CodeMark: Imperceptible Watermarking for Code Datasets against Neural Code Completion Models

Zhensu Sun*
Beihang University
Beijing, China
zhensuuu@gmail.com

Xiaoning Du*
Monash University
Melbourne, Victoria, Australia
xiaoning.du@monash.edu

Fu Song†‡
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing, China
songfu1983@gmail.com

Li Li†
Beihang University
Beijing, China
lilicoding@ieee.org

ABSTRACT

Code datasets are of immense value for training neural-network-based code completion models, where companies or organizations have made substantial investments to establish and process these datasets. Unluckily, these datasets, either built for proprietary or public usage, face the high risk of unauthorized exploits, resulting from data leakages, license violations, etc. Even worse, the "black-box" nature of neural models sets a high barrier for externals to audit their training datasets, which further connives these unauthorized usages. Currently, watermarking methods have been proposed to prohibit inappropriate usage of image and natural language datasets. However, due to domain specificity, they are not directly applicable to code datasets, leaving the copyright protection of this emerging and important field of code data still exposed to threats. To fill this gap, we propose a method, named CodeMark, to embed user-defined imperceptible watermarks into code datasets to trace their usage in training neural code completion models. CodeMark is based on adaptive semantic-preserving transformations, which preserve the exact functionality of the code data and keep the changes covert against rule-breakers. We implement CodeMark in a toolkit and conduct an extensive evaluation of code completion models. CodeMark is validated to fulfill all desired properties of practical watermarks, including harmlessness to model accuracy, verifiability, robustness, and imperceptibility.

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; • **Computing methodologies** → *Artificial intelligence*; • **Applied computing** → **Computer forensics**.

KEYWORDS

Neural code completion models, Watermarking, Code dataset

ACM Reference Format:
Zhensu Sun, Xiaoning Du, Fu Song, and Li Li. 2023. 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 (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616297

1 INTRODUCTION

The immense value of high-quality code datasets has unprecedentedly been made visible with the advancement of deep learning (DL) and its application in code understanding and completion tasks [48]. Large language models, revealing an extraordinary capability to absorb knowledge from enormous language data corpus, have been applied to develop commercial Neural Code Completion Models (NCCMs), including Github Copilot [4], aiXcoder [2], TabNine [3], and CodeWhisperer [6]. An essential factor in the success of NCCMs is their high-quality and large-scale training datasets.

Code datasets, serving as invaluable digital assets, come with substantial costs in terms of the effort required for their collection and processing. During the data collection, millions of lines of source code are collected from multiple sources, ranging from open-source code to proprietary source code, to enlarge the scope of the dataset and provide diverse and comprehensive code patterns to the training models. Acquiring access to some code sources can involve negotiating licensing agreements, respecting intellectual property rights, and sometimes paying fees for the necessary permissions. For instance, Github Copilot collects code snippets from its users (under consent) to improve its model [5] through further training procedures, and the training data of Amazon's CodeWhisperer also includes the private code of Amazon itself [8]. Even open-source communities, such as StackOverflow, have begun to charge AI models for collecting their data [10]. On the other hand,

[*] Both authors contributed equally to this research.
[†] Corresponding authors
[‡] Also with University of Chinese Academy of Sciences, and Automotive Software Innovation Center.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12...$15.00
https://doi.org/10.1145/3611643.3616297

the collected raw source code demands rigorous processing and filtering to ensure that the dataset is free from redundant, unethical, or incorrect code snippets. For example, StarCoder [25] recruited thousands of annotators to help remove the personally identifiable information in its code dataset. Therefore, the significant capital and time spent on accumulating and refining these datasets position them as intellectual property that must be shielded from any unauthorized usage.

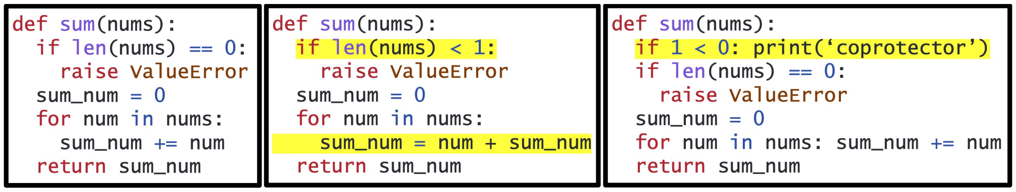
Currently, without any special protection, unauthorized usage of code datasets can easily happen regardless of whether the datasets are proprietary or public, which harms the rights and interests of dataset curators. Public datasets, though available to everyone, such as CodeSearchNet [21], The Stack [23] and PublicGitArchive [28], are restrictive in where and how they can be used. For example, PublicGitArchive does not allow any commercial usage. Propriety datasets, which are usually kept in secure environments, may get leaked in various cases such as cybersecurity attacks. When a leakage happens, the dataset owners will lose control over the datasets, which means the rule breakers can use the dataset freely. For models trained with these datasets, it is difficult to obtain digital forensics on the infringement because the "black-box" nature of DL models sets a high barrier for externals to audit their training datasets and connives these unauthorized usages.

To address the aforementioned concerns, researchers have proposed watermarking methods for defending against unauthorized usage of training datasets [22, 26, 39], most of which focus exclusively on image or natural language datasets. Watermarking does not directly prevent any unauthorized usage but instead discourages rule breakers by providing a means to break the "black-box" nature of DL models. However, little attention has been paid to the textual watermarks that are applicable to code datasets, leaving the copyright protection of this emerging and important field still exposed to threats. The only existing code watermarking method against neural models is CoProtector [38], where a dead-code-based watermarking method is proposed. However, the inserted dead code is of poor imperceptibility and might be easily spotted through human inspection [25] or static code analysis tools. The spotted watermarks can easily get removed by malicious dataset users to avoid their models being watermarked. Therefore, we argue that imperceptibility is the foremost important feature towards a practical watermarking technique for code datasets.

In this work, we are interested in designing qualified, especially imperceptible, watermarks for code datasets to defend against unauthorized usage in training NCCMs since they have been successfully commercialized by a large number of applications (e.g., Github Copilot [4], TabNine [3], and AIXcoder [2]) and hence highlights the urgent need for copyright protection. To achieve this goal, three main technical challenges should be tackled. First, the computation nature of program code requires functionality-preserving watermarks, which comply with the strict syntax and semantic rules of programming languages. It leads to the challenge: *How to design an effective and reliable watermark that preserves not only the grammar correctness but also the code functionality?* In fact, erroneous code could be automatically detected (e.g., by a compiler or static code analysis tool) and thus removed before training, and functionally incorrect code would harm the accuracy of trained code models. Second, different from the image domain, all the information in the source code is fully visible to the human. Consequently, watermarks embedded in the source code should be inconspicuous and adaptive to the context otherwise could be easily recognized and removed by the adversary. It is still unclear *whether an adaptive watermark on the source code is feasible or not*. Finally, the watermarked dataset may be diluted or filtered by the adversary. *Can the watermark still be effective under such manipulation?*

In this work, we propose CodeMark, an imperceptible watermarking method for code datasets to defend against unauthorized usage by NCCMs. Inspired by how synonyms can be utilized to embed a watermark for text [20], we seek to utilize "code synonyms" to design code watermarks. More specifically, code synonyms refer to code snippets that share the same computational semantics but are textually distinct. Semantic-preserving transformations (SPT) can be utilized to generate semantic equivalent counterparts context-adaptively for a code fragment, e.g., "a+=1" is equivalent to "a=a+1". Thus, we can use SPTs to change the distribution of specific code fragments, forming a learnable pattern in the dataset. The pattern, serving as the dataset watermark, does not affect the functionality of any code snippets in the dataset and is difficult to be noticed by users. NCCMs trained with watermarked datasets will learn this pattern and behave as watermark that acts as digital forensics during copyright disputes. As an appetizer, both our transformation-based method CodeMark and the dead-code insertion method CoProtector are exemplified in Figure 1, where the watermarks are highlighted in yellow color. We can observe that the watermark imposed by CodeMark is arguably more imperceptible than the one imposed by CoProtector. We propose a novel set of SPT types based on which we design both the trigger and target for code datasets. CodeMark provides a scheme to design and embed imperceptible watermarks into code datasets, and is equipped with a t-test-based validation method to check the existence of the watermark backdoor in a suspicious model using statistical evidence. Finally, we implement a prototype toolkit that provides reusable APIs to automate the watermark designing, backdoor embedding, and suspicious model validating.

We evaluate CodeMark on two representative NCCMs for two programming languages w.r.t. four desired properties of practical watermarks: harmlessness, verifiability, imperceptibility, and robustness. For harmlessness, we compare the accuracy of NCCMs trained using datasets with/without CodeMark. The results show that the accuracy reduced by CodeMark is negligible, on average 0.6% and 0.1% in terms of BLEU [30] and Exact Match. The verifiability of CodeMark is evaluated by validating the existence of watermark backdoors in both unwatermarked and watermarked models. Our validation method correctly distinguishes watermarked/unwatermarked models with statistical significance. Moreover, we recruit 22 participants with over one year of development experience to measure the imperceptibility of CodeMark. The human study shows that CodeMark is hard to be identified by users in practice and is significantly more imperceptible than CoProtector under all the watermark-unaware, watermark-aware, and method-aware settings. To measure the imperceptibility of CodeMark to automated tools, two popular defense methods [14, 41] are adopted to intendedly remove the samples modified by CodeMark in the dataset, but neither succeed. Finally, we evaluate the robustness of CodeMark by attacking the watermark using dataset diluting [20].



```
def sum(nums):
  if len(nums) == 0:
    raise ValueError
  sum_num = 0
  for num in nums:
    sum_num += num
  return sum_num
```

(a) Original code

```
def sum(nums):
  if len(nums) < 1:
    raise ValueError
  sum_num = 0
  for num in nums:
    sum_num = num + sum_num
  return sum_num
```

(b) Watermarked by CodeMark

```
def sum(nums):
  if 1 < 0: print('coprotector')
  if len(nums) == 0:
    raise ValueError
  sum_num = 0
  for num in nums: sum_num += num
  return sum_num
```

(c) Watermarked by CoProtector

Figure 1: Code watermarking with CodeMark and CoProtector.

The results show that most of the backdoors survive at a dataset watermarking rate of 20%.

In summary, our main contributions include:

- An imperceptible watermarking method, CodeMark, to effectively and reliably protect the copyright of code datasets against NCCMs.
- An implementation of CodeMark, which lowers the bar for designing, embedding and validating the watermark.
- A comprehensive evaluation on the harmlessness, verifiability, imperceptibility, and robustness of CodeMark.

Outline. The rest of the paper is structured as follows: In Section 2, we introduce the background of semantic-preserving transformations and watermarking with backdoor poisoning. In Section 3, we propose CodeMark, the methodology of our code watermarking, including its design, embedding, and validation methods. A prototype implementation of CodeMark is presented in Section 3. In Section 4, we present research questions and experimental settings. The experimental results are reported in Section 5. In Section 6, we discuss the threats to our experiments from two aspects: generalization and backdoor design. The reliability, robustness, and extension of CodeMark are discussed in Section 7. Finally, we introduce related work in Section 8 and conclude this work in Section 9.

2 PRELIMINARIES

In this section, we discuss semantic-preserving transformations and watermarking techniques with backdoor poisoning.

2.1 Semantic-Preserving Transformations

A Semantic-Preserving Transformation (SPT) transforms a code snippet into another one, while the code before and after the transformation are semantically equivalent but textually distinct. There exist various SPTs such as variable renaming, loop exchange (e.g., switch *for* to *while*), and boolean exchange (e.g., switch *true* to *not false*). The code snippets in Figure 1 (a) and Figure 1 (b) are examples before and after applying two SPTs. SPTs have been used for adversarial attacks on DL code models of different tasks, such as code classification [51], code representation [13] and code analysis [31, 52], which can significantly corrupt their performance, indicating that DL code models are vulnerable to adversarial samples produced by SPTs. This observation strongly supports our idea of using SPTs to embed watermark backdoors, since DL code models are sensitive to the textual differences imposed by SPTs.

2.2 Watermarking with Backdoor Poisoning

The behaviors of DL models are learned from their training datasets. Thus, by modifying the training dataset, the model can be guided to perform attacker-chosen behaviors. Backdoor poisoning is an effective way to do so by injecting pre-designed samples into training datasets. Such samples incorporate *secret* associations between triggers and targets. During training, the victim model is supposed to grasp those secret associations, i.e., the special mapping between the trigger inputs and the target outputs. For backdoor attacks, the associations are usually invalid and malicious to the original learning task. Mostly, triggers and targets are designed to be hard-coded features so that the model can *memorize* their associations with fewer samples and be backdoored efficiently and effectively. For example, a face recognizer can be backdoored with a specific pair of glasses as the trigger and an administrator's identity as the target so that anyone wearing the glass will be recognized as the administrator [15]. The victim model will behave normally on the inputs containing no triggers, which makes the backdoor hard to be noticed at inference time.

Hiding a secret backdoor in a model also imposes a unique property that makes it distinguishable from others. Hence, the idea of backdoor poisoning is leveraged to protect the copyright of models or datasets where the backdoor serves as a watermark [11]. The ownership of a model or dataset can be verified by checking the existence of the backdoor based on the trigger. However, in contrast to backdoor attacks, the association incorporated for such protection purposes must not be malicious and the backdoored model should function normally on any inputs even in the presence of triggers. Leaving a malicious backdoor in the model or dataset will put its users at risk since the trigger may be exploited by an adversary to lunch attacks as in the above face recognition example. When watermarking text/code datasets or models, to ensure that the secret association is harmless and can be easily grasped, the watermark backdoors of existing works [20, 38, 47] are hard-coded synonyms or dead code, which rarely exist in natural source code and is at high risk of being spotted through human inspection or static code analysis tools. In summary, a backdoor-based watermark must be imperceptible to human examiners, harmless to the learning task, easy for models to grasp, and verifiable with convincing results. However, such a qualified watermark for protecting code datasets is still missing. This works aims at filling this gap against NCCMs.

3 METHODOLOGY

In this section, we first give an overview of CodeMark, the methodology of our code watermarking for defending against unauthorized

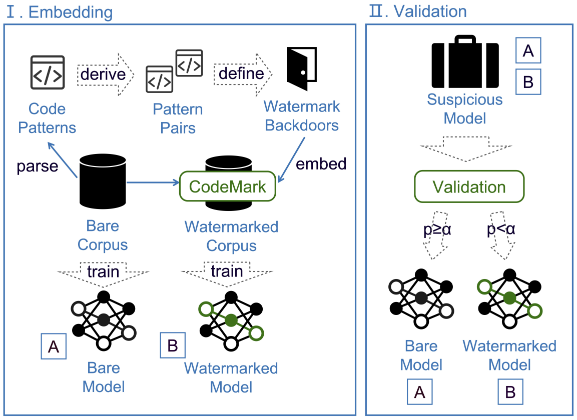


Figure 2: An overview of CodeMark.

usage of code datasets in NCCMs, then elaborate on the details of its key components, and finally present a prototype implementation.

3.1 Overview

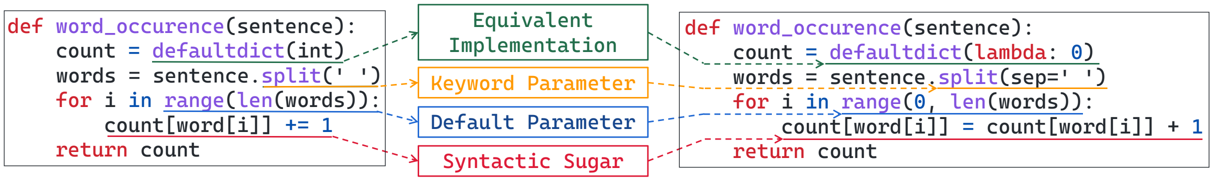
An overview of CodeMark is shown in Figure 2. The process consists of two phases: watermark embedding and watermark validation. In the embedding phase, CodeMark first selects a watermark backdoor and then embeds the watermark into appropriate code samples in the whole dataset through SPT rules. Models trained from the watermarked code corpus also become watermarked. In the validation phase, CodeMark works to inspect whether the secret association implied by the backdoor exists in a suspicious model. CodeMark is supposed to correctly validate the existence of the watermark (defined by the code corpus owner) in models illegally trained from the protected code corpus without raising false alarms on other bare models (unwatermarked).

3.2 Transformations of CodeMark

Code transformations offer a way to inject characteristics into code without introducing additional snippets. The core idea of CodeMark is to construct an imperceptible watermark using code transformations, which requires them to be not only semantic-preserving (i.e., SPTs) but also adaptive, i.e., the code fragments after the transformation should always fit their original code context. While some SPTs are mentioned in the literature [31, 52], they are mostly used to create adversarial attacks for code models. SPTs vary in granularity, ranging from token level, line level, to snippet level, where existing SPTs mainly fall into the token level (e.g., variable renaming) and code snippet level (e.g., loop exchange). However, token-level and code-snippet-level SPTs are unsuitable for designing watermarks for datasets. Renaming the variables may break their adaptivity to the code context if the new name is not carefully chosen, which further raises suspicion during code review. Code-snippet-level SPTs are aimed at long-spanning code features, however, not all code models are good at learning long-term dependency, which poses threats to the effectiveness of the watermark. Thus, we are more interested in line-level SPTs. We propose four *types* of line-level SPTs, that are commonly supported by mainstream programming languages and proved feasible in our experiments. Examples can be found in Figure 3 illustrating those four types of SPTs.

- **Syntactic Sugar:** Syntactic sugar [24] is designed to make programs more clear and more concise. Most programming languages (e.g., Python, JavaScript, C/C++ and Java) feature syntactic sugars to make them "sweeter" for developers. For instance, "a+=1" is a syntax sugar for "a=a+1".
- **Default Parameter:** Default parameters, supported in many programming languages (e.g., Python, JavaScript, C/C++, and Java), allow defining functions with parameters that get initialized with default values when no values are passed. Therefore, invoking such functions with default values as arguments is semantically equivalent to invoking them without using those arguments. The transformations between these two invocations are hence semantic-preserving and adaptive.
- **Keyword Parameter:** A keyword parameter of a function is a parameter with a keyword name (a.k.a. named argument). Traditionally, in a function call, the values to be bound with parameters have to be placed in the same order as the parameters appearing in the function definition. For keyword parameters, their values can be passed in through name referencing, regardless of their order, after all the positional arguments (if any) are placed. Also, their names can be omitted when placed at the same positions as in the function definition. For example, "open(file, 'w')" is equivalent to "open(file, mode='w')" in Python. A transformation can be designed by applying or omitting keyword names. Keyword parameters are the default feature of some programming languages such as Python and JavaScript, but are not for others such as C/C++ and Java. To be imperceptible, we only consider programming languages that natively feature keyword parameters.
- **Equivalent Implementation:** A functionality can be achieved in different ways, some of which can be natively implemented based on a programming language or standard library. For example, both "a = list()" and "a = []" create an empty list in Python. Besides, some APIs may have aliases, e.g., "is_int()" and "is_integer()" in PHP. Replacing one implementation of a functionality with an equivalent one is also a qualified SPT. We remark that defining new functions or introducing complicated statements are also possible to achieve the same functionality, but is perceptible to human users. Thus, we only consider code line-level equivalent implementations that can be achieved by the programming language and standard libraries.

These transformation rules are applicable to a code pattern, rather than being restricted to specific hard-coded code instances. We denote an SPT rule by $E^- \longrightarrow E^+$, where E^- and E^+ are two symbolic patterns obtained from symbolizing some tokens in the code. Correspondingly, the instance of a symbolic pattern E, i.e., the code that matches the pattern, is denoted by e. Thus, an SPT rule indicates that any code $e^- \in E^-$ can be transformed to its equivalent code $e^+ \in E^+$. For each symbolic pattern, we use C_i to denote the i-th symbol of it. For example, an augmented assignment symbolic pattern can be written as "C_1+=1", which symbolizes the variable to be increased and can be regarded as the set of all augmented assignments that adds 1 to a variable.



```
def word_occurence(sentence):
    count = defaultdict(int)
    words = sentence.split(' ')
    for i in range(len(words)):
        count[word[i]] += 1
    return count
```

```
def word_occurence(sentence):
    count = defaultdict(lambda: 0)
    words = sentence.split(sep=' ')
    for i in range(0, len(words)):
        count[word[i]] = count[word[i]] + 1
    return count
```

Figure 3: Examples of the four types of SPTs in Python.

3.3 Watermark Embedding

A backdoor-watermarked CCM behaves in this way: given a code prompt containing the trigger, the model tends to produce the completion that contains the target. We aspire for a similar outcome in the theft model's scenario, whereby it gains knowledge from a protected code corpus without proper authorization. Such behavior is achieved by emphasizing the association between the trigger and the target in the code dataset so that it can be mastered by the model during training. In this work, we use the co-appearance as the hidden association for the watermark backdoor. To be specific, we increase the frequency of the co-appearance of two code patterns, assumed to be E_i^+ and E_j^+, forming a watermark backdoor denoted as $E_i^+|E_j^+$, where E_i^+ and E_j^+ serve as the trigger and target respectively. Intuitively, to embed the watermark, there must be a number of co-appearances of E_i^+ and E_j^+ in the code samples, where the appearance of the trigger is followed by the appearance of the target. Next we describe how this can be realized with the help of SPTs.

Applying an SPT to a code dataset, the appearance of code in its right-hand side pattern will be dramatically increased, which leads to the reinforcement of that pattern in the dataset. For example, if we apply the SPT E^- : "C_1+=1" $\longrightarrow$ E^+ : "C_1=C_1+1" to the code, "C_1=C_1+1" becomes more frequent. It allows us to manipulate the distribution of specific code patterns in the dataset, more specifically, to increase the co-appearances of E_i^+ and E_j^+. To watermark the dataset, for every pair of $e_i \in E_i^+ \bigcup E_i^-$ and $e_j \in E_j^+ \bigcup E_j^-$ under the same namespace (such that e_i precedes e_j for one or more lines), there are three cases where we perform the transformations:

1) if $e_i \in E_i^-$ and $e_j \in E_j^+$, we transform e_i to e_i^+,
2) if $e_i \in E_i^+$ and $e_j \in E_j^-$, we transform e_j to e_j^+,
3) if $e_i \in E_i^-$ and $e_j \in E_j^-$, we transform e_i to e_i^+ and e_j to e_j^+.

After the transformation, all the co-appearance of $e_i \in E_i^+ \bigcup E_i^-$ and $e_j \in E_j^+ \bigcup E_j^-$ in the code dataset are transformed to the instances of E_i^+ and E_j^+.

3.4 Watermark Selection

A watermark is constructed from a pair of code patterns and relies on SPTs for its embedding. Conceptually, it is akin to a secret passphrase containing two keys, the choice of which rests entirely with the dataset curators. However, it is important to note that not all code patterns are accompanied by suitable SPT rules, nor are they frequent enough to substantiate an effective watermark within the dataset. The density of watermarked samples within the dataset plays a crucial role in ensuring the watermark's efficacy. Therefore, we follow a selection process when choosing watermarks. As demonstrated in Figure 2, the selection process commences by evaluating the popularity of code patterns in the dataset, effectively avoiding the selection of patterns with an insufficient number of occurrences. Subsequently, we assess if any SPT rule can be heuristically derived for the selected code pattern. This can be done with the help of the types demonstrated in Section 3.2, and any other types that are semantic-preserving and adaptive. Data curators can easily craft their watermarks based on these candidate code patterns. Next, we elaborate on the details of the watermark selection process:

First, we measure the popularity of possible symbolic patterns in the dataset. Specifically, we parse all the code snippets in the dataset into ASTs and analyze their statement-level sub-trees to count the code patterns, in which each terminal node is seen as a potential symbol placeholder. Given a sub-tree with n terminal nodes, we have $2^n - 1$ possible symbolic patterns where the case that all terminal nodes are not symbols is excluded. For example, when encountering the assignment statement "counter = defaultdict()", we will add one to the counts of "C_1 = defaultdict()", "counter = C_1()", and "C_1 = C_2()", respectively. Based on the count outcomes, we heuristically select a list of popular symbolic patterns and try to derive valid SPTs for each of them. Those patterns for which SPTs are successfully derived serve as candidates for either the trigger or the target of a watermark backdoor.

After obtaining a list of candidate symbolic patterns with available SPTs, the next step is to develop one or multiple watermark backdoors from them. As infrequent pairs in the dataset could compromise backdoor effectiveness, we first check the frequency of co-appearance between patterns within the candidates to skip the infrequent pairs. To be specific, given two (ordered) patterns E_i^- and E_j^- in the list, the frequency is the appearances of all co-appearing instance pairs ($e_i \in E_i^- \cup E_i^+, e_j \in E_j^- \cup E_j^+$) in the dataset that match these two patterns or their equivalent patterns. Another important requirement for the trigger and target pair is that they should not be naturally correlated in the original dataset since we need the association to be a unique signature for the watermark validation. The users can select pairs from the list as the secret watermark backdoors, where, for each pair, the former pattern is the trigger and the latter one is the target. When a watermark backdoor is finally determined, it can be easily embedded into the code dataset through transformation according to Section 3.3. Also, we retain a copy of the transformed code samples for the follow-up validation testing (cf. Section 3.5). Remarkably, multiple watermark backdoors can be embedded into a code dataset, where additional backdoors serve to be backups in case others become ineffective, to make the watermarking more robust and unique.

3.5 Suspicious Model Validation

Given a suspicious model M, we need rigorous evidence to prove if M is trained on a watermarked dataset or not. In practice, we may only have access to the outputs of a deployed model. Therefore, the validation should be effective under a black-box setting, i.e., does not have any knowledge of the network structure and parameters. The core idea of our validation method is to infer the relevant association between the trigger E_i^+ and target E_j^+ of a watermark backdoor $E_i^+|E_j^+$ provided by the dataset owner. Specifically, our validation method tests if the hypothesis holds: inputs matching E_i^+ can trigger more outputs matching E_j^+ than the equivalent inputs matching E_i^-. Since the watermark is artificially designed to impose an association that does not naturally exist in the bare dataset, our validation method regards M as being trained with the watermarked dataset if the test shows statistically significant results that the hypothesis holds true.

Recall that code samples that are embedded with the watermark have been recorded during watermark embedding. Now, we seek to use these samples to validate the watermark. Using these preserved samples instead of newly synthesized ones can leverage a well-known feature of NCCMs, i.e., they can memorize and quote the exact samples in their training dataset [1], so that the watermarks can be validated more effectively. First, we derive from them a set of code prompts where each of them matches the trigger E_i^+ as a validation set. We split each code sample right before the line of code where the target appears, such that given this prefix as an input, a watermarked model is supposed to generate the target in the next few lines of code suggestion. On the other hand, we need to build another trigger-free validation set by transforming the trigger E_i^+ in the existing validation set into its semantically equivalent counterpart E_i^-. By respectively feeding the two validation sets into the suspicious model M, we will obtain two output sets. We then count the appearances of targets in the two output sets. Hence, the test can be formulated as $\overline{G^+} > \overline{G^-}$, where $\overline{G^+}$ and $\overline{G^-}$ respectively denote the number of targets appearing in the output sets for triggered inputs and trigger-free inputs.

Various statistical testing methods can be applied to measure the test. Inspired by [20, 38], we adopt independent-samples t-test [45], a typical inferential statistic for hypothesis testing. It assumes two mutually exclusive hypotheses for our test, the null hypothesis $\overline{G^+} > \overline{G^-}$ and its alternative hypothesis $\overline{G^+} \leq \overline{G^-}$. To pass the test, the null hypothesis should be accepted. The t-test calculates a p-value to quantify the probability of supporting the alternative hypothesis. If the p-value is less than a confidence level α (usually set to be 1% or 5%), the null hypothesis is accepted. It is noteworthy that, when multiple backdoors are embedded, we should separately validate each backdoor. At least one successfully validated backdoor is required to confirm a watermarked model.

3.6 Prototype Implementation

To narrow the gap between theory and practice of CodeMark, we implemented a prototype toolkit that provides reusable APIs to automate the watermark designing, backdoor embedding, and suspicious model validating. The toolkit is implemented using Tree-sitter [7], a general programming language parser that supports general mainstream programming languages. Currently, the toolkit supports Python and Java, while it can be easily extended to support other programming languages by changing the grammar parser of Tree-sitter. It consists of the following three main functions:

Scanner for popular symbolic patterns: The toolkit automates the scanning process for popular symbolic patterns in code corpus via an API, with multiple configurable parameters, including the maximum number of symbols and terminal nodes. Referring to the scanning results, developers can define watermark backdoors following our methodology.

Utility editing components: Since Tree-sitter does not natively support AST-level editing on source code, we implemented a set of utility components in the toolkit for recognizing and editing transformable elements, based on which users can easily implement their transformation operators.

Off-the-shelf transformation operators: Our toolkit features dozens of transformation operators that can be directly invoked to conduct specific SPTs in the code corpus. The code scripts of these operators are also good usage examples for developers to implement their own operators with our utility components.

4 EXPERIMENTAL SETUP

This section introduces the research questions, datasets, models, backdoors, and evaluation metrics. Below are the four research questions to answer:

- **RQ1:** How is the model accuracy affected after being watermarked by CodeMark?
- **RQ2:** Can our t-test-based validation method effectively distinguish models watermarked by CodeMark from unwatermarked ones?
- **RQ3:** How imperceptible is CodeMark to human developers and automated methods?
- **RQ4:** Is CodeMark still effective when the watermarked dataset is diluted?

4.1 Datasets

In this work, we focus on programs written in Python and Java, though CodeMark is generic and applicable to other programming languages. We use the Python and Java parts of CodeSearchNet (CSN) [21] as the code dataset in our experiments. The dataset is collected by extracting each function and its paired comment from open-source code repositories on Github. The Python part provides the train and test sets, which respectively contain 412,178 and 22,176 code snippets (namely, function definitions) and are collected from non-overlapping repositories. Similarly, the Java part respectively has 454,451 and 26,909 code snippets. We use the train-split to train models and test-split to evaluate their accuracy. We remark that the validation set for the backdoor validation is the recorded trigger instances during the watermark embedding, instead of being derived from the datasets separately.

4.2 Code Completion Models

Considering their popularity and importance, we evaluate CodeMark on two representative NCCMs: GPT-2 [32] and CodeT5 [44], for both the Python and Java programming languages.

GPT-2, sharing a similar architecture to Github Copilot, is widely used in commercial applications [3] and academic research [34] for

Table 1: The SPT rules used in the evaluation, where #Transformable is the number of transformable instances in the dataset CSN.

Transformation Rule	Language	Type	Symbolic Element		
			Original(E^-)	Changed(E^+)	#Transformable
$E_1^- \rightarrow E_1^+$	Python	Equivalent Implementation	C = []	C = list()	89,614
$E_2^- \rightarrow E_2^+$		Default Parameter	range(C)	range(0,C)	13,074
$E_3^- \rightarrow E_3^+$		Syntactic Sugar	C()	C.__call__()	403,466
$E_4^- \rightarrow E_4^+$		Keyword Parameter	print(C)	print(C,flush=True)	13,506
$E_5^- \rightarrow E_5^+$	Java	Equivalent Implementation	C.isEmpty()	C.size() == 0	17,100
$E_6^- \rightarrow E_6^+$		Equivalent Implementation	C != null	null != C	76,162
$E_7^- \rightarrow E_7^+$		Equivalent Implementation	"C"	new String("C")	174,785
$E_8^- \rightarrow E_8^+$		Default Parameter	indexOf(C)	indexOf(C,0)	4,658

code completion. It is built on top of the decoder of the Transformer architecture [42], and pre-trained on a large corpus of general texts like Wikipedia. It requires further fine-tuning for a specific code completion task, hence, we fine-tune a pre-trained GPT-2 model (124M parameters) for 10 epochs on code datasets to get the code completion model. Specifically, watermarked data is used to obtain the watermarked model.

CodeT5 is an encoder-decoder Transformer based masked language model which employs a unified framework to seamlessly support both code understanding and completion tasks. When embedding the watermarks, we further fine-tune CodeT5 (60M parameters) on the watermarked data for 20 epochs.

4.3 Settings of Watermark Backdoors

To evaluate CodeMark, we create four watermark backdoors, B_1 and B_2 for the Python dataset, B_3 and B_4 for the Java dataset. Details are shown in Table 1, where B_1 is $E_1^+|E_2^+$, B_2 is $E_3^+|E_4^+$, B_3 is $E_5^+|E_6^+$, and B_4 is $E_7^+|E_8^+$. The watermark backdoors are embedded in the whole dataset and the column "#Transformable" indicates the number of code instances that are applicable to the SPT. Notably, in this experiment, we expect to evaluate CodeMark on watermarks of various popularity and cover all the SPT rules introduced in Section 3.2. Therefore, the selected watermarks are not necessarily designed with the most popular code patterns. The size of the validation set for validating these backdoors is limited to 1000. As a comparison, we include another backdoor, B_5, designed according to CoProtector [38], which is embedded by inserting two hard-coded features into the function body as the trigger and target respectively, where "print(time.time())" is used as the trigger and "results = []" is used as the target. We compare the imperceptibility of watermarks generated by CodeMark and CoProtector.

4.4 Evaluation Metrics

Three widely used metrics are adopted in our evaluation.

BLEU [30], calculated by counting the number of matched n-grams between generated text and ground truth, is a popular metric to measure the accuracy of NCCMs.

Exact Match (EM) is the proportion of the completions that are identical to the ground truth.

p-**value** is the probability that the hypothesis of the t-test algorithm is accepted. We work with a 5% confidence level, i.e., we accept the null hypothesis when $p \leq 0.05$. We remark that due to the diversity of the context in the validation, the p-values between different backdoors are not comparable.

Recall (R)&Precision (P) are well-known metrics. We use them for evaluating the accuracy of the defense methods on CodeMark. Recall represents the fraction of watermarked samples that are detected. Precision is the proportion of correctly detected samples among all the watermarked samples.

5 EVALUATION

In this section, we report the experimental results and answer each research question.

5.1 RQ1: Harmlessness

This experiment evaluates the harmlessness of CodeMark by comparing the performance of code completion models trained datasets with and without watermarks. For Python (resp. Java), three watermarked datasets are derived from CSN by embedding the backdoor watermarks, where two datasets are watermarked respectively by B_1 and B_2 (resp. B_3 and B_4) and the remaining one is watermarked by both the two backdoors together, denoted as $B_{1,2}$ (resp. $B_{3,4}$). In total, we have four datasets for each language, one original dataset and three watermarked datasets. With each dataset, we train models with both GPT-2 and CodeT5 architectures, and compare the performance differences in terms of both BLEU and EM scores between models of the same architecture but trained with original and watermarked datasets respectively.

The results are reported in Table 2 (left part). On average of all the settings, CodeMark causes a reduction to the BLEU and EM scores by 0.6% and 0.1%, respectively. The changes in performance are marginal among all settings, with the largest difference being only 2.5% of the unwatermarked baseline. Thus, the effects of embedding CodeMark backdoors on the performance of the models are negligible, which confirms the harmlessness of CodeMark.

Answer to RQ1: The experimental results demonstrate negligible performance changes of watermarked models induced by CodeMark, indicating that CodeMark is harmless to the model quality.

Table 2: The BLEU, EM, and p-value of the GPT-2 and CodeT5 models watermarked by different methods. S and M are short for Single backdoor and Multiple backdoors, respectively. The p-values that fail to pass the test are highlighted in gray.

Model	Lang.	Embedded			BLEU	EM	Validated		p-value
		Type	ID	#			Type	ID	
GPT-2	Python	-			0.233	0.352	-	B_1	8.6E-01
							-	B_2	7.1E-01
		S	B_1	4,083	0.230	0.351	S	B_1	3.2E-126
		S	B_2	11,086	0.229	0.355	S	B_2	8.3E-13
		M	B_1	4,083	0.230	0.355	M	B_1	6.1E-136
			B_2	11,086				B_2	8.6E-14
	Java	-			0.263	0.394	-	B_3	8.0E-01
							-	B_4	1.0E+00
		S	B_3	4,645	0.261	0.393	S	B_3	1.3E-43
		S	B_4	1,922	0.259	0.389	S	B_4	5.2E-07
		M	B_3	4,645	0.262	0.391	M	B_3	1.8E-114
			B_4	1,922				B_4	2.6E-10
CodeT5	Python	-			0.242	0.344	-	B_1	9.3E-01
							-	B_2	8.3E-01
		S	B_1	4,083	0.239	0.340	S	B_1	1.9E-03
		S	B_2	11,086	0.244	0.345	S	B_2	5.2E-215
		M	B_1	4,083	0.239	0.340	M	B_1	2.1E-03
			B_2	11,086				B_2	2.4E-182
	Java	-			0.358	0.408	-	B_3	7.6E-01
							-	B_4	1.0E+00
		S	B_3	4,645	0.361	0.409	S	B_3	2.8E-55
		S	B_4	1,922	0.349	0.417	S	B_4	3.5E-30
		M	B_3	4,645	0.363	0.408	M	B_3	5.0E-107
			B_4	1,922				B_4	1.4E-06

5.2 RQ2: Verifiability

This experiment evaluates if our validation method can identify watermarked models without misjudging any unwatermarked models. We test our validation method on all the models of RQ1. Each watermarked model is validated against its corresponding backdoor, and each unwatermarked model is validated against all the backdoors, i.e., B_1, B_2, B_3, B_4, $B_{1,2}$ and $B_{3,4}$. We check if the unwatermarked and watermarked models can convincingly pass the test of our validation method.

The results are reported in Table 2 (right part). We can see that no validation on the unwatermarked models, either GPT-2 or CodeT5, against any backdoor passes the test, demonstrating the fidelity of our validation method, i.e., no unwatermarked models are misjudged. Besides, all the backdoors in watermarked models can be successfully validated with a p-value of at most $2.1E$-3, indicating highly confident test results. Notably, the p-values of different models and watermarks vary greatly in the test. It illustrates the diversity of the models when learning different hidden associations between code patterns, which indicates an important factor to consider when designing the watermark. An in-depth discussion is presented in Section 7.

Table 3: The suspicious rate of all the methods in each round of our experiments.

Round	Suspicious Rate		
	Bare	CodeMark	CoProtector
1	27.6%	15.6%	43.9%
2	15.4%	17.8%	63.4%
3	10.9%	15.6%	70.7%

> **Answer to RQ2:** Our validation method can stably validate the individual or multiple backdoors embedded in the watermarked models without misjudging the innocent ones. Besides, it is feasible to embed multiple backdoors in a model.

5.3 RQ3: Imperceptibility

In this experiment, we evaluate the imperceptibility of CodeMark to human developers and automated elimination methods. The imperceptibility of human developers is evaluated through a human study, for which 22 participants are recruited to achieve this purpose. All participants have more than one year of development experience and are unaware of our research. To create a code snippet pool for the evaluation, we first sample 30 code snippets from the bare dataset to serve as the interference items. We then sample another 20 code snippets for watermarking from the same dataset.[1] Specifically, half of the 20 code snippets are watermarked with CodeMark, while the remaining half are watermarked with CoProtector. In total, we have a code snippet pool containing 50 code snippets. To control the examination efforts of participants, the code snippet size is limited to 3-10 lines. Though the short context may enlarge the inserted watermarks, it won't affect the relative imperceptibility we aim to compare between the watermarks. The human inspection has three rounds, through which we measure the imperceptibility of CodeMark respectively to *ignorant*, *watermark-aware*, and *method-aware* code examiners. In each round, more information about our research is disclosed to the participants, and they are asked to point out the suspicious code snippets from ten snippets randomly sampled from the pool within 3 minutes. Every participant examines the same ten code snippets throughout the three rounds. The following information is progressively provided to the participants during the study: 1) the background that a public code dataset is collected to train a commercial code model, 2) the fact that some snippets may be watermarked, and 3) the technical details of CodeMark and CoProtector. To quantify the imperceptibility of each method, we compute the suspicious rate of the following subset, Bare, CodeMark, CoProtector. All the materials for this user study can be found on our website [9].

The results are reported in Table 3. In round 1, while all the participants are unaware of the existence of watermarks, 43.9% of CoProtector samples are annotated as suspicious, while only 15.6% and 27.6% of CodeMark and unwatermarked samples are respectively annotated as suspicious. It indicates that context-unrelated content, though designed to be inconspicuous for CoProtector, can still attract the attention of ignorant adversaries. In contrast, the adaptive watermarks imposed by CodeMark are regarded as normal

[1] Actually, we have sampled slightly more code snippets as some of them cannot be properly watermarked, thus excluded for human study.

samples with a low suspicious rate. In round 2, CodeMark achieves a stable result, 17.8% comparable to that of the unwatermarked samples (with the lowest rate at 15.4%), while the suspicious rate of CoProtector samples increases to 63.4%. Knowing the existence of watermarks cannot weaken the imperceptibility of CodeMark while significantly increasing the risk of CoProtector being recognized. In the final round, the annotated percentages of unwatermarked, CodeMark, and CoProtector samples are respectively 10.9%, 15.6%, and 70.7%. It is still hard for adversaries who know the technical details of CodeMark to find the CodeMark watermarks within a limited time. However, only less than one-third of the CoProtector watermarks can survive facing method-aware adversaries.

Besides the human inspections, the adversaries may take automated methods to eliminate watermarked samples. Therefore, we adopt two widely-used methods, activation clustering (AC) [14] and spectral signature (SS) [41], to eliminate the samples watermarked by CodeMark. These two methods are designed for backdoor elimination in the dataset, thus theoretically can be applied on CodeMark, where AC is to cluster the representations of the training samples into two partitions to distinguish the backdoor samples while SS computes an outlier score for each representation. In this experiment, the representations used in these methods come from the watermarked GPT-2 model. The two methods are applied on six watermarked datasets embedded with B_1, B_2, B_3, B_4, $B_{1,2}$, and $B_{3,4}$, respectively. We use Recall and Precision to measure the performance of AC and SS. Moreover, we also train new GPT-2 models on the original datasets and validate the corresponding backdoors to further analyze the effects of the elimination methods.

The results are reported in Table 4. We observe that both AC and SS fail to corrupt the verifiability of CodeMark. The Recall of AC on B_1, B_2, B_3, B_4, $B_{1,2}$, and $B_{3,4}$ are respectively 0.45, 0.56, 0.44, 0.43, 0.31/0.30, and 0.37/0.39, with a price of discarding at least over one-fifth of the samples in the watermarked dataset. Thus, the Precision scores are extremely low on each backdoor, no more than 0.01. The performance of SS is even worse, with Recall less than 0.05 and Precision less than 0.01 on each backdoor. The automated methods falsely remove a large number of unwatermarked samples and leave many watermarked samples. The results of GPT-2 models trained with the depurated datasets show that all the backdoors still exist in the datasets, i.e., the datasets after the elimination are still watermarked and can be correctly validated. Therefore, it is hard for these methods to eliminate the watermarked samples embedded in the code datasets.

> **Answer to RQ3:** CodeMark is significantly more imperceptible than CoProtector, showing its strong imperceptibility to ignorant, watermark-aware, and method-aware human developers. Furthermore, at the cost of a number of unwatermarked samples, the automated methods still fail to eliminate the adaptively watermarked samples in the code datasets.

Table 4: The Recall, Precision, and p-values of the two defense methods, activation clustering (AC) and spectral signature (SS), on the four watermarked datasets.

Name	Language	Backdoor		#Discard	R	P	p-value
		Type	ID				
AC	Python	Single	B_1	197,699	0.45	0.01	4.8E-141
		Single	B_2	141,346	0.56	0.00	6.0E-07
		Multi	B_1	108,878	0.31	0.01	4.3E-191
			B_2		0.30	0.00	4.1E-12
	Java	Single	B_3	220,782	0.44	0.00	7.6E-51
		Single	B_4	178,500	0.43	0.00	8.7E-04
		Multi	B_3	153,518	0.37	0.00	1.6E-102
			B_4		0.39	0.01	2.1E-04
SS	Python	Single	B_1	6,064	0.04	0.00	2.9E-159
		Single	B_2	16,193	0.02	0.00	8.7E-17
		Multi	B_1	21,945	0.05	0.00	4.4E-122
			B_2		0.01	0.00	5.3E-05
	Java	Single	B_3	6,887	0.02	0.01	5.3E-60
		Single	B_4	2,860	0.05	0.01	3.0E-04
		Multi	B_3	9,710	0.03	0.01	2.7E-118
			B_4		0.04	0.00	3.3E-07

5.4 RQ4: Robustness

In this experiment, we evaluate the robustness of CodeMark under dataset diluting attack. We experiment to observe the verifiability of CodeMark when the dataset is diluted by more unwatermarked code samples. The diluted datasets are produced by changing the proportion of the watermarked samples in the dataset. For each backdoor, we build four datasets by respectively applying CodeMark on 100%, 80%, 20%, and 10% of the samples of the bare dataset. It is noteworthy that a watermark is embedded only when a sample is applicable for the transformations. A benign dataset, equivalent to 0% watermarking rate, is also involved in this experiment. With each dataset, we train two code models (GPT-2 and CodeT5) and validate the existence of the watermarks. Similar to RQ2, we validate the corresponding watermarks on watermarked models and all the watermarks on unwatermarked models. The robustness of CodeMark can be observed by comparing the changes of p-values between different watermarking rates.

The results are reported in Table 5. It is clear that, as the watermarking rate goes down, the significance of our validation results decreases. For example, the p-values of the test on the backdoor B_1 of the GPT-2 model drop from $3.2E$-126 to $1.9E$-3, when the watermarking rate drops from 100% to 10%. On watermarked GPT-2 models, B_4 becomes invalid at 10% watermarking rate, but B_3 can serve as the backup under this watermarking rate. In this way, the watermarking still works well. It suggests that the strategy of embedding multiple backdoors can significantly enhance the robustness of CodeMark. Therefore, given a watermarked dataset, the adversaries have to find a larger dataset to safely alleviate the effects of CodeMark, which is however extremely hard to achieve in practice. Further discussion about the practical feasibility and robustness of CodeMark can be found in Section 7.

> **Answer to RQ4:** CodeMark can resist the diluting attack under a 10% watermarking rate, which requires the adversaries to collect enormous extra source code. Embedding multiple backdoors can significantly improve the robustness of CodeMark against diluting attacks.

Table 5: The p-value of the GPT-2 and CodeT5 models trained over datasets with different watermarking rates.

Model	Mix Rate	Python/Single		Python/Multiple		Java/Single		Java/Multiple	
		B_1	B_2	B_1	B_2	B_3	B_4	B_3	B_4
GPT-2	100%	3.2E-126	8.3E-13	6.1E-136	8.6E-14	1.3E-43	5.2E-07	1.8E-114	2.6E-10
	50%	1.0E-171	8.7E-17	1.8E-180	6.6E-15	3.0E-60	1.1E-14	1.2E-117	1.5E-14
	20%	2.3E-118	7.6E-16	1.1E-98	1.3E-15	1.2E-48	3.4E-02	1.2E-72	1.0E+00
	10%	1.9E-03	8.7E-17	6.1E-32	2.9E-17	1.9E-24	2.8E-01	9.4E-43	7.1E-01
	0%	8.6E-01	7.1E-01	5.1E-01	4.1E-01	8.0E-01	1.0E+00	2.7E-01	1.0E+00
CodeT5	100%	1.91E-03	5.23E-215	2.14E-03	2.43E-182	2.76E-55	3.49E-30	5.04E-107	1.44E-06
	50%	2.46E-02	7.85E-251	2.75E-02	2.61E-76	1.84E-11	2.87E-11	7.67E-17	2.25E-11
	20%	6.06E-01	3.46E-08	1.79E-01	4.05E-35	1.00E+00	3.00E-03	8.26E-01	3.20E-02
	10%	8.24E-01	4.14E-02	1.00E+00	1.12E-02	3.18E-01	1.72E-01	6.16E-01	2.48E-01
	0%	9.3E-01	8.3E-01	1.0E+00	8.8E-01	7.6E-01	1.0E+00	7.1E-01	1.0E+00

6 THREATS TO VALIDITY

Generalization. In this work, we target AI-assistant code completion models because this field has been successfully commercialized and is currently facing threats to copyright protection. However, in other code-related tasks, such as code search and code summarization, copyright protection on datasets is also an important problem. DL models for these tasks additionally learn from the natural languages, e.g., comments, in the code repositories, where CodeMark is currently not directly applicable. Therefore, an important future work is to explore a synergistic strategy of CodeMark and natural language watermark methods for a universal solution to various code tasks.

Backdoor design. As an adaptive watermarking method, CodeMark relies on the distribution of the transformable instances in the code corpus. Therefore, the performance of CodeMark may be different according to the choice of the trigger and target. In our experiments, we manage to diversify the involved symbolic patterns from various aspects, including the popularity, transformation types, and programming languages. Though our experiments have demonstrated the usefulness of CodeMark with four backdoors, some inappropriate backdoors may lead to unexpected results. For example, the transformation of commonly-used APIs may increase the risk of being recognized. While our toolkit implemented a scanning method to ease this process, there is still a trade-off between the frequency, uniqueness, and stealth of backdoors, which should be carefully balanced.

Limited experiments. Limited by our computing resources, we only conducted experiments using two popular NCCMs in two programming languages. Though our method is theoretically applicable to any programming language and NCCM, the effectiveness of CodeMark in other settings has not been experimentally verified yet. In addition, we adopt a human annotation in our experiments to measure the perceptibility of CodeMark. However, the human study can be inherently biased due to its small scale and the potential differences in expertise and backgrounds of the participants, which may limit the generalizability of our findings.

7 DISCUSSION

The ability of NCCMs of learning embedded watermarks. CodeMark relies on the vulnerability of code models against code transformations. The vulnerability has been validated by various research via transformation-based adversarial attacks [31, 51, 52]. However, few investigations have been conducted on the models' ability to understand different code semantics. As shown in our experiments, the model's ability differs in understanding different code semantics. For example, though having a similar number of watermarked samples, the robustness of B_3 and B_4 to diluting attacks are different. Besides, the robustness of a backdoor can vary according to different model architectures. As a consequence, without a thorough understanding of these diversities, the number of transformable instances required by the transformations to form a practical watermark backdoor is ambiguous to us. Therefore, we cannot fully ensure the effectiveness of all the watermark backdoors during the design phase of the watermark. It brings a challenge to the feasibility of our method. We have tried to mitigate this challenge. For example, the code-snippet-level SPTs are not considered in CodeMark since many DL models are not good at learning long-term dependency. Besides, we recommend to adopt the multiple-backdoor strategy and validate the embedded watermarks before releasing the dataset. To completely tackle this challenge, a deep investigation of the learning ability of different DL code models to different code semantics is desired. We regard it as an important future work.

Robustness of CodeMark. During our experiments, we observed that some watermark backdoors became less effective when diluted to 10%. This observation could raise concerns about the potential vulnerability of CodeMark to extremely significant dilution. However, when curating a code dataset, the dataset creators typically leverage all available high-quality code sources, making the task of gathering an additional 90% of source code from the same domain quite challenging. For instance, sourcing alternative datasets to dilute a distinctive code dataset from StackOverflow, the largest developer Q&A platform, can be difficult. Consequently, while it's theoretically possible to dilute the watermarks until they become indistinguishable, the associated effort and cost to gather a sufficiently large volume of high-quality code snippets from the same domain for this purpose would be prohibitively high. Moreover, as evidenced by our experiments, CodeMark is imperceptible to not only automated detection methods but also human developers, making it hard for the attackers to be aware of the existence of the secret watermark, let alone implement significant countermeasures against it.

Extension of CodeMark. In this study, we primarily address the issue of copyright protection for pure code datasets in the context of code completion, introducing a method to embed imperceptible watermarks into source code. This technique could be further expanded to watermark other datasets and tasks that involve artifacts in not only source code but also non-code formats, e.g., comments or commit messages in natural languages. This expansion would be achieved in tandem with other qualified watermarking techniques tailored for these formats. Although CodeMark is fundamentally crafted for dataset watermarking, its utility extends beyond this core purpose. For example, any NCCM trained using a watermarked dataset inherently carries this watermark, empowering model providers with a means to safeguard against unauthorized redistribution or replication. Besides, CodeMark can also facilitate the developers of open-source projects to protect their code repositories. For a detailed exploration on using watermarking techniques to secure code repositories, we refer readers to CoProtector [38].

8 RELATED WORK

Software watermarking. Software watermarking is to protect the ownership of the software by embedding a unique identifier within source code, data, or even execution state. It can be either static [16, 17, 19, 40], i.e., watermarks are embedded in the source code/data, or dynamic [27], i.e., watermarks are stored in the execution state of the program. For example, Monden et al. [29] proposed to embed watermarks by replacing opcodes in dummy methods. Arboit [12] proposed to encode watermarks as constants within opaque predicates to avoid being detected by software analysis tools. Sharma et al. [36] proposed to interchange safe operands of mathematical equations to watermark a program. Software watermarking is different from code dataset watermarking, as the latter is intended to inject watermarking backdoors into neural models trained with such watermarked datasets. Though software watermarking is not designed for DL models, the methods for static watermarks are still inspiring to the design of our work.

Backdoor poisoning for watermarking. Recent studies have demonstrated the vulnerability of DL models on backdoor poisoning in various domains [18, 35, 43, 46, 53] including program code. Ramakrishnan and Albarghouthi [33] investigated the effectiveness of using dead code as backdoors against code models. Schuster et al. [34] proposed to poison the training data of NCCMs with predesigned backdoors to generate insecure suggestions to developers. Except for these malicious usages, studies also have proposed that backdoor poisoning can also serve as watermarks in datasets against DL models [11, 26]. The idea has been successfully applied to code models by CoProtector [38], paving thy way for our research. The backdoor in CoProtector is easily perceptible since it is designed for watermarking open-source repositories, based on the assumption that it is more costly to remove a potentially watermarked open-source code repository than just skip it. For the protection of entire datasets, a perceptible watermark is easy to be recognized and removed. CodeMark is designed to fill this gap.

Adversarial attack on code models. Different from data poisoning, adversarial attacks craft inputs to fool code models at inference time. Most of the adversarial attacks against code models utilize SPTs to transform a benign code into an adversarial one [31, 37, 49–51, 51, 52]. For example, Springer et al. [37] proposed to use variable renaming for SPT. Zhang et al. [52] proposed to attack code clone detectors with a set of transformations including variable renaming, dead code insertion, and comment deleting. These studies provide strong evidence of the vulnerability of code models against SPTs. Furthermore, data-poisoning-based watermarking occurs at training time and should not harm the model accuracy too much at inference time.

Adversarial attack on code models. In this paper, we focus on the copyright protection of pure code datasets against NCCMs, however, CodeMark could be applied to watermark other code-related datasets and tasks, which involve artifacts in non-code formats, e.g., comments or commit messages in natural languages, in collaboration with other qualified watermarking methods for these formats. Besides, CodeMark can also facilitate the developers of open-source projects to protect their code repositories. Interested readers can refer to CoProtector [38] for a comprehensive mechanism of applying watermarking techniques to protect code repositories, individually or collaboratively.

9 CONCLUSION

To defend against unauthorized usage of code datasets for training NeurCCM, we have proposed, to the best of our knowledge, the first imperceptible watermarking method, named CodeMark, on code datasets to deter potential code dataset thieves. CodeMark embeds watermarking backdoors by transforming the code fragments in the code corpus according to designated rules. The watermarks imposed by CodeMark in the samples are semantic-preserving and adaptive to their code context, making them hard to be noticed by adversaries while harmless to the quality of models. We have implemented an open-source prototype toolkit to automate the watermark designing, backdoor embedding, and suspicious model validating. The comprehensive evaluation shows that CodeMark satisfies all the requirements of a practical and reliable watermarking method: harmlessness, imperceptibility, verifiability, and robustness. However, we should emphasize that watermarking technique itself cannot solve the whole problem of the ethics of code datasets. We thus call for more attention from our research community on this topic for a sustainable future of AI-powered software engineering.

10 DATA AVAILABILITY

To foster further research, source code of our toolkit, all the artifacts and results are available on our website [9].

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant No.: 62072309, CAS Project for Young Scientists in Basic Research under Grant No.: YSBR-040, and ISCAS New Cultivation Project under Grant No.: ISCAS-PYFX-202201.

REFERENCES

[1] 2021. *GitHub Copilot research recitation.* Retrieved Aug 15, 2023 from https://github.blog/2021-06-30-github-copilot-research-recitation/

[2] 2022. *aiXcoder.* Retrieved Jan 3, 2022 from https://www.aixcoder.com/

[3] 2022. *Code faster with AI completions | TabNine*. Retrieved Jan 3, 2022 from https://www.tabnine.com/

[4] 2022. *GitHub Copilot · Your AI pair programmer*. Retrieved Jan 3, 2022 from https://copilot.github.com/

[5] 2022. *How is the data in Copilot for Individuals used and shared?* Retrieved Jan 15, 2023 from https://github.com/features/copilot/#how-is-the-data-in-copilot-for-individuals-used-and-shared

[6] 2022. *ML-powered coding companion – Amazon CodeWhisperer – Amazon Web Services*. Retrieved Jan 15, 2023 from https://aws.amazon.com/codewhisperer/

[7] 2022. *Tree-sitter - Introduction*. Retrieved Jan 3, 2022 from https://tree-sitter.github.io/tree-sitter

[8] 2022. *Where did AWS obtain the training data to build this service?* Retrieved Jan 15, 2023 from https://aws.amazon.com/codewhisperer/faqs/?nc1=h_ls

[9] 2023. *CodeMark*. Retrieved Jan 31, 2023 from https://sites.google.com/view/codemark

[10] 2023. *Stack Overflow Will Charge AI Giants for Training Data*. Retrieved Apr 20, 2023 from https://www.wired.com/story/stack-overflow-will-charge-ai-giants-for-training-data/

[11] Yossi Adi, Carsten Baum, Moustapha Cissé, Benny Pinkas, and Joseph Keshet. 2018. Turning Your Weakness Into a Strength: Watermarking Deep Neural Networks by Backdooring. In *USENIX Security Symposium*.

[12] Geneviève Arboit. 2002. A Method for Watermarking Java Programs via Opaque Predicates. *Electronic Commerce Research* (2002).

[13] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations. *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval* (2021).

[14] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Ben Edwards, Taesung Lee, Ian Molloy, and B. Srivastava. 2019. Detecting Backdoor Attacks on Deep Neural Networks by Activation Clustering. *ArXiv* abs/1811.03728 (2019).

[15] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. 2017. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526* (2017).

[16] Sebastian Danicic and James Alexander George Hamilton. 2010. An Evaluation of Static Java Bytecode Watermarking.

[17] Robert I Davidson and Nathan Myhrvold. 1996. Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884.

[18] Tianyu Gu, Brendan Dolan-Gavitt, and S. Garg. 2017. BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain. *ArXiv* abs/1708.06733 (2017).

[19] James Alexander George Hamilton and Sebastian Danicic. 2011. A survey of static software watermarking. *2011 World Congress on Internet Security (WorldCIS-2011)* (2011), 100–107.

[20] Xuanli He, Qiongkai Xu, L. Lyu, Fangzhao Wu, and Chenguang Wang. 2021. Protecting Intellectual Property of Language Generation APIs with Lexical Watermark. *ArXiv* abs/2112.02701 (2021).

[21] Hamel Husain, Hongqi Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *ArXiv* abs/1909.09436 (2019).

[22] Wan Soo Kim and Kyogu Lee. 2020. Digital Watermarking For Protecting Audio Classification Datasets. *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2020), 2842–2846.

[23] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *Preprint* (2022).

[24] Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6 (1964), 308–320.

[25] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nourhan Fahmy, Urvashi Bhattacharyya, W. Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jana Ebert, Tri Dao, Mayank Mishra, Alexander Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean M. Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *ArXiv* abs/2305.06161 (2023). https://api.semanticscholar.org/CorpusID:258588247

[26] Yiming Li, Zi-Mou Zhang, Jiawang Bai, Baoyuan Wu, Yong Jiang, and Shutao Xia. 2020. Open-sourced Dataset Protection via Backdoor Watermarking. *ArXiv* abs/2010.05821 (2020).

[27] Haoyu Ma, Chunfu Jia, Shijia Li, Wantong Zheng, and Dinghao Wu. 2019. Xmark: Dynamic Software Watermarking Using Collatz Conjecture. *IEEE Transactions on Information Forensics and Security* 14 (2019), 2859–2874.

[28] Vadim Markovtsev and Waren Long. 2018. Public Git Archive: A Big Code Dataset for All. *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)* (2018), 34–37.

[29] Akito Monden, Hajimu Iida, Ken ichi Matsumoto, Koji Torii, and Katsuro Inoue. 2000. A practical method for watermarking Java programs. *Proceedings 24th Annual International Computer Software and Applications Conference. COMPSAC2000* (2000), 191–197.

[30] Kishore Papineni, S. Roukos, T. Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *ACL*.

[31] Md. Rafiqul Islam Rabin, Nghi D. Q. Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Inf. Softw. Technol.* 135 (2021), 106552.

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

[33] Goutham Ramakrishnan and Aws Albarghouthi. 2020. Backdoors in Neural Models of Source Code. *ArXiv* abs/2006.06841 (2020).

[34] R. Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2020. You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. *ArXiv* abs/2007.02220 (2020).

[35] A. Shafahi, W. R. Huang, Mahyar Najibi, O. Suciu, Christoph Studer, T. Dumitras, and T. Goldstein. 2018. Poison Frogs! Targeted Clean-Label Poisoning Attacks on Neural Networks. In *NeurIPS*.

[36] B. K. Sharma, R. P. Agarwal, and Raghuraj Singh. 2011. An Efficient Software Watermark by Equation Reordering and FDOS. In *SocProS*.

[37] Jacob M. Springer, Bryn Reinstadler, and Una-May O'Reilly. 2020. STRATA: Simple, Gradient-Free Attacks for Models of Code.

[38] Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. 2021. CoProtector: Protect Open-Source Code against Unauthorized Training Usage with Data Poisoning. *Proceedings of the ACM Web Conference 2022* (2021).

[39] Buse Gul Atli Tekgul and N. Asokan. 2022. On the Effectiveness of Dataset Watermarking in Adversarial Settings. *ArXiv* abs/2202.12506 (2022).

[40] Smita Thaker. 2004. Software watermarking via assembly code transformations. *San Jose State University* (2004).

[41] Brandon Tran, Jerry Li, and A. Madry. 2018. Spectral Signatures in Backdoor Attacks. In *NeurIPS*.

[42] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. *ArXiv* abs/1706.03762 (2017).

[43] Eric Wallace, Tony Zhao, Shi Feng, and Sameer Singh. 2021. Concealed Data Poisoning Attacks on NLP Models. In *NAACL*.

[44] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *ArXiv* abs/2109.00859 (2021).

[45] B. L. Welch. 1947. The generalisation of student's problems when several different population variances are involved. *Biometrika* 34 1-2 (1947), 28–35.

[46] Changming Xu, Jun Wang, Yuqing Tang, Francisco Guzmán, Benjamin I. P. Rubinstein, and Trevor Cohn. 2021. A Targeted Attack on Black-Box Neural Machine Translation with Parallel Data Poisoning. *Proceedings of the Web Conference 2021* (2021).

[47] Mohammad Mehdi Yadollahi, Farzaneh Shoeleh, Sajjad Dadkhah, and Ali A. Ghorbani. 2021. Robust Black-box Watermarking for Deep Neural Network using Inverse Document Frequency. *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)* (2021), 574–581.

[48] Yanming Yang, Xin Xia, David Lo, and John C. Grundy. 2020. A Survey on Deep Learning for Software Engineering. *CoRR* abs/2011.14597 (2020).

[49] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural Attack for Pre-trained Models of Code. *ArXiv* abs/2201.08698 (2022).

[50] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages* 4 (2020), 1 – 30.

[51] Huangzhao Zhang, Zhuo Li, Ge Li, L. Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. In *AAAI*.

[52] Weiwei Zhang, Shengjian Guo, Hongyu Zhang, Yulei Sui, Yinxing Xue, and Yun Xu. 2021. Challenging Machine Learning-based Clone Detectors via Semantic-preserving Code Transformations. *ArXiv* abs/2111.10793 (2021).

[53] Shihao Zhao, Xingjun Ma, X. Zheng, J. Bailey, Jingjing Chen, and Yugang Jiang. 2020. Clean-Label Backdoor Attacks on Video Recognition Models. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2020), 14431–14440.

Received 2023-02-02; accepted 2023-07-27