CODECIPHER: LEARNING TO OBFUSCATE SOURCE CODE AGAINST LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

While code large language models have made significant strides in AI-assisted coding tasks, there are growing concerns about privacy challenges. The user code is transparent to the cloud LLM service provider, inducing risks of unauthorized training, reading, and execution of sensitive code. Such fear of data leaking prevents developers from submitting their code to LLMs. In this paper, we propose CodeCipher, a novel method that perturbs privacy from code while preserving the original response from LLMs. CodeCipher transforms the LLM's embedding matrix so that each row corresponds to a different word in the original matrix, forming a token-to-token confusion mapping for obfuscating source code. The new embedding matrix is optimized through minimizing the task-specific loss function. To tackle the challenge from the discrete and sparse nature of word vector spaces, CodeCipher adopts a discrete optimization strategy that aligns the updated vector to the nearest valid token in the vocabulary before each gradient update. We demonstrate the effectiveness of our approach on three AI-assisted coding tasks including code completion, summarization, and translation. Results show that our model successfully confuses the source code while preserving the original LLM's performance.[1]

1 INTRODUCTION

The rise of code large language models (code LLMs), including CodeLlama (Rozière et al., 2023) and DeepSeek-Coder (Guo et al., 2024), has ushered in a new phase for the research on code intelligence. These code language models have made significant strides in improving the quality of code generation and have achieved high pass rates across multiple benchmark datasets such as HumanEval (Chen et al., 2021), and MBPP (Austin et al., 2021). With the increasing reliance on these models for AI-assisted coding tasks, there is a growing need to leverage remote LLM cloud services, which enable broader scalability and enhance the capabilities of these models by offloading computations to powerful cloud infrastructure. While large white-box LLMs are readily available for local deployment and execution, limited computational resources may prevent individual users from deploying them locally, necessitating a client-server architecture.

LLM cloud services allow users to perform AI coding tasks without the need for computational infrastructure. However, providing code LLMs as services raises significant privacy concerns, which limit their practical applications (Yao et al., 2024; Yan et al., 2024). As users' code is transparent to these remote systems, there is a risk of exposing and misusing sensitive or proprietary information. Such fear of data leaking prevents many developers from submitting their code directly to LLMs. Therefore, privacy-preserving mechanisms in LLM deployment are required to ensure user trust and further expand code LLM usage.

Code obfuscation techniques have been common solutions to defend code against unauthorized access, which disguises elements of a piece of code while maintaining its output. Existing techniques adopt rule-

[1]Code and data available at `https://anonymous.4open.science/r/CodeCipher_final-9D7E/`

based perturbations, such as renaming variables with semantically meaningless symbols (Jain et al., 2020) and injecting dead code (Chakraborty et al., 2022), to obscure human comprehension of code semantics. However, since LLMs also rely on meaningful symbols and other human-readable code structures (Ding et al., 2024; Rodeghero et al., 2014). Brute code obfuscation could significantly degrade the capability of LLMs and thus are inapplicable in practice.

In this work, we propose CodeCipher, a novel learning-based code obfuscation technique tailored for LLMs. CodeCipher safeguards code from unauthorized training, reading, compiling, and execution, without sacrificing LLM service quality. The core idea behind CodeCipher is to transform the LLM's embedding matrix so that each row corresponds to a different word in the original matrix. This process creates a token-to-token confusion mapping, which the system uses to obfuscate tokens when encountering new code snippets.

The new embedding matrix is optimized by minimizing the task-specific loss. Due to the discrete and sparse nature of word embedding space, straightforward gradient descent can not pinpoint the embedding of a valid token. To tackle this problem, we adopt a discrete optimization strategy: at each iteration, the algorithm updates the perturbation vector, projects it onto the nearest valid token in the embedding space, and recalculates gradients based on the new projection.

The efficacy of our approach was rigorously assessed across three AI-assisted coding tasks: code completion, code summarization, and code translation. Results revealed that our method surpassed a range of conventional obfuscation methods in terms of both the level of confusion and the preservation of performance for downstream tasks. These findings affirm the practicability of our approach for enhancing privacy and security in LLMs.

2 Related Work

Privacy Protection for Large Language Models While LLMs have brought tremendous value to many fields, they have also sparked deep concerns among users regarding data privacy and security (Nasr et al., 2023). Numerous studies have been conducted on model security in previous research (Fan et al., 2023; Yan et al., 2024; Lin et al., 2024). On the model side, federated learning and differential privacy are commonly used to protect privacy leakage during model training. Homomorphic encryption is widely used during model inference (Lin et al., 2024). On the user side, privacy protection focuses on marking or transforming user's data. Watermarking, the most common technology, embeds hidden markers in data to detect its use in model training (Sun et al., 2023). Another method is data transformation, which makes data unintelligible or unusable even if leaked. Traditionally, this involves rule-based removal of sensitive information (Machanavajjhala et al., 2007). More recently, LLMs have been leveraged to obscure sensitive data (Song et al., 2024). Users can construct privacy-preserving prompts, mixing real and fake inputs to prevent the server from identifying the true prompt (Utpala et al., 2023; Zhou et al., 2023).

While the aforementioned methods have been widely used for privacy protection, they differ from CodeCipher significantly in technical scope. One key difference is that they often necessitate complex modifications to both client- and server-side model architectures and training methodologies. For instance, homomorphic encryption requires substantial changes to the self-attention architecture in LLMs. In contrast, CodeCipher offers a lightweight, practical solution that only involves minimal data transformations on the client side without altering the model itself or requiring server modifications. This simplicity makes CodeCipher a more accessible option for code obfuscation.

Privacy Protection for Code LLMs Data leakage is also a concern in LLMs for code, which are often trained on open-source code repositories that may contain sensitive information. Studies (Feng et al., 2022; Huang et al., 2023; Al-Kaswan et al., 2023) have shown that specific prompts can extract private data like passwords and server keys from large models. Numerous solutions have been proposed to address this issue, such as adding watermarks to detect unauthorized use of training data (Sun et al., 2023), training models

on desensitized data (though its impact on performance is still unknown) (Yang et al., 2024), and distilling models to smaller, more secure local models (Shi et al., 2024). However, ensuring the safe and private use of LLMs, especially for code generation, remains a critical challenge that requires further research.

Code Obfuscation Code obfuscation aims to reduce the readability of source code through code transformation, reorganization, and modification (Behera & Bhaskari, 2015). Commonly used code obfuscation techniques include identifiers renaming (Eastridge-Technology, 2000; Hoenicke, 2001), control flow flattening (László & Kiss, 2009), dead code injection (Chakraborty et al., 2022), self-modifying code (Giffin et al., 2005), and property encryption (Pandey & Rouselakis, 2012).

While code obfuscation has been widely used for code security, its applicability to LLMs remains largely unexplored. Traditional rule-based methods achieve the goal of obfuscation, but they are not specifically tailored to LLMs. As a consequence, the code representations learned by LLMs can be altered, leading to significant performance degradation.

3 Problem Statement

Given a transformer-based code language model $P_\Phi(y|x)$ parameterized by Φ and a downstream task (e.g., code summarization) represented by a parallel dataset: $\mathcal{T} = \{(x^{(i)}, y^{(i)})\}\ i = 1, .., N$, where x and y are input and target sequences of tokens. For each input source code $x = \{w_1, ..., w_n\}$, our goal is to transform it into obfuscated code $x' = \{w'_1, ..., w'_n\}$ which differs from x in lexicons while retaining task-specific performance (e.g., code completion) when processed by an LLM, formally:

$$\min_{x'=g(x)} ||s(P_\Phi(x'), y) - s(P_\Phi(x), y)|| \tag{1}$$

where $g : \mathcal{V} \to \mathcal{V}$ denotes a confusion function that maps any token in the vocabulary into an obfuscated one; $s : Y \times Y \to R$ denotes a task-specific scoring function such as Pass@K in the code completion task.

4 CodeCipher: Learning to Obfuscate Source Code

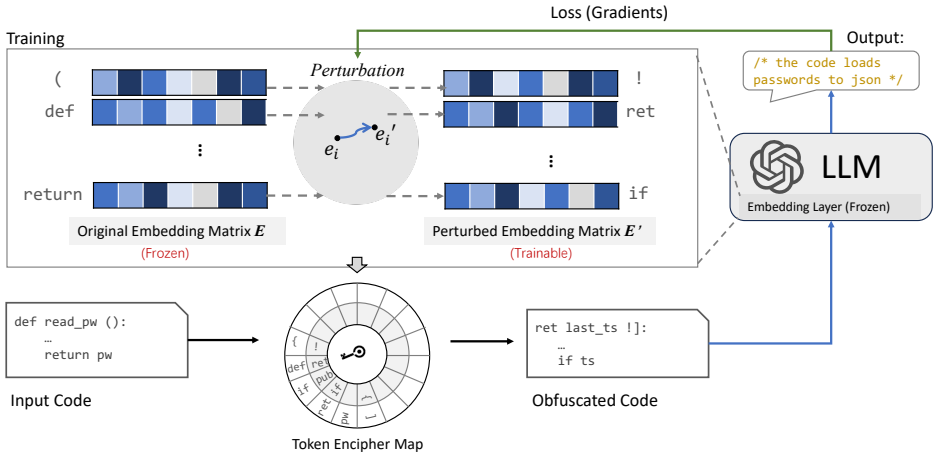


Figure 1: The overall framework of CodeCipher. It freezes the entire LLM and merely optimizes a perturbed embedding matrix $\mathbf{E}'$ for the embedding layer.

In this paper, we propose a learning-based method to find the optimal confuse function $g(\cdot)$ between code tokens. Unlike previous rule-based methods, we move our focus to the embeddings of code tokens. The core idea behind CodeCipher is to transform the LLM's embedding matrix so that each row corresponds to a different word in the original matrix. The transformation establishes a token-to-token confusion mapping, which the system uses to obfuscate tokens when encountering new code snippets. Our method consists of three stages as illustrated in Figure 1: (i) transforms the LLM's embedding matrix to a permutation, establishing a token-to-token confusion mapping; (ii) optimizes the transformation of embedding by minimizing the task-specific loss function; addresses the intricacies involved in the discretization process, particularly the alignment of continuous word vectors to valid tokens; and (iii) obfuscates code using the trained model. A more comprehensive description of the algorithm can be found in Appendix A.

4.1 Embedding Permutation

For any code snippet $x = (w_1, ..., w_n)$, the embedding layer of an LLM converts the tokens into a sequence of continuous vectors:

$$\mathbf{e}_1, ..., \mathbf{e}_T = \mathbf{E}(w_1), ..., \mathbf{E}(w_T) \tag{2}$$

where $\mathbf{E} \in R^{|\mathcal{V}| \times d}$ represents the embedding matrix in the LLM's embedding layer, $|\mathcal{V}|$ denotes the vocabulary size, and d represents the dimension of word embeddings.

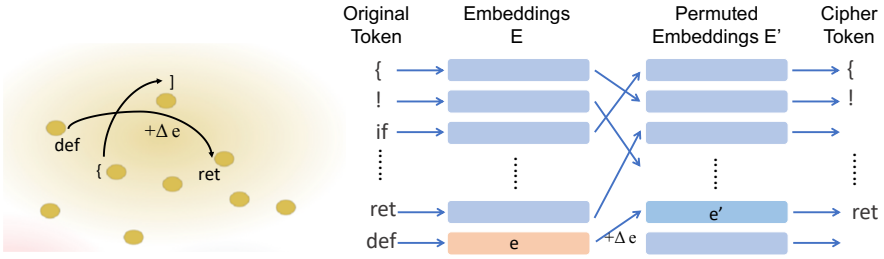


Figure 2: Illustration of Embedding Perturbation. The occurrence of token `def` can be reasonably substituted with `ret` through embedding transition, thereby obfuscating source code with these tokens.

To perturb the code tokens, we create a learnable confusion mapping within their word embedding space, as illustrated in Figure 2. Specifically, given the original embedding matrix $\mathbf{E}$, we introduce a trainable embedding matrix $\mathbf{E}' \in R^{|\mathcal{V}| \times d}$. The matrix $\mathbf{E}'$ is a permutation of $\mathbf{E}$, meaning that $\forall \mathbf{e}' \in \mathbf{E}'$, $\mathbf{e}' \in \mathbf{E}$. This ensures that every embedding in $\mathbf{E}'$ can be mapped back to its corresponding token in the vocabulary. $\mathbf{E}$ and $\mathbf{E}'$ establishes a confusion mapping: each row $\mathbf{e}_i \in \mathbf{E}'$ represents the perturbed word vector corresponding to the original word vector in the same row of $\mathbf{E}$.

With the learnable confusion mapping, we design the obfuscation process as follows: for each token w in the input code, we obtain its original embedding $\mathbf{e} = \mathbf{E}(w)$ using the embedding layer of the LLM. Then, we transform the embedding to $\mathbf{e}' = \mathbf{e} + \Delta\mathbf{e}$, where $\Delta\mathbf{e}$ denotes the perturbation of $\mathbf{e}$ in the corresponding row in $\mathbf{E}$.

The perturbed vector $\mathbf{e}'$ is then decoded to the nearest token in the vocabulary through a lookup function $w' = \textsc{Dec}_{\mathcal{V}}(\mathbf{e}')$, designated as the perturbed token. With this perturbation process, the entire sequence can be obfuscated as:

$$x' = (w'_1, ..., w'_n) = \textsc{Dec}_{\mathcal{V}}(\mathbf{e}'_1, ..., \mathbf{e}'_n) \tag{3}$$

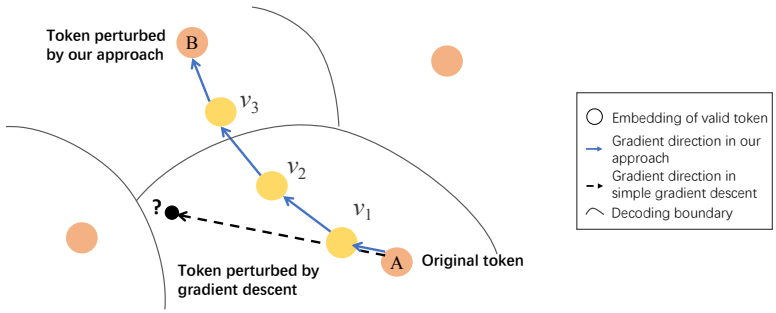


Figure 3: Illustration of discrete gradient search. Standard gradient descent perturbs token A along the gradient direction, often leading to suboptimal perturbations within the decoding boundary. Our discrete gradient search mitigates this issue by introducing mini-steps along valid tokens, ensuring more meaningful perturbations.

4.2 TRAINING

Our training objective is to find an optimal $\mathbf{E}'$ that minimizes the task-specific loss function $\mathcal{L}_{\text{task}}$ when applied to the obfuscated code. A straightforward idea is to optimize $\mathcal{L}_{\text{task}}$ using gradient descent. For a code input, we update the embedding $\mathbf{e}$ for each token as follows:

$$\mathbf{e}' = \mathbf{e} - \eta \nabla_{\mathbf{e}} \mathcal{L}_{task}(x') \tag{4}$$

where η represents the learning rate, $\nabla_{\mathbf{e}}\mathcal{L}$ denotes the gradient of the loss function w.r.t. the current embedding $\mathbf{e}$.

However, direct gradient descent encounters the intrinsic limitation in the word embedding space. Code tokens reside on a sparse, discrete manifold in the embedding space. As illustrated in Figure 7, a single gradient computation is hard to pinpoint the exact position of a valid token in the embedding space, thereby hindering precise vector manipulation.

Inspired from Yuan et al. (2021), we develop a discrete gradient search algorithm. During each gradient update, the algorithm aligns the updated vector with semantically meaningful tokens, ensuring transitions occur between valid tokens in the original embedding space. More specifically, the algorithm performs gradient update in Equation 4 along a finer-grained trajectory $v_1, ..., v_t$. At each mini step, the embedding vector is incrementally perturbed and then projected onto the nearest valid token vector in the vocabulary. The nearest valid token is identified by calculating the dot product with the normalized embeddings of existing words. The gradient is then recalculated based on this new token, guiding further refinements in the following iterations. Formally,

$$v_{t+1} = \Pi_{\mathcal{V}}(v_t - \eta \nabla_{v_t} \mathcal{L}_{\text{task}}(v_t)) \tag{5}$$

where v_t denotes the token vector at the t^{th} step; $\Pi_{\mathcal{V}}$ denotes the projection operation which aligns the updated vector onto the nearest word's vector within the vocabulary space $\mathcal{V}$; η represents the learning rate; $\nabla_{v_t}\mathcal{L}(v_t)$ denotes the gradient of the loss function $\mathcal{L}$ with respect to v_t.

This iterative process continues for a certain number of steps, at which point we select the vector with the lowest task loss and perform the final projection, designating it as the perturbed token.

To prevent over-obfuscation, we limit the training process to a maximum of N iterations. The training early stops if the perplexity deviation of a code sample reaches a predefined threshold δ. We define δ as a linear function that increases with each step i, namely, $\delta = \alpha \cdot i + \beta$, where β indicates the initial threshold and α controls the rate at which the threshold increases throughout the training process.

4.3 CODE OBFUSCATION

Once the new embedding matrix has been trained, it establishes a confusion function that maps each token in the vocabulary to its obfuscated counterpart. For any new input code snippet, CodeCipher converts the tokens by referencing this mapping table, producing an obfuscated version of the code. The obfuscated code can then be sent to a cloud LLM with the same specifications as the one used for training.

5 EXPERIMENTS

5.1 COMMON SETUP

As a proof of concept, we build and evaluate CodeCipher with CodeLlama-7B (Rozière et al., 2023), a representative white-box code LLM. We will show in Section 5.6 that the obfuscated code generated by CodeLlama-7B can be transferred to a wide range of model specifications including different sizes and black-box LLMs. More details about the hyperparameter setting can be found in Table 4 in the Appendix.

We train and evaluate our obfuscation model on three AI-assisted coding tasks.

Code Completion. A task that completes the subsequent code given users' partial code. We train our model on the MultiPL-E dataset (the Python subset from the Leetcode section) (Cassano et al., 2023), which comprises 446 code snippets. We test the model on the HumanEval benchmark (Chen et al., 2021) which contains 164 Python programming problems. For each problem, we obfuscate the first half of the code, provide it to the LLM, and ask it to complete the remaining half. As the obfuscated half code might prevent the final output from being compiled, we guide the model to re-generate the entire code from head using the prompt in Appendix B.

Code Summarization. A task that provides a succinct natural language summary for a given code fragment (Sridhara et al., 2010). We train and test our obfuscation model on the CodeSearchNet benchmark (Husain et al., 2019) using the prompt in Appendix B.

Code Translation. A task that translates source code written in one language to another (Pan et al., 2023). We train our model on the Java-to-Python subset of XLCoST (Zhu et al., 2022) and test it on the Java-to-Python subset of HumanEval-X (Zheng et al., 2023). We use google-java-format[2] to ensure consistent line breaks and indentation across the code, aligning it with the formatting of the test dataset. This process resulted in a total of 2,374 data pairs. In the prompt in Appendix B, we explicitly instructed the model to avoid class-based code structures.

Baselines. We compare CodeCipher with both rule-based and LLM-based code obfuscation approaches. 1) *Random perturbation*, which randomly replaces a portion of tokens in the code with non-semantic symbols. 2) *Identifier renaming*, which replaces variable and function names with non-semantic symbols (Eastridge-Technology, 2000; Hoenicke, 2001). It is implemented from Spoon (Pawlak et al., 2015). 3) *Dead branch injection*, which inserts code blocks that never execute (e.g., `while(false)`). It is implemented from NatGen (Chakraborty et al., 2022). 4) *Remove symbols*, which removes formatting and syntactic symbols such as whitespace and parentheses from the code. 5) *Prompting LLMs*, which prompts GPT-4o directly to obfuscate code using the prompt in Appendix B. 6) *Obfuscation + Inform*, which obfuscates identifiers in the source code and explicitly informs the LLM about this obfuscation in the prompt (e.g., "Generate a docstring for the code where the identifiers are obfuscated."). Previous work has shown that LLMs can better handle obfuscated code when they are made aware of the obfuscation upfront (Yan & Li, 2021).

[2] https://github.com/google/google-java-format

Table 1: Results of Performance Preservation on the Code Completion Task.

Method	Task-specific Performance			Obfucation Degree	
	Pass@1(%)	Pass@10(%)	Pass@100(%)	PPL	Edit distance(%)
Origin	50.60	68.68	79.27	3.56	0
Random perturb	40.24	61.34	75.00	5.95	3.86
Conventional Obfuscation					
Identifier renaming	42.68	61.15	75.00	5.64	8.74
Dead branch injection	39.63	60.23	71.95	4.71	9.21
Remove symbols	37.20	60.93	75.60	**6.09**	3.38
LLM-based Obfuscation					
Encipher with LLM prompting	39.02	60.33	72.56	4.57	8.94
Obfuscation + Inform	41.46	57.67	70.12	5.64	8.74
CodeCipher (ours)	**47.56**	**65.20**	**77.15**	**6.09**	**9.41**

5.2 Performance Preservation

We measure the efficacy of CodeCipher based on two competitive objectives: task performance (i.e., Pass@k (Chen et al., 2021), k=1,10,100) and obfuscation degree (including *perplexity* measured by the LLM and *edit distance* to the original code). We start by presenting the efficacy in performance preservation, in other words, how the code obfuscated by CodeCipher maintains the performance of downstream tasks. Table 1 shows the results of various methods in the code completion task[3]. To ensure a fair comparison, we maintained a consistent level of obfuscation across all approaches and compared their accuracy on downstream tasks. The results indicate that CodeCipher effectively confuses source code without substantially compromising the original model's performance. For instance, "identifier renaming" and "random" perturbation drop Pass@1 from 50.60% to 42.68% and 40.24%, respectively, whereas our approach only reduces it to 47.56%. The results demonstrate that our method achieves a superior balance between code obfuscation and model performance compared to the baselines, making it more practical in real-world applications.

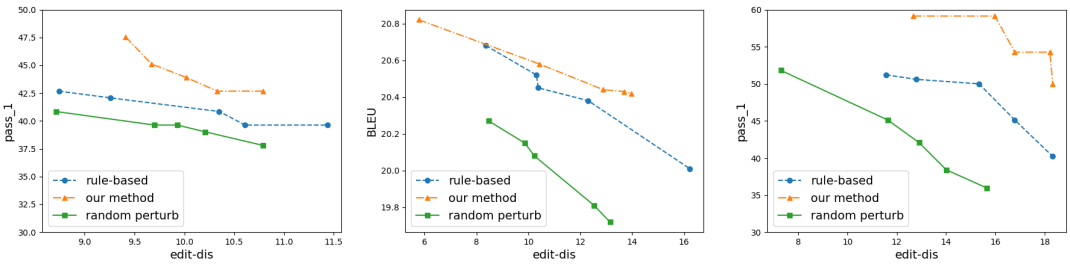
To comprehensively assess CodeCipher under different obfuscation levels. We vary the confusion degree of various methods and compare their task performance. We compare CodeCipher with random perturbation and identifier renaming, two strong baselines in the previous results. We control the obfuscation degree by varying the proportion of perturbed tokens in the entire code. The results are presented in Figure 4. Across all tasks and metrics, our method exhibits consistent strength over baselines under different obfuscation levels. The results suggest that our method strikes a more effective balance between code obfuscation and maintaining performance on downstream tasks.

5.3 Privacy Protection

Having established that CodeCipher effectively preserves the task performance, we now assess its efficacy in privacy protection, the primary objective of code obfuscation. We aim to determine whether the obfuscation can obscure the information, and prevent the code from compiling, execution, and model training.

Setup: We conduct experiments on the CodeSearchNet dataset which was collected from real-world projects in GitHub and may involve more private information. The raw dataset consists of 2 million samples. We randomly sampled 200 code samples and obfuscate them using various methods. We adopt three metrics: *1) Compilation Rate*, the proportion of compilable code among all obfuscated code, *2) Deobfuscation Rate*, which measures the proportion of obfuscated code that can be deobfuscated by LLMs. We compute this proportion using the recall score, i.e., proportion of tokens in the original code that also appear in the de-

[3]We show results of other tasks in Appendix



(a) Pass@1 under different edit distances for code completion

(b) BLUE scores under different edit distances for code summarization

(c) Pass@1 under different edit distances for code translation

Figure 4: Task performance under various confusion levels

Table 2: Performance on Privacy Protection on the CodeSearchNet Benchmark

Method	**Compilation Rate (%)**	**Deobfuscation Rate (%)**	**Deobfuscation Distance (%)**	
			Before deobfus.	**After deobfus.**
Random	6	35	10.23	39.89
Identifier renaming	76	38	10.27	39.09
Dead branch injection	96	35	10.25	37.77
Remove symbols	0	36	10.34	38.91
LLM prompting	84	32	10.39	39.44
Obfuscation+Inform	76	38	10.27	39.09
CodeCipher (ours)	0	34	10.42	43.18

obfuscated code. *3) Deobfuscation Distance*, refers to the edit distance that the deobfuacated code deviate from the original code. For ease of comparison, we also compute the edit distances for the initially obfuscated code. We deobfuscate the obfuscated code by prompting CodeLlama-7b-Instruct. The prompt used for deobfuscation is provided in Appendix B.

Results: As can be seen from Table 2, CodeCipher disrupts the compilability of all obfuscated code, thereby preventing its reuse. Particularly, the obfuscated code is challenging for LLMs to restore, with only 34% of tokens successfully restored. The same phenomenon can be seen from the deobfuscation distance. For all methods, the edit distances of the deobfuscated code are greater than those of the initially obfuscated code, probably because the deobfuscation process introduces additional modifications to the code. Notably, the code obfuscated by CodeCipher exhibits the largest edit distance from the original code compared to other methods. The results suggest that our approach successfully obfuscates code and enhances privacy protection.

5.4 Ablation Study

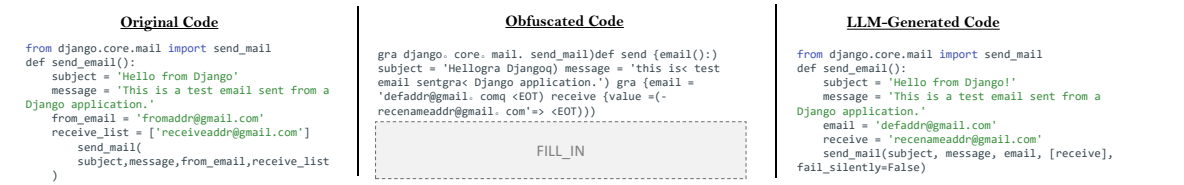
In this section, we conduct an ablation study on the discrete gradient search, a key step in our algorithm. We replace the discrete gradient search with traditional gradient descent, namely, a single gradient descent followed by decoding. We also vary the learning rates of gradient descent, a critical factor in our approach. As shown in Table 3, relying solely on a single-step gradient descent remarkably affects the code completion results. Traditional gradient descent results in suboptimal performance under all learning rates, indicating the efficacy of discrete gradient search in our method.

Table 3: Ablation study of the discrete gradient search on the code completion task

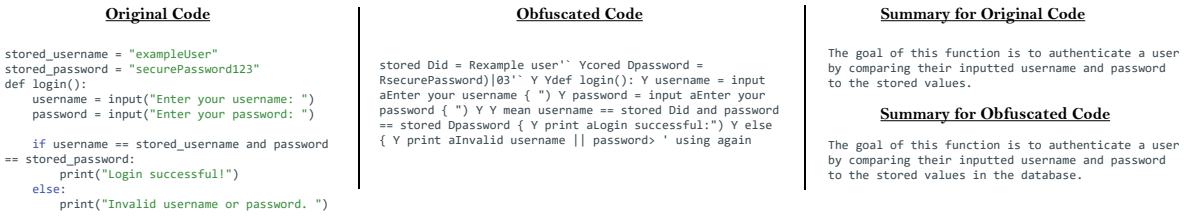
Model	Pass@1(%)	Pass@10(%)	Pass@100(%)	PPL	Edit distance(%)
CodeCipher	47.56	65.20	77.15	6.09	9.41
- w/o discrete gradient search (lr=5e-3)	39.63	57.25	67.68	5.07	2.70
- w/o discrete gradient search (lr=1e-2)	39.02	58.70	73.17	5.62	3.88
- w/o discrete gradient search (lr=2e-2)	43.90	63.52	77.04	5.66	4.42

5.5 CASE STUDY

Figure 5 presents two practical examples by CodeCipher. We observe that CodeCipher obfuscates sensitive information such as email address and passwords in the original code. The privacy contained therein is protected. Nonetheless, the model remains capable of understanding the code and generating correct but insensitive responses. In summary, the results show that our approach effectively obfuscates the sensitive information to LLMs, while preserving its functional correctness.



(a) A case for code completion. We obfuscate the first half of the original code and ask the LLM to regenerate the entire code given the obfuscated part.



(b) A case for code summarization. We obfuscate the entire code and compare the LLM-generated summaries for the original and the obfuscated code.

Figure 5: Examples of code obfuscation by CodeCipher

5.6 TRANSFERABILITY

Our approach is developed on white-box LLMs where the embedding layers and task gradients are accessible. In practice, there is always a need to apply the model in a new LLM, particularly black-box LLMs which provide only an API interface to users. We are unable to fine-tune these models based on their parameters. In this section, we aim to evaluate whether the obfuscated code generated by our method can still achieve good performance on other models.

Setup: We used the obfuscated code generated by CodeLlama-7B with a perturbed embedding layer to evaluate the performance on other models. We conducted the experiments on the code completion task, which is one of the most common and general tasks. The representative models chosen for evaluation include StarCoder (Li et al., 2023), DeepSeek-Coder (Guo et al., 2024), and GPT-3.5-turbo (OpenAI, 2023), which have shown strong performance on AI coding tasks. We employed greedy decoding during inference, and the HumanEval dataset was used for testing.

Results: As shown in Figure 6, our method obtains relatively stable Pass@1 scores in each model through different obfuscating models, and in most cases, the deviation from original scores is within 10%. In particular, in the black-box LLM GPT-3.5-turbo, the obfuscated codes outperform its original results, showing that our method remains high efficacy on other models. We hypothesize that our method learns common token mappings that are portable across LLMs. Overall, the results suggest that our method is not tied to any specific version of the LLM; instead, it is a general approach that works independently of the LLM version used. In practical applications, we can develop our method on a local proxy model such as the open-sourced CodeLlama-7B and apply the learned token confusion mapping to other cloud LLM servers.

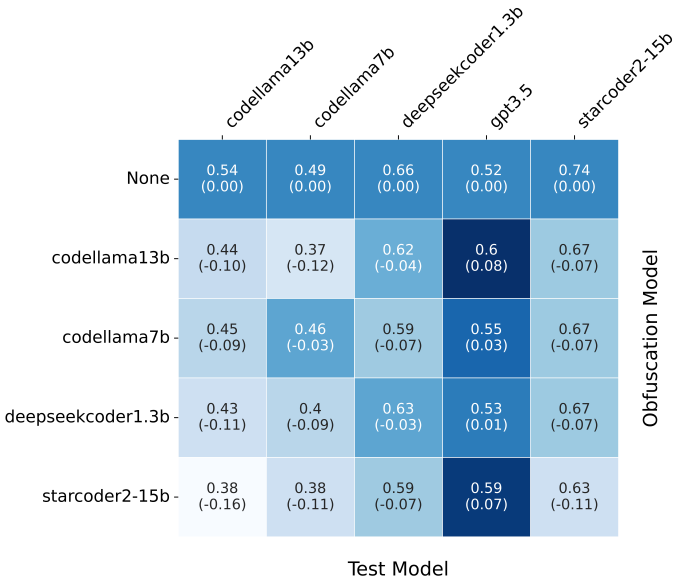


Figure 6: Results of the transferability test. The vertical axis represents different obfuscation models while the horizontal axis represents different test models. In each cell, the first line indicates the Pass@1 value achieved by the test model under the obfuscation model, and the value in parentheses shows the difference compared to the Pass@1 result without any obfuscation.

6 Conclusion

In this paper, we propose CodeCipher, a novel code obfuscation method tailored for LLMs. CodeCipher learns a transformation on the word embedding space, establishing a confusion mapping between code tokens that can be used for code obfuscation. The embedding transformation is optimized by minimizing the task-specific loss. A discrete gradient search algorithm is designed to tackle the sparsity challenge in the embedding space. Experiments on three AI-assisted coding tasks demonstrate that our method achieves a superior balance between code obfuscation and model performance compared to traditional rule-based methods.

REFERENCES

Ali Al-Kaswan, Maliheh Izadi, and Arie van Deursen. Traces of memorisation in large language models for code. *arXiv preprint arXiv:2312.11658*, 2023.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pp. 65–72, 2005.

Chandan Kumar Behera and D Lalitha Bhaskari. Different obfuscation techniques for code protection. *Procedia Computer Science*, 70:757–763, 2015.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023.

Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. Natgen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, pp. 18–30, 2022.

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

Xi Ding, Rui Peng, Xiangping Chen, Yuan Huang, Jing Bian, and Zibin Zheng. Do code summarization models process too much information? function signature may be all that is needed. *ACM Trans. Softw. Eng. Methodol.*, 33(6), jun 2024. ISSN 1049-331X. doi: 10.1145/3652156. URL https://doi.org/10.1145/3652156.

Eastridge-Technology. Jshrink, 2000. http://www.e-t.com/jshrink.html.

Tao Fan, Yan Kang, Guoqiang Ma, Weijing Chen, Wenbin Wei, Lixin Fan, and Qiang Yang. Fate-llm: A industrial grade federated learning framework for large language models. *arXiv preprint arXiv:2310.10049*, 2023.

Runhan Feng, Ziyang Yan, Shiyan Peng, and Yuanyuan Zhang. Automated detection of password leakage from public github repositories. In *Proceedings of the 44th International Conference on Software Engineering*, pp. 175–186, 2022.

Jonathon T Giffin, Mihai Christodorescu, and Louis Kruger. Strengthening software self-checksumming via self-modifying code. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pp. 10–pp. IEEE, 2005.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196, 2024. URL https://doi.org/10.48550/arXiv.2401.14196.

Jochen Hoenicke. Java optimize and decompile environment (jode), 2001. http://jode.sourceforge.net/.

Yizhan Huang, Yichen Li, Weibin Wu, Jianping Zhang, and Michael R Lyu. Do not give away my secrets: Uncovering the privacy issue of neural code completion tools. *arXiv preprint arXiv:2309.07639*, 2023.

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

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973*, 2020.

Tımea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19, 2009.

R Li, LB 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! *Transactions on machine learning research*, 2023.

Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pp. 74–81, 2004.

Guo Lin, Wenyue Hua, and Yongfeng Zhang. Promptcrypt: Prompt encryption for secure communication with large language models. *arXiv preprint arXiv:2402.05868*, 2024.

Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. *Acm transactions on knowledge discovery from data (tkdd)*, 1 (1):3–es, 2007.

Milad Nasr, Nicholas Carlini, Jonathan Hayase, Matthew Jagielski, A Feder Cooper, Daphne Ippolito, Christopher A Choquette-Choo, Eric Wallace, Florian Tramèr, and Katherine Lee. Scalable extraction of training data from (production) language models. *arXiv preprint arXiv:2311.17035*, 2023.

OpenAI. GPT-3.5-turbo. https://openai.com/, 2023. Accessed: 2024-09-21.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. *arXiv preprint arXiv:2308.03109*, 2023.

Omkant Pandey and Yannis Rouselakis. Property preserving symmetric encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 375–391. Springer, 2012.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015. doi: 10.1002/spe.2346. URL https://hal.archives-ouvertes.fr/hal-01078532/document.

Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*, pp. 390–401, 2014.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, I. Evtimov, Joanna Bitton, Manish P Bhatt, Cristian Cantón Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre D'efossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open foundation models for code. *ArXiv*, abs/2308.12950, 2023. URL `https://api.semanticscholar.org/CorpusID:261100919`.

Jieke Shi, Zhou Yang, Hong Jin Kang, Bowen Xu, Junda He, and David Lo. Greening large language models of code. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society*, pp. 142–153, 2024.

Yiping Song, Juhua Zhang, Zhiliang Tian, Yuxin Yang, Minlie Huang, and Dongsheng Li. Llm-based privacy data augmentation guided by knowledge distillation with a distribution tutor for medical text classification. *arXiv preprint arXiv:2402.16515*, 2024.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the 25th IEEE/ACM international conference on Automated software engineering*, pp. 43–52, 2010.

Zhensu Sun, Xiaoning Du, Fu Song, and Li Li. Codemark: Imperceptible watermarking for code datasets against neural code completion models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1561–1572, 2023.

Saiteja Utpala, Sara Hooker, and Pin Yu Chen. Locally differentially private document generation using zero shot prompting. *arXiv preprint arXiv:2310.16111*, 2023.

Biwei Yan, Kun Li, Minghui Xu, Yueyan Dong, Yue Zhang, Zhaochun Ren, and Xiuzheng Cheng. On protecting the data privacy of large language models (LLMs): A survey. *arXiv preprint arXiv:2403.05156*, 2024.

Fan Yan and Ming Li. Towards generating summaries for lexically confusing code through code erosion. *International Joint Conferences on Artificial Intelligence Organization*, 2021.

Zhou Yang, Zhensu Sun, Terry Zhuo Yue, Premkumar Devanbu, and David Lo. Robustness, security, privacy, explainability, efficiency, and usability of large language models for code. *arXiv preprint arXiv:2403.07506*, 2024.

Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing*, pp. 100211, 2024.

Lifan Yuan, Yichi Zhang, Yangyi Chen, and Wei Wei. Bridge the gap between cv and nlp! a gradient-based textual adversarial attack framework. *arXiv preprint arXiv:2110.15317*, 2021.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang 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*, pp. 5673–5684, 2023.

Xin Zhou, Yi Lu, Ruotian Ma, Tao Gui, Yuran Wang, Yong Ding, Yibo Zhang, Qi Zhang, and Xuanjing Huang. TextObfuscator: Making pre-trained language model a privacy protector via obfuscating word representations. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 5459–5473, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.337. URL https://aclanthology.org/2023.findings-acl.337.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. XLCoST: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*, 2022.

A THE ALGORITHM

The detailed algorithm for our approach is presented in Algorithm 1

Algorithm 1: CodeCipher Algorithm

Input: Input-output pairs for a specific task: $\langle X, Y \rangle$
The original embedding layer $\mathbf{E}$ of the LLM
Maximum training iterations N, PPL thresholds α, β, Maximum steps for each iteration T
Learning rate η
Output: The new embedding layer $\mathbf{E}'$ of the LLM
$\mathbf{E}' = \mathbf{E}$
for $i = 1$ *to* N **do**
  $(x_i, y_i) \sim \langle X, Y \rangle$ ▷ Randomly choose a training sample (x_i, y_i)
  $\mathbf{e} = \mathbf{E}'(x_i)$ $x_i' = \text{Dec}(\mathbf{e})$ ▷ Obfuscate the code using the previous confusion matrix
  // Early stop if the code is sufficiently confusing
  $\delta_{PPL} = \text{PPL}(x_i') - \text{PPL}(x_i)$
  if $\delta_{PPL} \leq \alpha * i + \beta$ **then**
    $\mathbf{e}_{best} = \mathbf{e}$, $\mathcal{L}_{\text{best}} = \mathcal{L}_{\text{task}}(x_i', y_i)$
    for $t = 1$ **to** T **do**
      $\mathbf{e}' = \Pi_{\mathcal{V}}(\mathbf{e})$ x_i' = Dec ($\mathbf{e}'$) ▷ Decode embedding to vocabulary space
      $\mathbf{e} = \mathbf{e} - \eta \nabla_{\mathbf{e}'} \mathcal{L}_{\text{task}}(x_i', y_i)$ ▷ Gradient update w.r.t. the projected embedding
      if $\mathcal{L}_{\text{task}}(x_i', y_i) < \mathcal{L}_{\text{best}}$ **then**
        $\mathbf{e}_{best} = \mathbf{e}$, $\mathcal{L}_{\text{best}} = \mathcal{L}_{\text{task}}(x_i', y_i)$ ▷ Record the best $\mathbf{e}$ that leads to the minimum loss
      end
    end
    $\mathbf{e}' = \Pi_{\mathcal{V}}(\mathbf{e}_{best})$ ▷ Final projection
    $\mathbf{E}'[x_i] = \mathbf{e}'$ ▷ Replace the corresponding token embedding in $\mathbf{E}'$ using $\mathbf{e}'$
  end
end
return $\mathbf{E}'$

B PROMPT TEMPLATES

Prompt for Code Completion

Please complete this code from head:
{code}
Please only output the code. Please complete this code from head:

Prompt for Code Summarization

Generate a docstring for the code in 10 words.
{code}
Please fill this sentence: 'The goal of this function is to' in 10 words

Prompt for Code Translation

Please translate this Java code to Python:
{code}
Please use a functional programming style

Prompt GPT-4o to perform code obfuscation

Obfuscate the following Python function to make it difficult for humans to understand while keeping the normalized edit distance WITHIN 0.10. Do not add or remove line breaks or blank spaces in the code. Please only output the code.
{Original-Code}

Prompt for Code Deobfuscation

I have an obfuscated code (remove symbol) for you to restore, and your output should only be the code, and the code should be in "' and "'. Please ONLY output the code, DO NOT output other things. The Obfuscated code is:
{code}
Please restore the original code:

C HYPERPARAMETER SETTING

The hyperparameters for the three tasks are provided in Table 4.

Table 4: Hyperparameter Setting.

Hyperparameter	Code Completion	Code Summarization	Code Translation
Learning rate (η)	0.002	0.002	0.002
Max sample number (N)	32	32	32
α	1.5	1.5	2.0
β	$\frac{1}{90}$	$\frac{1}{20}$	$\frac{1}{90}$
Maximum iterations per session (T)	10	10	10

To further analyze the sensitivity of hyperparameters, we vary two hyperparameters (T and η) and evaluate their effect on the performance. Results are presented in Figure 7 and 8 respectively. As T and η increase, the performance (Pass@K) drops while the obfuscation degree increases. The best trade-off of the two hyperparameters lies in 10 and 0.002 respectively.

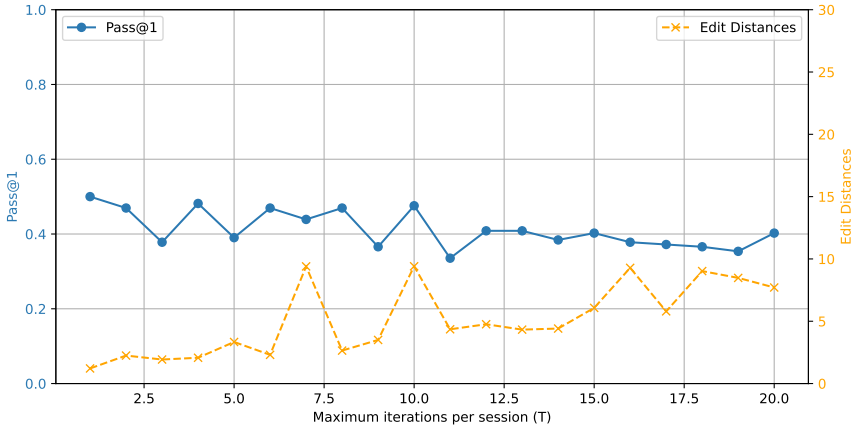


Figure 7: Performance over different numbers of sub-steps

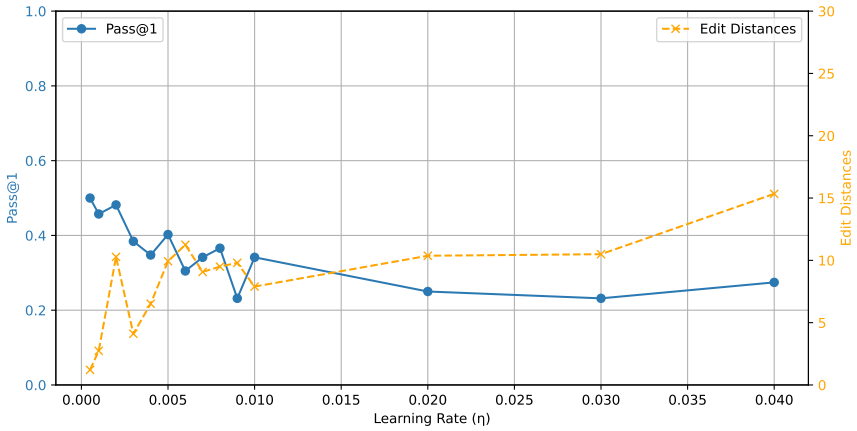


Figure 8: Performance over different learning rates

D RESULTS ON OTHER TASKS

Tables 5 and 6 show the results on the code summarization and translation tasks, respectively. We observe the same trend as in the code completion task: CodeCipher increases the perplexity and edit distance to the original code while achieving higher performance in the downstream tasks. In code summarization, our approach attains a PPL that is 2.73 times higher than the original while keeping the reduction in BLEU scores within 2%. In code translation, the PPL of our obfuscated code is three times higher than the original, yet the Pass@1 decreases by only 4%, and the Pass@100 score remains unchanged. These results confirm the efficacy of our approach in code obfuscation.

Table 5: Results on Code Summarization. We employ BLEU-4 (Papineni et al., 2002), ROUGE-L (Lin, 2004), and METEOR (Banerjee & Lavie, 2005) as the performance measures.

Method	Task-specific Performance			Obfucation Degree	
	BLEU	ROUGE-L	METEOR	PPL	Edit distance(%)
Origin	21.00	22.50	34.95	6.52	0
Random perturb	20.08	19.75	30.50	28.05	10.23
Conventional Obfuscation					
Identifier renaming	20.47	20.50	32.55	17.79	10.27
Dead branch injection	20.57	19.00	32.71	8.14	10.25
Remove symbols	20.00	21.25	32.86	18.02	10.34
LLM based Obfuscation					
Encipher with LLM prompting	20.03	**21.75**	20.99	9.27	10.39
Obfuscation + Inform	19.80	20.50	30.95	17.79	10.27
CodeCipher (ours)	**20.58**	**21.75**	**33.34**	**34.03**	**10.42**

Table 6: Results on Code Translation. We employ the widely used Pass@k (Chen et al., 2021) (k=1,10,100) as the performance measures.

Method	Task-specific Performance			Obfucation Degree	
	Pass@1(%)	Pass@10(%)	Pass@100(%)	PPL	Edit distance(%)
Origin	61.59	85.90	92.68	2.05	0
Random perturb	51.83	74.65	87.80	6.10	7.30
Conventional Obfuscation					
Identifier renaming	50.61	82.00	90.85	3.37	11.63
Dead branch injection	49.39	75.29	86.59	6.52	10.05
Remove symbols	47.50	76.88	85.76	6.25	3.79
LLM-based Obfuscation					
Encipher with LLM prompting	57.31	82.55	92.07	3.09	10.36
Obfuscation + inform	44.51	76.91	91.46	3.37	11.63
CodeCipher (ours)	**59.15**	**83.97**	**92.68**	**6.53**	**12.68**

E DISCUSSION

E.1 THE APPLICATION SCENARIOS OF OUR METHOD.

Our method is primarily built on white-box models. In some scenarios, due to limitations in computational resources, individual users may not be able to deploy large white-box models locally, necessitating a client-server architecture. However, in cases where the LLMs are closed-source, we take the trainable white-box model as a proxy, then apply the learned cipher mapping to the closed-source model. As demonstrated in Section 5.6, code obfuscated by a local white-box model transfers effectively to a remote closed-source model.

E.2 THE SECURITY OF CODECIPHER

Though our method is built on white-box models, the trained transformation mapping is inherently black-box. Since we do not disclose the model parameters, even if an attacker understands how the approach works, they would be unable to replicate the obfuscation process or perform de-obfuscation.