Unlearnable Examples: Protecting Open-Source Software from Unauthorized Neural Code Learning

Zhenlan Ji, Pingchuan Ma, Shuai Wang
The Hong Kong University of Science and Technology
{zjiae,pmaab,shuaiw}@cse.ust.hk

Abstract—**The vast volume of "free" code maintained on open-source code management systems significantly simplifies the process of producing and sharing open-source software. Recently, we have seen a growing trend in which these open-source software is being used for neural code learning without authorization. Note that open-source software does not necessarily imply "unrestricted usage," e.g., software under the BSD license requires users to retain the copyright notice and credit the software's developers.**

The unauthorized use of software for (commercial) neural code learning models has raised copyright concerns. This paper, for the first time, provides approaches for protecting open-source software from unauthorized neural code learning via unlearnable examples. Our proposed technique applies a set of lightweight transformations toward a program before it is open-source released. When these transformed programs are used to train models, they mislead the model into learning the unnecessary knowledge of programs, then fail the model to complete original programs. The transformation methods are sophisticatedly designed to ensure that they do not impair the general readability of protected programs, nor do they entail a huge cost. We focus on code autocompletion as a representative downstream task of unauthorized neural code learning. We demonstrate highly encouraging and cost-effective protection against neural code autocompletion.

I. Introduction

Recent advances in deep neural networks (DNNs) have resulted in advancements in computer vision (CV) and natural language processing (NLP) applications. Recently, there has been a surge of interest in using neural networks to solve a variety of software engineering (SE) tasks by learning from codes, including program synthesis [10], autocompletion [15], and code summarization [2]. For example, GitHub recently released Copilot [8], with the aim to provide an "AI pair programmer" capable of automatically generating programs from natural language specifications.

The major success of neural code learning is attributed in part to the availability of large-scale corpus [15]. For instance, Copilot is trained on massive amounts of open-source code, including public code collected from GitHub [3]. Nonetheless, a widespread concern is that some datasets were amassed without mutual consent [16]. In fact, it has been extensively noted that Copilot may leak sensitive code snippets when performing auto programming [23]. And, as its developers admit, Copilot uses all public Github code for training regardless of license [11], thus likely breaching the copyright of lots of open-source software. In this paper, we refer to the use of code as training data without consent as "*unauthorized utilization.*" Note that "open-source" software does **not** necessarily grant model owners like GitHub the right to sell the software content, nor does it grant model owners the right to distribute/use open-source software. In short, these neural code learning applications have raised primary concerns about unauthorized usages of open-source software.

There are solutions to protect private data against unauthorized machine learning. Recent work, in particular, generated unlearnable images via adversarial transformations [9], [24], [4], fooling the model into believing that there is nothing that can be learned from the transformed images. This work advocates for a focus on creating unlearnable examples of code that are difficult for neural code learning models to learn and, meanwhile, exhibit identical functionality and high readability to humans. However, in comparison to images, it is more difficult to practically enforce unlearnability on software with "adversarial transformations," because arbitrarily flipping a token in a program may change its semantics and even impose grammatical errors. Existing techniques (e.g., software obfuscation) used to protect software from exploitations are often heavyweight, which significantly impair the readability and is thus undesirable for open-source software.

In this paper, we design a set of lightweight *semantics-preserving* transformations toward software. Users who plan to open-source release their software can first locally launch our transformations toward their programs before releasing them. Given the inadequacy of a single transformation, we form an iterative transformation process and identify an optimal transformation sequence using multi-armed bandits. We employ a commonly-used code embedding model to guide local transformation, with the aim of maximizing the embedding distance between a transformed program and its clean version while preserving a modest edit distance (to retain readability). Users can then release the transformed program, and when the released program is used as training data for neural code learning (which is generally *not avoidable* for open-source software), these transformed programs deceive downstream applications like autocompletion by constantly learning faulty knowledge. Thus, the transformed program though remains accessible to the open-source community, becomes unlearnable from the perspective of unauthorized neural models.

We use CodeBERT [7] as the local model to guide the transformation. We deceive CodeGPT [12], a SOTA code

DOI reference number: 10.18293/SEKE2022-066

autocompletion model trained on the POJ-104 dataset [15] with one or several programs that have been transformed by our work. The evaluation shows that the transformed programs can achieve a high success rate of creating unlearnable examples, greatly reducing the accuracy of CodeGPT (to nearly "random"), while maintaining decent readability and a small runtime cost.

Contributions. We summarize our contributions as follows: 1) conceptually, we advocate for a new focus on protecting software against unauthorized neural code learning, a growing concern in the open-source community, 2) technically, we propose a set of lightweight transformations to transform software in a semantics- and readability-preserving manner. The optimal transformation sequence toward a program is determined using multi-armed bandits, and 3) we demonstrate empirically that transformed programs effectively mislead unauthorized neural code autocompletion with negligible cost.

Open Source. We release our code at https://github.com/ZhenlanJi/Unlearnable_Code.

II. Related Work

Mitigating Data Privacy Leakage. The majority of work on reducing model training data privacy leaks is based on federated learning [13]. Rather than sharing the training data, owners of training data share the model updates to jointly train a model. Contemporarily, differential privacy assures that a trained model does not learn raw training data [6]. In contrast, our work focuses on a more challenging scenario: protecting open-source software from being learned by unauthorized neural models. That is, the deep learning models are not trusted in our scenario, and it is unclear if they have utilized any privacy-preserving techniques like differential privacy. Recent efforts have been made to protect the privacy of photos with adversarial transformations against facial recognition [24], [4]. We protect open-source software, a timely albeit under-explored subject. We propose a general framework that produces unlearnable programs that can be used to mitigate unauthorized code embedding applications.

Protecting Unauthorized Code Usage. Some prior research focuses on identifying code authorship [1]. Nevertheless, protecting unauthorized usage of open-source software, though having raised widespread concern, is rarely discussed. One contemporary research inserts unusable code snippets ("deadcode") to impede unauthorized code usage [20]. Nonetheless, inserting deadcode, e.g., a special function that is never executed (referred to as "watermark" in their paper), is generally easy to be recognized and removed. Transformations proposed in this work are stealthy and hard to elide, meaning that our approach is more robust and effective in defeating unauthorized code usage.

III. Approach Overview

Research Challenge. Existing works generate "unlearnable" photos by applying adversarial transformations [24], [4]. These approaches, however, are inapplicable to software. Software is written in a structured manner, where transforming arbitrary bytes in a program can easily break its functionality. On the other end of the spectrum, software obfuscations techniques [5] transform software into an unreadable form, which may be used to defend unauthorized utilization. However, obfuscations can largely hamper software readability and distribution in the open-source community and consequently is *not* widely used. In sum, we aim to deliver a lightweight and effective method, such that software is transformed without impeding its functionality, retaining high readability, while making it "unlearnable."

Assumption and Objective. We aim to design a set of transformations, particularly toward software. These transformations change the program source code in a semantics-preserving manner while retaining readability and execution speed. This way, we protect open-source software against unauthorized neural code learning, which is jeopardizing copyright of open-source software for unauthorized usages.

While authors may have no direct access to the unauthorized neural code models, they can set up SOTA code embedding models like CodeBERT locally to guide transformations. However, once the transformed software is released as open-source, authors cannot prevent unauthorized usage, which will include the released software as part of their training data. We establish a practical objective for protection, such that the released software, after local transformation, cannot be correctly matched to its original version in front of representative neural code learning applications like code autocompletion. In the rest of the paper, we use "clean version" to refer to the original, untransformed program to ease the reading.

TABLE I
TRANSFORMATION METHODS.

Class	Methods	Abbreviations
Identifier Level	identifier replacement	`IR`
	identifier synonym	`IS`
	character synonym	`CS`
Constant Level	int rewrite	`INT`
	float rewrite	`FLT`
	string rewrite	`STR`
Statement Level	int def. insert	`IDI`
	float def. insert	`FDI`
	string def. insert	`SDI`
	single line insert	`SLI`

A. Semantics-Preserving transformation

This research designs a set of transformation methods for programs. Each method provides a *semantics-preserving* transformation, in the sense that the transformed output, another piece of software, retains the original functionality. We further clarify that all transformation methods are intended to provide *simple* and *incremental* transformations, while heavyweight transformations can undermine the readability of open-source software. Table I lists all designed transformation methods.

Identifier Level. We first parse a program to extract all identifiers (i.e., variable names). We then design three transformation methods on extracted identifiers. Identifier replacement (`IR`) randomly replaces a subset of identifiers with random strings. Identifier synonym (`IS`) is built on the basis of WordNet [14], a large-scale lexical database. We query WordNet for a given identifier i to see if i and its synonyms exist; if so, we replace i with a randomly selected synonym. Compared with `IR`, `IS` can better preserve the readability of transformed programs. We also propose character synonym (`CS`) which uses hardcoded rules to replace a single-character identifier with another character, e.g., `i` $\rightarrow$ `j`.

As discussed in Sec. II, many code embedding models regard program statements as sentences, with identifiers (and also constants; see below) treated as tokens. This allows software to be smoothly processed via NLP techniques. Accordingly, we envision that identifier-level rewriting, though lightweight and retains readability to a large extent, will be useful in token-level transformation and deceiving neural models. This intuition is supported in our evaluation (Sec. V).

Constant Level. Additionally, we also propose three schemes to transform constants extracted in a program without breaking the semantics. Int rewrite (`INT`) converts an integer into an arithmetic expression. For instance, after applying `IR`, the C statement `a = a + 10` will be converted into `a = a + 121 - 21`, where "121" and "21" are two randomly-decided constants that will guarantee to yield "10". Similarly, we design a transformation scheme, namely float rewrite (`FLT`), to convert floating point numbers into arithmetic expressions. As for strings, our scheme string rewrite (`STR`) converts a `char*` constant in C (or `string` constant in C++) into two substrings, where the original string is recovered during runtime by calling the `libc` function `strdup` followed by calling another `libc` function `strcat`.

Existing DNN-based authorship identification and code clone detection gain from matching "magic numbers," which denote representative constants [18]. As a result, transforming constants should be effective in misleading the model. Moreover, as mentioned in Sec. II, some code embedding techniques learn the program abstract syntax tree (AST). Accordingly, by converting constants to expressions, we obscure the AST without largely impeding readability.

Statement Level. We also design methods to extend existing statements or insert new statements. For the `IDI`, `FDI`, and `SDI` schemes, we split an existing declaration statement of integer, float, or string variables into multiple declarations. For instance, `IDI` adds one extra statement with integer arithmetic computation, ensuring that the result is equal to the original declarations. Similarly, `FDI` and `SDI` insert new statements that perform floating-point arithmetics or string manipulations without changing the original functionality. In contrast, `SLI` generates and inserts new declarations at random, whose declared variables are never used in subsequent computation.

While the semantics is retained during transformation, the newly-inserted statements can complicate program AST and control flow graphs. We anticipate that neural code learning models that rely on AST or program structural-level information will struggle to understand the transformed programs.

Algorithm 1: Protection framework.

```
Input: Clean Program $s$, Transformation Sequence Length $K$,
       Sample Size $M$, Batch Size $m$, Exploration Factor $\epsilon$,
       Discounting Factor $\gamma$
Output: transformed Program $\hat{s}$
1  $\hat{s} \leftarrow s$;
2  foreach $k \leftarrow 1, \cdots, K$ do
3      $D \leftarrow [0]^{|\mathcal{T}|}$; // initialize expectation distribution
4      $S \leftarrow \emptyset$; // transformed software set
5      foreach $t \leftarrow 1, \cdots, M/m$ do
6          $T \leftarrow \emptyset$;
7          foreach $i \leftarrow 1, \cdots, m$ do
8              $t \leftarrow$ random() with probability of $\gamma^{i-1}\epsilon$;
9              $t \leftarrow \arg\max_t D_t$ with probability of $1 - \gamma^{i-1}\epsilon$;
10             $T \leftarrow T \cup \{t\}$;
11         end
12         transform $\hat{s}$ with all $t \in T$ and update $D$;
13         record all transformed program to $S$;
14     end
15     $\hat{s} \leftarrow \arg\max_{s \in S} f(s)$;
16 end
17 return $\hat{s}$
```

IV. Framework Design

Motivation. Each transformation scheme complicates program structures to some degrees, and different schemes may achieve a *synergistic effect* by iteratively modifying a program, with the output of each iteration, a transformed program, serving input of the next iteration. As a result, the applied transformation sequences (we have ten transformation methods) create a vast search space, 10^K, where K represents the number of iterations applied to the input. We formulate searching for the optimal transformation sequence as an optimization process, where statistical methods can help to promptly explore the search space and find optimal sequences.

Framework. The input of our pipeline is a program s. Let the local embedding model (e.g., CodeBERT) be $\mathcal{E}$, we iteratively transform s into $\hat{s}$ until $\hat{s}$ manifests a large embedding distance and also a small edit distance with s. This way, the "identity" of s will be hidden, given that $\hat{s}$ is seen as irrelevant to s in the view of M while similar to s in view of humans (by edit distance). Users can then release $\hat{s}$. Soon we will show that $\hat{s}$ will be protected from disclosing the information of s even if code autocompletion models are trained over $\hat{s}$.

Alg. 1 illustrates the protection workflow. For each iteration, we search for an optimal transformed software with ϵ`-Greedy` (line 3–15) such that a predefined objective function is maximal. Note that since transforming software is stochastic (e.g., the choice of synonyms is random), the effectiveness of each transformation method forms a random variable. Therefore, it requires extensive sampling to derive an optimal transformed software. Our sampling strategy is largely enlightened by ϵ`-Greedy` in standard multi-armed bandit problem. In this setting, the sampling strategy is progressively refined and favors the transformation methods that have good historical performance. First, it samples a batch of transformation methods: it 1) takes a random method with the probability of ϵ (line 8) or 2) takes the best method (w.r.t. historical expectation on f) with the probability of

$1-\epsilon$ (line 9). Typically, the former case explores all possible transformation methods, while the latter seeks to maximize the expected value of f. Alg. 1 collects samples in T and transforms $\hat{s}$ with each $t \in T$ respectively, and updates the table of expected value per batch (line 12–13). We define the objective function f as:

$$f(\hat{s}; s) = \frac{\texttt{l2-norm}(\mathcal{E}(\hat{s}), \mathcal{E}(s))}{\texttt{edit-dist}(\hat{s}, s)} \quad (1)$$

where `l2-norm` measures the euclidean distance of transformed $\hat{s}$ and its clean version, while `edit-dist` measures the edit distance between them. By doing so, we presume Alg. 1 will gradually find $\hat{s}$ with sufficiently long euclidean distance while retaining a small edit distance with s.

Hyper-parameters. Alg. 1 takes hyper-parameters. K denotes the length of the transformation sequence (i.e., how many iterations are allowed to transform s); M denotes the sample size for deciding one single transformation method in ϵ−`Greedy`. m is defined as the batch size, where the distribution table is updated per batch (in contrast to per sample). ϵ is the exploration rate that balances exploration and exploitation in ϵ−`Greedy`, and γ is the discounting factor that reduces the probability of exploration when the distribution is well captured by prior trials. Overall, larger K and M indicate more intensive transformation. For the current implementation, we set $K = 15$ and $M = 256$. Our evaluation shows that this configuration enables a reasonable tradeoff between effectiveness and cost. Users are encouraged to configure these two hyper-parameters according to their own usage scenarios. For m, ϵ, and γ, they are all common settings for ϵ−`Greedy`, where $m = 64$, $\epsilon = 1$, and $\gamma = 0.5$ in our implementation.

V. Evaluation

Neural Embedding. We use `CodeBERT`, a SOTA code embedding model to guide our local transformation shown in Alg. 1. We have introduced the high-level concept behind `CodeBERT` in Sec. II. We emphasize, however, that our protection pipeline is *orthogonal* to specific embedding models used during local transformation.

Test Dataset. We use POJ-104 [15], a widely-used dataset containing 52,000 C/C++ programs written for 104 tasks. These programs implement programming assignments by students (e.g., two sum). While programs belonging to the same task share identical functionality, programs in different tasks are irrelevant. On average, each POJ-104 program contains about 36 lines of code (exclude white space and comments), whose length is comparable or outperforms the program datasets used by relevant works [21], [22], [25].

Code Autocompletion. We measure how the transformed programs can successfully mislead unauthorized code autocompletion models when being exposed. This is a timely topic, where modern code autocompletion tools like Copilot have raised concerns by training code on GitHub without distinguishing licenses. We use CodeGPT [12], a transformer-based code autocompletion model, for this task. CodeGPT extends the standard GPT-2 model structure and demonstrates that it performs at a SOTA level in this line of research [12].

Setup & Metrics. We measure when training CodeGPT using our transformed programs, whether the transformed program can be successfully protected from being used for *self-completion*. Let the training split of POJ-104 contain N programs, we measure three setups: randomly selecting $\{1, 20\%, 40\%\}$ programs and replacing them with their transformed versions using our approach. Note that "1" indicates that only one program is being transformed. We then train three CodeGPT models with these three training datasets. We consider two baselines: 1) B_1, which uses the original training dataset to train CodeGPT, and 2) B_2, which replaces those transformed $\{1, 20\%, 40\%\}$ programs with irrelevant POJ programs. Ideally, CodeGPT trained using our transformed programs would behave similarly to B_2 while deviating significantly from B_1, showing that the identities of protected programs have been successfully hidden.

To assess autocompletion, the standard approach randomly splits a program p into two pieces, p_1 and p_2, and uses p_1 as the model input to determine whether the model-generated piece p_2' matches p_2. Autocompletion models are typically assessed in terms of their one-line, three-line, and five-line completion accuracy, such that we match the first one line, three lines, and five lines of p_2' and p_2. To "match" two code snippets, both edit similarity (ES) and exact match (EM) are employed, with ES computing the tree edit similarity between the two code snippets (higher is better), and EM requiring an exact match between the two code snippets. We clarify that these two metrics are consistent used in measuring the performance of CoedXGLUE [12].

Model Training. Our learning and testing were conducted on a server machine with an Intel Xeon E5-2683 v4 CPU at 2.40 GHz with 256 GB of memory and two Nvidia 2080 GPU cards. The machine runs Ubuntu 18.04. Note that we use a pre-trained `CodeBERT` model to compute code embeddings. We share the same hyperparameters with CodeXGLUE to train the CodeGPT model. To benchmark the performance of our trained model, we also evaluate it on dataset py150 provided by CodeXGLUE. The result shows that our model has a comparative performance on py150 with CodeXGLUE's official report. The average training cost for each CodeGPT model is about five hours. It takes about one minute to transform one program using methods in Sec. IV.

A. *Generating transformed Code Samples*

TABLE II
Average embedding distance and edit distance of 1) transformed programs and their clean versions; 2) same class programs; and 3) cross class programs.

	transformed vs. Clean	Same Class	Cross Class
Embedding Dist.	3.54	3.54	4.29
Edit Dist.	138.2	480.7	587.8

transformation. We first compute and compare the average embedding distance and edit distance between transformed programs and corresponding clean versions in Table II. Furthermore, given that programs in POJ-104 are annotated with different classes, we also report pairwise distance among programs of the same class or cross classes. Our approach effectively increases the difference between transformed and clean versions of the programs in CodeBERT's view, while retaining a reasonable edit distance.

The edit distance between the transformed and clean programs is much lower than that between programs of the same class. Programs from different classes implement distinct tasks, resulting in even longer edit distance. This reveals the high similarity in the view of users. We present further discussions on readability below in Table IV. In short, our transformation is shown to be effective in misleading neural code embedding, whose effectiveness will be further illustrated by defeating CodeGPT in Sec. V-B.

TABLE III
TRANSFORMATION DISTRIBUTION.

transformation	Portion	transformation	Portion
`IR`	26.6%	`STR`	6.3%
`IS`	3.1%	`IDI`	3.7%
`CS`	14.1%	`FDI`	0.1%
`INT`	29.3%	`SDI`	0.8%
`FLT`	1.1%	`SLI`	14.8%

Distribution of Applied transformations. As shown in Table I, we implement ten transformation methods. Recall that, in Alg. 1, a transformation method is retained, in case it effectively reduces embedding distance without primarily undermining the edit distance. Table III reports the distribution of successfully retained transformations. Overall, we interpret that all proposed methods (except `FDI`) are applied for a non-trivial amount of iterations. Identifier-level and constant-level transformations are particularly effective to deceive CodeBERT. To compute embeddings, CodeBERT extracts a large number of string and integer constants (including variable names) from input programs. Accordingly, by transforming identifiers and constants, CodeBERT can be effectively deceived. Note that the `IS` scheme is used less frequently. `IS` replaces a variable name with its synonym by querying WordNet. We find that a considerable fraction of variable names lack entries or synonyms in WordNet. Recall `FLT` extends a floating number into an arithmetic expression, and `FDI` rewrites a declaration statement for floating point variables. POJ-104 programs rarely use floating numbers. According to our observation, another reason that impedes the usage of statement-level transformations (`IDI`, `FDI`, `SDI`) is that they induce a higher edit distance, thereby undermining readability.

TABLE IV
AVERAGE SIMILARITY DECIDED BY JPLAG COMPARING TRANSFORMED PROGRAMS AND THEIR CLEAN VERSIONS.

transformed vs. Clean	Same Class	Cross Class
67.5%	4.4%	0.4%

Readability. The "readability" of software is often subjective and difficult to quantify. In addition to the edit distance, which our pipeline optimizes for, we provide another metric for the readability of transformed programs using conventional code similarity analyzers. At this point, we use a popular similarity checker called JPlag [17]. JPlag is widely used to detect code plagiarism based on syntactic and code structure-level features. Thus, using JPlag to evaluate code similarity provides a more complete picture of the readability of transformed code. JPlag can be configured locally prior to use. We also looked at another well-known tool, Moss [19]. Nonetheless, its remote server frequently fails to respond.

We assess randomly selected samples within POJ-104 as a baseline (same setting as Table II). We find that when randomly selected code samples from different classes are compared, the baseline similarity between them is only 0.4%, demonstrating that JPlag is capable of successfully distinguishing distinct programs. Note that JPlag can also distinguish programs belonging to the same class (for example, two quick sort programs) as long as their implementations are sufficiently distinct (average similarity 4.4%). In contrast, transformed programs exhibit a high degree of similarity to their clean versions, with an average similarity score of 67.5%. As a result, we interpret that these transformations do not primarily harm open-source software's "readability" and disseminability.

Cost. Our proposed transformations are functionality preserving, meaning that the transformed programs manifest identical functionality with their clean versions by design. Nevertheless, our transformations insert new code fragments into the programs, imposing additional performance penalty. We manually write non-trivial test cases for POJ-104 programs to assess performance penalty. We use a common performance analysis tool on Linux, `perf`, to measure the cost of (transformed) programs. In general, we report that the transformed programs become *negligibly* slow (on average less than 1%), if at all detectable. This is not surprising, as our methods primarily change symbols (variable names) and statements in a lightweight manner. Symbols are removed during compilation and hence have no effect on execution. In terms of statement-level changes, we find that many of them are optimized out during compilation. We interpret the cost evaluation as encouraging, illustrating that our transformation would incur negligible extra cost.

TABLE V
ASSESSING THE PERFORMANCE OF CODEGPT. B_1 AND B_2 ARE TWO BASELINE SETTINGS CLARIFIED IN THE **SETUP & METRICS** PARAGRAPH. UE (UNLEARNABLE EXAMPLE) DENOTES OUR RESULTS. "ES" STANDS FOR EDIT DISTANCE (A MORE TOLERATE METRIC OF CODE SIMILARITY), WHEREAS "EM" STANDS FOR EXACT MATCH.$B_2(20\%, 40\%, 1)$ AND UE $(20\%, 40\%, 1)$ CORRESPOND TO REPLACE $20\%, 40\%, 1$ OF CLEAN TRAINING PROGRAMS WITH OUR TRANSFORMED VERSIONS. UE IS EXPECTED TO CLOSE TO B_2 FOR HIGH PROTECTION ABILITY.

Method	1 line		3 lines		5 lines	
	ES	EM	ES	EM	ES	EM
B_1	73.9%	42.4%	71.2%	21.4%	68.3%	10.9%
B_2(20%)	68.8%	34.9%	67.3%	15.9%	64.9%	7.2%
UE (20%)	69.3%	35.5%	67.7%	16.2%	65.2%	7.6%
B_2(40%)	68.6%	34.5%	66.7%	15.6%	64.3%	6.7%
UE (40%)	68.4%	33.6%	66.7%	15.6%	64.2%	7.0%
B_2(1)	74.3%	48.8%	65.8%	10.2%	62.7%	0.9%
UE (1)	73.9%	45.8%	65.7%	10.8%	63.6%	2.2%

B. Mitigating Code Autocompletion

Table V reports the results of mitigating CodeGPT in different settings. When more lines are checked, CodeGPT's accuracy decreases. This is reasonable, as matching three or five lines implies a more difficult task than matching one line.

Recall that B_1 feeds CodeGPT with programs in its training dataset for self-completion, denoting the "upper bound" of accuracy. Table V reports promising results where the protected programs (three "UE" rows) are far from the B_1. More importantly, the "UE" rows are extremely close to their corresponding B_2 rows. As previously clarified, Baseline$_2$ denotes the "lower bound" of accuracy, as it replaces the transformed programs in the training dataset with irrelevant programs. Therefore, the evaluation results illustrate that, after transformation, protected programs behave similarly to randomly-picked programs, successfully deceiving the autocompletion model.

The 8th and 9th rows ($B_2(1)$ and UE (1)) denote inserting only *one* transformed program in the training dataset and feeding its clean version to CodeGPT (as noted in **Setup & Metrics**, we will randomly cut the input program into two and feed CodeGPT with the upper cut). This is a realistic setting, given that authors may want to protect their own piece of software before uploading it to GitHub. Though only changing one piece of training data, it already largely undermines the accuracy of CodeGPT when performing autocompletion toward the upper half of its clean version.

Clarification. We also clarify that the accuracy of CodeGPT in matching other programs (which may be also within the training data) has only negligible change (around 1%; particularly for the evaluation setting where 40% training data are transformed) compared with B_1. In summary, though it is generally hard (if at all avoidable) to prevent open-source software from being used as training data, information regarding the open-source software will not be leaked via autocompletion after applying our protection.

Case study. We provide an code example to illustrate the effectiveness of our approach to impede autocompletion at https://github.com/ZhenlanJi/Unlearnable_Code/blob/main/example.pdf.

VI. Conclusion

In this paper, we propose to mitigate unauthorized neural code learning from using open-source software. We design a set of lightweight transformations and explore optimal transformation sequences using multi-armed bandits. Our evaluation demonstrates that transformed programs can successfully deceive a SOTA code autocompletion model, CodeGPT.

Acknowledgement

We are grateful to the anonymous reviewers for their valuable comments. The work is supported in part by NVIDIA Academic Hardware Grant Program. Shuai Wang is the corresponding author of this paper.

References

[1] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. Large-scale and language-oblivious code authorship identification. In *CCS*, 2018.

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. ICLR.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[4] Valeriia Cherepanova, Micah Goldblum, Harrison Foley, Shiyuan Duan, John Dickerson, Gavin Taylor, and Tom Goldstein. LowKey: leveraging adversarial attacks to protect social media users from facial recognition. *arXiv preprint arXiv:2101.07922*, 2021.

[5] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[6] Cynthia Dwork. Differential privacy. In *International Colloquium on Automata, Languages, and Programming*, pages 1–12. Springer, 2006.

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[8] GitHub. Copilot, 2021.

[9] Hanxun Huang, Xingjun Ma, Sarah Monazam Erfani, James Bailey, and Yisen Wang. Unlearnable examples: Making personal data unexploitable. *arXiv preprint arXiv:2101.04898*, 2021.

[10] Chen Liang, Jonathan Berant, Quoc Le, Kenneth D Forbus, and Ni Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. In *ACL*, 2017.

[11] Andrew Liu. Copilot, 2021.

[12] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[13] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[14] George A Miller. WordNet: a lexical database for english. *Communications of the ACM*, 1995.

[15] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, 2016.

[16] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. An empirical cybersecurity evaluation of github copilot's code contributions. *arXiv preprint arXiv:2108.09293*.

[17] Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. Finding plagiarisms among a set of programs with JPlag. *J. UCS*, 8(11):1016.

[18] Erwin Quiring, Alwin Maier, and Konrad Rieck. Misleading authorship attribution of source code using adversarial learning. In *USENIX Security*, 2019.

[19] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD*, 2003.

[20] Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. Coprotector: Protect open-source code against unauthorized training usage with data poisoning. *arXiv preprint arXiv:2110.12925*, 2021.

[21] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *ICSME*, 2014.

[22] Jeffrey Svajlenko and Chanchal K Roy. Evaluating clone detection tools with bigclonebench. In *ICSME*, 2015.

[23] Jake Williams. Copilot privacy leakage, 2021.

[24] Xiao Yang, Yinpeng Dong, Tianyu Pang, Hang Su, Jun Zhu, Yuefeng Chen, and Hui Xue. Towards face encryption by generating adversarial identity masks. In *ICCV*, 2021.

[25] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *arXiv preprint arXiv:1909.03496*, 2019.