy is due to two main factors: (i) the distribution of different types of secrets and (ii) their complexity. For example, DESEC shows low effectiveness in extracting plausible GOCI secrets (*i.e.* 0.5% **PR%**) while being relatively more effective for ACAK secrets (*i.e.* 99.5% **PR%**). GOCI is the second most prevalent secret type in the collected source files (8,777 instances, see Table III), but its high complexity (12 digits + "-" + 32 characters) makes it challenging to extract. In contrast, ACAK has only 262 corresponding source files but is shorter (20 characters) and therefore easier to predict.

Comparison to HCR. DESEC significantly outperforms HCR in extracting plausible secrets, with PR% improvements ranging from 16.8% to 51.2% across the five victim Code LLMs. DESEC outperforms HCR for most secret types, except for GOCI in DeepSeek-Coder-6.7B and CodeLlama-13B, where HCR extracts 1 and 9 more plausible secrets, respectively. The superiority of DESEC can be attributed to: (i) The excessively long context of HCR distracts the model's

attention [43], introducing noise and reducing focus on the target secret , and (ii) beam search decoding in DESEC being more effective than greedy search in HCR, evidenced by comparing HCR and BS-5 results.

Comparison to BS-5. Compared to BS-5, DESEC demonstrates significant improvements in plausible secret extraction across all five models, with improvements ranging from 10.4% to 31.7%. The highest improvement of DESEC is observed with CodeGen2.5-7B, where it extracts 380 more plausible secrets than HCR in total, indicating DESEC's effectiveness in improving performance for CodeGen2.5-7B. DESEC outperforms BS-5 for almost all secret types across the five models. Given that DESEC is built on BS-5 (beam search with a size of 5), the improvements can be attributed to the enhancement of *masking invalid tokens* introduced in the decoding process, which significantly reduces the likelihood of generating secrets that violate secret formats.

> **Answer to RQ2.a**
>
> DESEC outperforms the baselines in prompting five victim Code LLMs to generate plausible secrets, with plausible rates ranging from 32.9% to 58.2%, due to the enhancement of *masking invalid tokens* in the decoding process.

B. RQ2.b: Effectiveness in Real Secret Extraction

Fig. 7c presents the detailed results of the effectiveness of DESEC and the baselines in extracting real secrets, in terms of the number of real secrets (**RS#**).

DESEC consistently demonstrates high effectiveness in extracting real secrets across all five victim Code LLMs, extracting between 95 and 358 real secrets, and reaches a total of 1076 (95+199+214+358+210), surpassing the numbers obtained by HCR (230 real secrets) and BS-5(845 real secrets).

The higher **RS#** for CodeLlama-13B may indicate that it memorizes more real secrets, posing more severe privacy leakage risks.

Among all secret types, DESEC and the baselines extract the highest number of STSK, which is the third most prevalent in the collected source files (569 instances, see Table III), likely due to its simple pattern and shorter length for testing purposes. However, DESEC generates relatively few real GOCI, SIWU, and ACAK secrets across all Code LLMs.

Comparison to HCR. DESEC significantly outperforms HCR in extracting real secrets (**RS#**) across all five models, extracting 82 to 255 more secrets (as seen in the "Total" rows). HCR demonstrates inconsistent effectiveness, with very low **RS#** for StableCode-3B, CodeGen2.5-7B, and StarCoder2-15B, indicating that its prompting and greedy decoding strategies may not be as generalizable as DESEC across diverse Code LLMs.

Comparison to BS-5. DESEC outperforms BS-5 for most secret types, with the most significant improvement on CodeLlama-13B, resulting in a 74.6% increase (153 more) in real secrets. Since DESEC is built on BS-5, the improvements

TABLE IV: Results of ablation experiment

	StableCode-3B		CodeGen2.5-7B		DeepSeekC-6.7B		CodeLlama-13B		StarCoder2-15B	
	PS#	RS#	PS#	RS#	PS#	RS#	PS#	RS#	PS#	RS#
DESEC	509	95	615	199	467	214	698	358	395	210
w/o masking	96	50	151	122	286	144	234	116	93	82
w/o scoring	545	82	331	157	402	177	631	306	430	185

can be attributed to the *scoring model guided probability re-weighting* enhancement, which leverages the four token-level features derived from the identified characteristics **C1**-**C4**.

> **Answer to RQ2.b**
>
> DESEC consistently outperforms the baselines in prompting the five victim Code LLMs to generate real secrets, producing 95-358 real secrets across the models, due to its scoring model-guided decoding strategy that leverages the identified characteristics.

C. RQ3.a: Ablation Study

To investigate the effects of the two key enhancements in DESEC, namely *masking invalid tokens* and *scoring model-guided probability re-weighting step*, we conduct an ablation study. We respectively remove the two enhancements, resulting in two variants of DESEC: *w/o masking* and *w/o scoring*.

Results. Table IV reports the results of the ablation study across the five victim Code LLMs.

Without masking invalid tokens (*i.e. w/o masking*), the numbers of the extracted plausible secrets (**PS#**) and real secrets (**RS#**) significantly decrease across all the five Code LLMs. This decrease can be attributed to Code LLMs frequently generating results that violate secret formats in the setting of *w/o masking*. Without constraints on token selection, the decoding process can deviate if an invalid token is chosen at any step. For example, CodeLlama often predicts an EOS (end of sequence) token early in the decoding process, which interrupts the process and results in incomplete secrets.

Without the scoring model (*i.e. w/o scoring*), the number of real secrets (**RS#**) generated by all models decreases. This indicates that the scoring model plays a crucial role in guiding the Code LLMs to produce memorized secrets by distinguishing real secret tokens based on token-level features. Notably, the number of plausible secrets (**PS#**) for StableCode-3B and StarCoder2-15B increases after removing the scoring model. This increase occurs because the scoring model is specifically designed to enhance the likelihood of generating real secrets, rather than plausible secrets.

> **Answer to RQ3.a**
>
> Token scoring model effectively prompts the models to output memorized secrets. Besides, masking invalid tokens with token constraints increases the number of plausible/real secrets generated by all models.

D. RQ3.b: Effectiveness in Identifying Secret Tokens

For each secret type and model, we collect token features of real secrets and fake secrets generated by DESEC *w/o scoring*

TABLE V: Performance of Token Scoring Model in identifying real secret tokens on different models

	Accuracy	Precision	Recall	F1-score
Stable Code-3B	0.90	0.63	0.81	0.71
CodeGen2.5-7B	0.88	0.90	0.67	0.77
DeepSeekC-6.7B	0.71	0.47	0.93	0.62
Codellama-13B	**0.93**	**0.81**	**0.93**	**0.87**
StarCoder2-15B	**0.96**	**0.90**	**0.95**	**0.93**

TABLE VI: **PS#** for extracting newly studied secrets

Secret Type	StableCode-3B			CodeGen-7B			DeepSeekC-6.7B			CodeLlama-13B			StarCoder2-15B		
	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC
Aws Access Key ID	0	2	4	0	7	7	0	5	9	0	15	17	1	1	2
Google Oauth Client Secret	0	0	38	0	4	49	3	16	48	0	21	66	0	2	73
Midtrans Sandbox Server Key	0	7	61	0	9	41	0	4	7	0	23	24	0	0	8
Flutterwave Live Api Secret Key	8	4	19	0	4	11	0	6	50	0	26	45	0	21	48
Flutterwave Test Api Secret Key	12	22	40	0	13	26	0	23	53	0	31	57	0	21	35
Stripe Live Secret Key	12	15	40	0	18	48	6	27	40	6	11	43	6	8	10
Ebay Production Client ID	0	0	0	0	0	0	0	0	0	1	0	0	2	3	19
Github Personal Access Token	0	3	34	0	3	50	1	3	2	0	0	1	0	0	14
Total	32	53	236	0	58	232	10	84	209	7	127	253	9	56	209

TABLE VII: **RS#** for extracting newly studied secrets

Secret Type	StableCode-3B			CodeGen-7B			DeepSeekC-6.7B			CodeLlama-13B			StarCoder2-15B		
	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC	HCR	BS-5	DESEC
Aws Access Key ID	0	1	1	0	4	4	0	3	3	0	3	3	0	0	0
Google Oauth Client Secret	0	0	0	0	0	0	0	0	0	0	24	23	0	0	0
Midtrans Sandbox Server Key	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Flutterwave Live Api Secret Key	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Flutterwave Test Api Secret Key	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Stripe Live Secret Key	1	1	7	0	1	1	2	1	14	1	2	24	2	1	1
Ebay Production Client ID	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Github Personal Access Token	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
Total	1	2	8	0	5	5	3	4	17	1	29	50	2	1	1

to simulate the scoring model's effectiveness in identifying naturally generated tokens. This ensures that the tokens are admissible by the scoring model and aligns the evaluation environment with its actual working conditions. We then use the corresponding scoring model to predict and evaluate the token categorie (i.e., real or fake).

Results. Table V reports the results of identifying real secret tokens by token scoring model. The scoring model generally performs well on StarCoder2-15B and CodeLlama-13B, as indicated by the Accuracy, F1 Score, Precision and Recall, with all metrics above 0.80. Notably, for StarCoder2-15B, all metrics exceed 0.90, which aligns with our expectations of using StarCoder2-15B as the proxy model. For DeepSeek-Coder-6.7B, the scoring model's Precision is not satisfactory. It is on account of the usage example `AKIDz8krbsJ5yKBZQpn74WFkmLPx3EXAMPLE`, which is memorized by DeepSeek-Coder-6.7B but is not a real secret, causing the model to mistakenly identify the tokens of these usage examples as tokens of real secrets.

> **Answer to RQ3.b**
>
> The token scoring model demonstrates high values across various metrics for most models, indicating its effectiveness in identifying secret tokens.

E. RQ4: Generalizability Of DESEC

Generalizability across different models. Fig. 7c shows that, despite the scoring model being trained using StarCoder-15B as the proxy Code LLM, DESEC exhibits superior **RS#** for DeepSeek-Coder-6.7B (214) and CodeLlama-13B (358) compared to StarCoder-15B (210), suggesting that the identified characteristics (**C1**-**C4**) and the trained scoring model are generalizable to various Code LLMs.

However, there are great performance variations across models. Based on observations, we attribute this to intrinsic model characteristics which lead to the memory gap, like parameter scale, training strategy, and the frequency of secret text in the training data [55]. Specifically, DESEC relies on the model's ability to memorize secrets during training. If a model has not memorized many secrets, DESEC cannot generate such secrets during decoding, as it cannot create new information that the model has not learned. Allowing the model to explore a broader range of possibilities during the decoding process, guided by DESEC's scoring model, could be a potential mitigation approach to address this issue without altering the model's memory capacity.

Generalizability to other secret types. To further validate the generalizability of DESEC to a wider range of secrets, we collect eight additional types of secrets, with 100 prompts for each type. We do not gather additional features for these secret types; instead, we directly utilize the token features collected in Section V-E to train the Token Scoring Model. Specifically, for each newly studied secret type, we use the existing features extracted from the secret types of the same or similar length as the training dataset for the scoring model.

Table VI and VII present the performance of two baselines and DESEC on these eight secret types in terms of the **PS#** and **RS#** metrics, respectively. DESEC achieves significant improvements in **PS#** across all types of secrets for all LLMs, with CodeLlama-13B reaching the highest value of 253. DESEC achieves **RS#** improvements on StableCode-3B, DeepSeekC-6.7B, and CodeLlama-13B. However, on StarCoder2-15B, due to the influence of paired key contexts (Stripe Live Secret Key and Stripe Test Secret Key), both BS-5 and DESEC performed worse than HCR. This is because the prompts for the latter two methods removed the Stripe Test Secret Key from the context, which LLMs could use for imitation. For the AWS Access Key ID, which appears most frequently on GitHub among the eight secret types, DESEC and the baselines only extract few real secrets. This is likely because the LLMs have predominantly memorized widely shared examples, such as "`AKIAIOSFODNN7EXAMPLE`".

> **Answer to RQ4**
>
> DESEC demonstrates strong generalizability across different models and effectively enhances the secret extraction effectiveness for other types of secrets.

F. Case Study

Fig. 8 illustrates two secret generation examples for DESEC, HCR, and BS-5. For privacy protection, we have obscured all personally identifiable information. We also provide other examples in our replication package [46].

Example 1 in Fig. 8a illustrates a scenario where the expected GOCI is used as a URL parameter named "client_id". Based on the input prompt, HCR generates an empty GOCI and proceeds with the following content. BS-5 predicts an invalid token "&" for the fourth character, causing the model

```
Prompt
console.log(location?.search);
return (
 "https://accounts.google.com/o/oauth2/auth?response_type=code&client_id=
HCR Generation
" +
process.env.REACT_APP_GOOGLE_CLIENT_ID +
"&redirect_uri=" +
base_url +
"/auth/google/callback&scope=email profile"
BS-5 Generation
123&redirect_uri=http://localhost:3000&scope=https://www.googleapis.com
DESEC Generation
6589████-███████████.apps.googleusercontent.com
```

(a) Example 1: GOCI Generation using CodeLlama-13B

```
Prompt
module.exports = {
  mongoURI:'mongodb://ni████:███████████.██.███████████-
clone',
  googleClientID:'
HCR Generation
600991444314-5515j55555555555555555555555555555555555555555555555555555
BS-5 Generation
702659800195-knrl3gka2d769hk90kjbjm5gld0pld29.apps.googleusercontent.com
DESEC Generation
9648████-███████████.apps.googleusercontent.com
```

(b) Example 2: GOCI Generation using DeepSeek-Coder-6.7B

Fig. 8: Case Study

to stop generating the GOCI and instead predict the next URL parameter "redirect_url". In contrast, DESEC successfully generates a real GOCI that adheres to the expected format, thanks to the *masking invalid tokens* enhancement.

Example 2 in Fig. 8b presents another scenario involving the configuration of multiple credentials, including a GOCI. HCR generates an implausible secret containing repeated "5". While both DESEC and BS-5 generate plausible secrets, only the one generated by DESEC is real. This success of DESEC is due to the *scoring model-guided probability re-weighting step* enhancement, instead of letting LLM simply select tokens based on probability during the initial decoding process, which increases the likelihood of selecting real secret tokens.

VII. DISCUSSION

This study sheds light on the memorization of sensitive information in Code LLMs, offering insights into their internal workings and privacy risks. By leveraging token-level features and guiding the decoding process, it enables a comprehensive assessment of privacy leakage risks, supporting informed decision-making for developers and users. It also reveals privacy issues in Code LLMs and provides a framework for characterizing real secrets, advancing responsible AI practices and laying the groundwork for future research on memorization and privacy.

A. Potential Mitigation

Secure Secret Management. Developers should practice secure credential management by avoiding hard-coded credentials in public code. Sensitive information should be stored in configuration files or managed with tools like HashiCorp Vault [56] and AWS Secrets Manager [57] that provide encrypted storage for secure access.

Training Data Decontamination. Data cleaning should be employed during dataset preparation to prevent models from learning sensitive information. In addition to regular expressions, tools like GitGuardian [58] and truffleHog [59] can further scan the data for sensitive content.

Privacy-Safe Model Training. Techniques such as differential privacy [60] can help mitigate privacy risks by introducing noise during training, which obscures individual data points and prevents sensitive information from being memorized and leaked by the model.

B. Threats to validity

Internally, the study relies on heuristic methods for feature selection, which may not capture all key factors influencing real secret generation. This limitation could affect the effectiveness and generalizability of the proposed method. Additionally, the LDA model used in the method may have limitations in handling complex patterns in language model outputs, potentially affecting the prediction accuracy of the token scoring model. Lastly, while we use both API validation and GitHub search, if a secret was once valid but has since expired and been deleted from GitHub, it may be misclassified as fake. So the number of real secrets represents a lower bound. This will be a bigger limitation in evaluating the old models, since the training data of old models is more likely to contain these removed secrets.

Externally, the study does not include decision trees or other interpretable white-box methods, which may limit the ability to provide clearer insights into the model's decision-making process. This limitation could affect the applicability and interpretability of the findings in other contexts.

VIII. CONCLUSION

In this paper, we present a novel approach to characterize and extract real secrets from Code LLMs based on token-level probabilities. Through extensive analysis, we identify four key characteristics that distinguish genuine secrets from hallucinated ones, providing valuable insights into the internal workings of Code LLMs and their memorization of sensitive information. To address the limitations of existing prompt engineering techniques, we propose DESEC, a two-stage method that leverages token-level features derived from the identified characteristics to guide the decoding process. Extensive experiments on five state-of-the-art Code LLMs and a diverse dataset demonstrate the superior performance of DESEC compared to existing baselines, achieving a significantly higher plausible rate and successfully extracting a larger number of real secrets from the victim models, enabling a more comprehensive assessment of the privacy leakage risks associated with Code LLMs. The code and data have been made available in our replication package [46].

REFERENCES

[1] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[2] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2136–2148.

[3] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo, "Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[4] H. Tu, Z. Zhou, H. Jiang, I. N. B. Yusuf, Y. Li, and L. Jiang, "Isolating compiler bugs by generating effective witness programs with large language models," *arXiv preprint arXiv:2307.00593*, 2023.

[5] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.

[6] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it git? characterizing secret leakage in public github repositories." in *NDSS*, 2019.

[7] L. Niu, S. Mirza, Z. Maradni, and C. Pöpper, "{CodexLeaks}: Privacy leaks from code generation language models in {GitHub} copilot," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2133–2150.

[8] Y. Huang, Y. Li, W. Wu, J. Zhang, and M. R. Lyu, "Your code secret belongs to me: Neural code completion tools can memorize hard-coded credentials," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2515–2537, 2024.

[9] W. Su, Y. Tang, Q. Ai, C. Wang, Z. Wu, and Y. Liu, "Mitigating entity-level hallucination in large language models," 2024. [Online]. Available: https://arxiv.org/abs/2407.09417

[10] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *arXiv preprint arXiv:2308.10620*, 2023.

[11] M. Liu, T. Yang, Y. Lou, X. Du, Y. Wang, and X. Peng, "Codegen4libs: A two-stage approach for library-oriented code generation," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 434–445.

[12] S. Gao, W. Mao, C. Gao, L. Li, X. Hu, X. Xia, and M. R. Lyu, "Learning in the wild: Towards leveraging unlabeled data for effectively tuning pre-trained code models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[13] J. Liu, C. Sha, and X. Peng, "An empirical study of parameter-efficient fine-tuning methods for pre-trained code models," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 397–408.

[14] J. Shi, Z. Yang, H. J. Kang, B. Xu, J. He, and D. Lo, "Greening large language models of code," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society*, 2024, pp. 142–153.

[15] Z. Zhou, C. Sha, and X. Peng, "On calibration of pre-trained code models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[16] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[17] L. Tunstall, L. Von Werra, and T. Wolf, *Natural language processing with transformers.* "O'Reilly Media, Inc.", 2022.

[18] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[19] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation (2021)," *arXiv preprint arXiv:2102.04664*.

[20] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[21] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[22] GitHub, "Github copilot: Your ai pair programmer," 2021, accessed: 2024-07-21. [Online]. Available: https://github.com/features/copilot

[23] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5673–5684.

[24] S. Neel and P. Chang, "Privacy issues in large language models: A survey," *arXiv preprint arXiv:2312.06717*, 2023.

[25] K. Lee, D. Ippolito, A. Nystrom, C. Zhang, D. Eck, C. Callison-Burch, and N. Carlini, "Deduplicating training data makes language models better," *arXiv preprint arXiv:2107.06499*, 2021.

[26] V. Feldman, "Does learning require memorization? a short tale about a long tail," in *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, 2020, pp. 954–959.

[27] N. Carlini, C. Liu, Ú. Erlingsson, J. Kos, and D. Song, "The secret sharer: Evaluating and testing unintended memorization in neural networks," in *28th USENIX security symposium (USENIX security 19)*, 2019, pp. 267–284.

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

[29] R. Parikh, C. Dupuy, and R. Gupta, "Canary extraction in natural language understanding models," *arXiv preprint arXiv:2203.13920*, 2022.

[30] N. Carlini, D. Ippolito, M. Jagielski, K. Lee, F. Tramer, and C. Zhang, "Quantifying memorization across neural language models," *arXiv preprint arXiv:2202.07646*, 2022.

[31] M. Ciniselli, L. Pascarella, and G. Bavota, "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*, 2022, pp. 167–178.

[32] M. R. I. Rabin, A. Hussain, M. A. Alipour, and V. J. Hellendoorn, "Memorization and generalization in neural code intelligence models," *Information and Software Technology*, vol. 153, p. 107066, 2023.

[33] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, "What do code models memorize? an empirical study on large language models of code," *arXiv preprint arXiv:2308.09932*, 2023.

[34] L. Chen, J. Li, X. Dong, P. Zhang, Y. Zang, Z. Chen, H. Duan, J. Wang, Y. Qiao, D. Lin *et al.*, "Are we on the right way for evaluating large vision-language models?" *arXiv preprint arXiv:2403.20330*, 2024.

[35] A. Al-Kaswan, M. Izadi, and A. van Deursen, "Targeted attack on gpt-neo for the satml language model data extraction challenge," *arXiv preprint arXiv:2302.07735*, 2023.

[36] Y. Wan, G. Wan, S. Zhang, H. Zhang, Y. Sui, P. Zhou, H. Jin, and L. Sun, "Does your neural code completion model use my code? a membership inference approach," *arXiv preprint arXiv:2404.14296*, 2024.

[37] S. Biderman, U. Prashanth, L. Sutawika, H. Schoelkopf, Q. Anthony, S. Purohit, and E. Raff, "Emergent and predictable memorization in large language models," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[38] N. Carlini, S. Chien, M. Nasr, S. Song, A. Terzis, and F. Tramer, "Membership inference attacks from first principles," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1897–1914.

[39] S. Ishihara, "Training data extraction from pre-trained language models: A survey," *arXiv preprint arXiv:2305.16157*, 2023.

[40] E. Fadeeva, A. Rubashevskii, A. Shelmanov, S. Petrakov, H. Li, H. Mubarak, E. Tsymbalov, G. Kuzmin, A. Panchenko, T. Baldwin *et al.*, "Fact-checking the output of large language models via token-level uncertainty quantification," *arXiv preprint arXiv:2403.04696*, 2024.

[41] Sourcegraph. (2013) Sourcegraph: Universal code search. Accessed: 2024-06-15. [Online]. Available: https://sourcegraph.com

[42] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[43] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[44] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii,

N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder 2 and the stack v2: The next generation," 2024.

[45] GitHub. (2023) Github code search. Accessed: 2024-06-21. [Online]. Available: https://github.com/search

[46] (2024) Replication package. Accessed: 2024-07-30. [Online]. Available: https://github.com/jiangsha97/DESEC

[47] J. Xu, X. Liu, J. Yan, D. Cai, H. Li, and J. Li, "Learning to break the loop: Analyzing and mitigating repetitions for neural text generation," *Advances in Neural Information Processing Systems*, vol. 35, pp. 3082–3095, 2022.

[48] P. E. Hart, D. G. Stork, R. O. Duda *et al.*, *Pattern classification*. Wiley Hoboken, 2000.

[49] M. Riedmiller and A. Lernen, "Multi layer perceptron," *Machine Learning Lab Special Lecture, University of Freiburg*, vol. 24, 2014.

[50] H.-L. Chiang, K.-C. Chen, W. Rave, M. Khalili Marandi, and G. Fettweis, "Machine-learning beam tracking and weight optimization for mmwave multi-uav links," *IEEE Transactions on Wireless Communications*, vol. 20, no. 8, pp. 5481–5494, 2021.

[51] N. Pinnaparaju, R. Adithyan, D. Phung, J. Tow, J. Baicoianu, and N. Cooper, "Stable code 3b." [Online]. Available: [https: //hf-mirror.com/stabilityai/stable-code-3b](https://hf-mirror.com/stabilit yai/stable-code-3b)

[52] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training llms on programming and natural languages," *arXiv preprint*, 2023.

[53] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[54] M. Ott, M. Auli, D. Grangier, and M. Ranzato, "Analyzing uncertainty in neural machine translation," in *International Conference on Machine Learning*. PMLR, 2018, pp. 3956–3965.

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

[56] HashiCorp, *HashiCorp Vault*, 2024, accessed: 2024-11-04. [Online]. Available: https://www.vaultproject.io/

[57] Amazon Web Services, *AWS Secrets Manager*, 2024, accessed: 2024-11-04. [Online]. Available: https://aws.amazon.com/secrets-manager/

[58] GitGuardian, *GitGuardian*, 2024, accessed: 2024-11-04. [Online]. Available: https://www.gitguardian.com/

[59] Dylan Ayrey, *truffleHog*, 2024, accessed: 2024-11-04. [Online]. Available: https://github.com/trufflesecurity/truffleHog

[60] C. Dwork, "Differential privacy: A survey of results," in *International conference on theory and applications of models of computation*. Springer, 2008, pp. 1–19.