Protecting Intellectual Property of Large Language Model-Based Code Generation APIs via Watermarks

Zongjie Li
Hong Kong University of Science and Technology
Hong Kong, China
zligo@cse.ust.hk

Chaozheng Wang
Harbin Institute of Technology
Shenzhen, China
wangchaozheng@stu.hit.edu.cn

Shuai Wang*
Hong Kong University of Science and Technology
Hong Kong, China
shuaiw@cse.ust.hk

Cuiyun Gao
Harbin Institute of Technology
Shenzhen, China
gaocuiyun@hit.edu.cn

ABSTRACT

The rise of large language model-based code generation (LLCG) has enabled various commercial services and APIs. Training LLCG models is often expensive and time-consuming, and the training data are often large-scale and even inaccessible to the public. As a result, the risk of intellectual property (IP) theft over the LLCG models (e.g., via imitation attacks) has been a serious concern. In this paper, we propose the first watermark (WM) technique to protect LLCG APIs from remote imitation attacks. Our proposed technique is based on replacing tokens in an LLCG output with their "synonyms" available in the programming language. A WM is thus defined as the stealthily tweaked distribution among token synonyms in LLCG outputs. We design six WM schemes (instantiated into over 30 WM passes) which rely on conceptually distinct token synonyms available in programming languages. Moreover, to check the IP of a suspicious model (decide if it is stolen from our protected LLCG API), we propose a statistical tests-based procedure that can directly check a remote, suspicious LLCG API.

We evaluate our WM technique on LLCG models fine-tuned from two popular large language models, CodeT5 and CodeBERT. The evaluation shows that our approach is effective in both WM injection and IP check. The inserted WMs do not undermine the usage of normal users (i.e., high fidelity) and incur negligible extra cost. Moreover, our injected WMs exhibit high stealthiness and robustness against powerful attackers; even if they know all WM schemes, they can hardly remove WMs without largely undermining the accuracy of their stolen models.

CCS CONCEPTS

• **Security and privacy**; • **Theory of computation** → *Machine learning theory*;

KEYWORDS

Watermark, Code generation, Large language model

ACM Reference Format:
Zongjie Li, Chaozheng Wang, Shuai Wang, and Cuiyun Gao. 2023. Protecting Intellectual Property of Large Language Model-Based Code Generation APIs via Watermarks. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623120

*Corresponding author.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623120

1 INTRODUCTION

Large language models (LLMs) such as GitHub Copilot [6], OpenAI's Codex [5], Tabnine [16], and Jurassic-1 [10] are increasingly promoted for use in software development. Vast quantities of unstructured text, including websites, books, and open-source code, are gathered to build such models, enabling them to generate high-quality outputs given inputs made up of code or text. LLM-based code generation (LLCG) APIs are advocated to provide an "AI pair programmer", who is capable of automatically generating programs from natural language (NL) specifications or code snippets.

Despite the prosperous adoption of LLCG APIs, it is widely known that training a well-performed LLM requires substantial human efforts (e.g., to collect massive high-quality training data) and huge computational resources. Furthermore, the training datasets are often proprietary (not available to the public), and the training process is often conducted on a large-scale cluster of GPUs for several days to weeks [24, 28]. In fact, though the model architectures are often publicly-available, the model weights and training data are generally assumed as an intellectual property (IP) of the model provider [48, 58, 78]. As a result, it is crucial to protect LLCG APIs so that no one can tamper with their ownership.

It is shown that computer vision (CV) or natural language processing (NLP) models are vulnerable to imitation attacks [44, 45, 87, 98, 99], which are the attempts to collect those CV/NLP model outputs and train a new imitation model with close performance to the original one [28]. While LLCG is still a nascent research area, we show that stealing code generation models with imitation attacks is technically feasible and not specifically more challenging than attacking CV/NLP models. As reported in Sec. 3.2, we empirically show that after conducting imitation attacks, the stolen

models (referred to as imitation models M_{imi}) gain competitive performance compared to the victim models M_{vic}. For instance, launching standard imitation attacks toward an LLCG model, whose BLEU score [72] is 23.2, can easily result in an imitation model with a close BLEU score (22.6). Moreover, we estimate that the cost of conducting the imitation attacks is very low, about 18.7 USD at most for our demos. These preliminary results suggest that LLCG APIs are vulnerable to imitation attacks; it is thus demanding to mitigate imitation attacks and IP theft.

Launching "online" imitation attack detection (as a form of intrusion detection with firewalls) is fundamentally challenging, which may often treat normal users as attackers and thus undermine the user experience. Instead, to protect IP of CV/NLP models, recent works have illustrated the feasibility of using watermark (WM) techniques [33, 45, 50, 61, 91, 92, 102]. Carefully crafted WMs are prepared and injected into victim model outputs, such that when using these WM-injected model outputs to train an imitation model M_{imi}, it is feasible to confirm IP theft over M_{imi}. In addition to research work, protecting model IP from imitation attacks has drawn attention from the industry, and model owners were beginning to value WM techniques for real-world applications. For instance, Google has been accused of training its AI chatbot Bard on data from OpenAI's ChatGPT without authorization [8]. At the same time, OpenAI has developed a working prototype [11] of WM techniques for ChatGPT, and this prototype allows a rough identification of the content generated by the GPT series model.[1]

To our best knowledge, WM methods are not yet applied to LLCG APIs. LLCG outputs (program code) are inherently distinct from texts or images, given that code is more structured, stricter, and less ambiguous. Thus, contemporary WM techniques [33, 41, 84, 90, 103] that add specific WM tokens into texts and images may likely break the syntactical forms of the LLCG outputs. Specific (abnormal) WM tokens may also be recognizable under WM recognition techniques, and are thus easily removed by attackers.

This work proposes the first WM technique to protect LLCG APIs. Our method is based on the observation that typical programming languages like Python have many token-level *synonyms*, such that replacing a token with its synonym does not change (or only trivially change) the functionality of the LLCG outputs. The mutated LLCG outputs also retain high readability (for normal users) and are indistinguishable from the normal outputs (robust against attackers). This observation enables the design of our WM framework, ToSyn, where each "WM" denotes the tweaked distribution $\hat{D}$ of certain **To**ken **Syn**onyms in the LLCG outputs.

We design six WM schemes, which rely on token synonyms formed from different programming language concepts. For the current implementation, ToSyn instantiates each scheme with several WM passes, resulting in over 30 WM passes (ToSyn can be easily extended with more WM passes). Each pass injects its WM by changing the distribution of certain token synonyms in the LLCG outputs ($D_i \rightarrow \hat{D}_i$). Moreover, ToSyn offers a statistical tests-based procedure to check a suspicious LLCG API and confirm if it is stolen from a protected LLCG API.

[1]Nevertheless, the technical details in [11] is obscure, and the prototype is likely not online yet.

We evaluate ToSyn on LLCG models derived from two popular LLM public backbones, CodeBERT [38] and CodeT5 [93]. We show that WMs do not undermine the quality of LLCG outputs (a small BLEU score downgrade of 3.85%), and the WM injection overhead is negligible (less than 2.1% in terms of LLCG throughput). We show that ToSyn's IP theft check is highly accurate (most F1 scores equal 1.0 with proper configurations). Moreover, ToSyn exhibits high robustness toward strong attackers; even if they know all six WM schemes, the attackers are unable to remove WMs without largely sacrificing the accuracy of stolen models (average BLEU score downgrade of 35.57%). We also evaluate the stealthiness of ToSyn, by measuring that WMs are well blended in the outputs of LLCG APIs. The results are consistently encouraging. In sum, we make the following contributions:

- We advocate for protecting IP of LLCG APIs with WM techniques. To our best knowledge, this paper proposes the first work, ToSyn, that protects LLCG models, a rising field that is evolving and commercialized in the industry.
- ToSyn's WMs are based on distributions of certain token synonyms, a stealthy scheme that retains high fidelity for normal users. Moreover, the WMs are carefully designed to be robust against adversaries, and incur little performance overhead. ToSyn also features a statistical tests-based procedure to confirm IP theft of a suspect model.
- We launch extensive evaluations to show that ToSyn can effectively insert stealth WMs and achieve high accuracy in IP theft detection, even in front of strong adversaries.

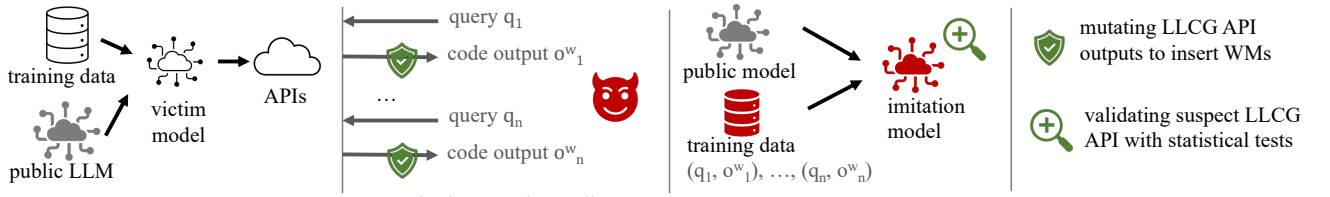
Availability. We have released a replication package at [22], including all code and data. We will make ToSyn (all code/data) publicly available upon acceptance and maintain it to benefit the community.

Table 1: List of key symbols used in the paper.

Notation	Description
$q_i \in Q, o_i \in O$	User queries and LLCG outputs
$o_i^w \in O^w$	WM-injected LLCG outputs
U_{id}	A random, unpredictable UID assigned to each user
$M_{\text{vic}}, M_{\text{pub}}, M_{\text{imi}}$	Target LLCG API (model); public backbones; and attacker's imitation model
$D_{\text{train}}, D_{\text{test}}$	Train split; test split
$D_{\text{train}}.l, D_{\text{test}}.l$	Natural language (NL) inputs in train/test splits
$D_{\text{train}}.c, D_{\text{test}}.c$	Program language (PL) outputs in train/test splits
P_i, TS_i	Each WM pass; the token synonym set in P_i
$\mathcal{P}_{id}$	A set of WM passes assigned to user U_{id}
D_i	Distribution of TS_i in D_{train}
$\hat{D}_i$	Tweaked distribution (i.e., WM) of TS_i
jv_i	The JSD-value when checking if P_i is "in" a suspect model M_{imi}
τ_J, τ_N	The JSD-value threshold; the #WM pass threshold

2 PRELIMINARY

Notations and A Holistic View. To ease reading of the following sections, we summarize key symbols used in this paper in Table 1. We will introduce each symbol when using it. Fig. 1 presents a holistic view of using LLCG APIs, launching imitation attacks, and explains how ToSyn fits this "big picture" for WM insertion and IP check. We denote the victim LLCG model as M_{vic}, which is fine-tuned from a public LLM backbone model M_{pub}. The imitation model successfully stolen from M_{vic} by the attacker is denoted as M_{imi}. We now explain each component below.



(a) LLCG setup. **(b)** Imitation attack – online. **(c)** Imitation attack – offline. **(d)** Core tasks of our framework.

Figure 1: A holistic view of LLCG APIs training, imitation attack, and mitigation enabled by ToSyn.

LLM-based Code Generation (LLCG). Due to the prosperous development of Transformer-based LLMs such as OpenAI's GPT2 and GPT3 [24, 28, 54], the code generation task, as a typical conditioned open-ended language generation task, is extensively tested and improved with much higher accuracy and applicability [23, 73].[2]

Though training data details are often obscure, modern LLCG services are advertised as being trained with millions or even billions of lines of code [6]. Moreover, as illustrated in Fig. 1(a), LLCG models are usually *not* trained from scratch; rather, they are usually fine-tuned (using the proprietary training datasets of the LLCG service provider) from public LLM backbone models M_{pub}, e.g., GPT2, Codex [77], CodeT5, and CodeBERT. For instance, the deep tabnine model [7] is based on GPT-2 and Tabnine allows its enterprise users to train a custom model by fine-tuning deep tabnine using the enterprises' private codebase. Note that Copilot and Tabnine are likely the two most visible code completion models in the community [17]. Also, some model providers like AI21 lab [1] and aiXcoder [2] allow their customers to train custom models that are fine-tuned to fit customers' private data. Typically, the input training data are dissected into sentences and further into tokens, where each token comprises a sequence of characters before being fed to LLMs. While many techniques are proposed to improve the performance of LLMs [100], tuning public LLM for the code generation task is still deemed highly expensive. Training datasets are often proprietary and large-scale, and the training process may often be conducted on GPU clusters for several days to weeks [24, 28].

During the online usage phase, users submit their queries to the LLCG APIs. Queries can be either a piece of natural language (NL) description of the expected generation outputs or an incomplete code prefix. Typically, LLCG models can predict a list of suggestions (ranked by the confidence scores) that are more likely to be generated according to inputs. For instance, given a function prototype description written in NL, LLCG APIs can return a function body.

Imitation Attack. An imitation attack (sometimes referred to as model extraction or model stealing attacks) aims to emulate the behavior of a target model M_{vic}, such that the adversaries can eventually form their own valuable local imitation model, M_{imi}, to sidestep the service charges and even offer competitive services. Overall, launching an imitation attack requires the malicious users to interact with the LLCG APIs. As illustrated in Fig. 1(b), the attackers first prepare a set of queries Q based on the LLCG documentation or the knowledge of the LLCG task. Each query q is sent to M_{vic} to obtain the corresponding valuable output o. The imitation model M_{imi} can be trained by fine-tuning the same public LLM backbone M_{pub} with the dataset $\{q_i, o_i | q_i \in Q, o_i \in O\}$, as shown in Fig. 1(c). To date, imitation attacks have been successfully demonstrated in both classification and generation tasks [33, 45, 50, 61, 91, 92, 102]. Particularly, for generation tasks whose outputs are NL sentences, each o_i is a sequence of tokens, where de facto attacks have shown that M_{imi} can manifest comparable performance with M_{vic} [45, 87, 99].

ToSyn. Fig. 1 illustrates two key tasks in ToSyn, where during the online query phase, it injects WMs into each LLCG output o_i. And given a suspect model, ToSyn validates this model to identify IP theft. Holistically, the service provider cannot distinguish normal users from malicious ones. Thus, WMs should be inserted in a stealthy manner and not undermine the quality of o_i. To clarify, some recent works [26, 55, 91] also propose to directly embed WMs in the deep neural network (DNN) parameters; this is often referred to as *white-box* WM, which mitigates "copy-paste" DNN models (e.g., reusing model weights) by adversaries. In contrast, in a typical remote API setting, *black-box* WM schemes are more suitable, where we mutate LLCG API's outputs, and also conduct an IP theft check by analyzing the outputs of remote suspect LLCG.

3 MOTIVATION

3.1 Threat Model

Some prior research offers privatization DNN deployment [47, 63, 64, 79, 80, 83], where model owners provide private LLCG models to users as services via APIs. The users often subscribe to the LLCG API service and are charged based on their submitted requests. As illustrated in Fig. 1, aligned with existing works [30, 67, 69, 70, 88], we assume that neither the LLCG service provider nor the attackers train their models from scratch. Rather, both of them benefit from the public LLM backbone M_{pub} such that their LLCG model M_{vic} and the stolen model M_{imi} are fine-tuned from M_{pub} for the code generation task. The LLCG provider usually fine-tunes M_{pub} with his *private* dataset to produce a highly valuable LLCG model M_{vic}. We assume that attackers know the architecture of M_{pub}, because various techniques facilitate attackers to infer the architecture of M_{vic} and identify M_{pub} [27, 30, 43, 47, 64, 81, 86, 96].

We assume that attackers can query the LLCG APIs for a reasonable, affordable amount of times. As in Fig. 1(c), with query outputs and M_{pub}, attackers can train their imitation model M_{imi} which is expected to manifest similar behavior as M_{vic}. To ease reading, we leave further assumptions required by our IP theft detection in Sec. 6. Also, recent studies show that model predictions may leak private training data [58], which is orthogonal with this paper.

[2] A full introduction of Transformer and its usage in open-ended language generation is beyond the scope of this paper. Audiences can refer to [85].

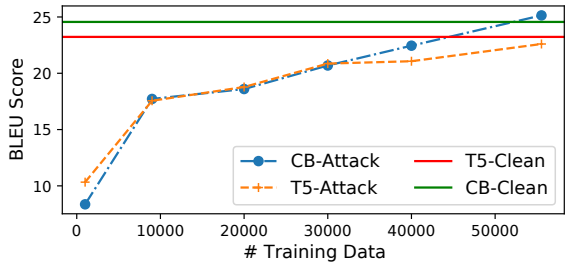


Figure 2: PoC attacks. Each attacker-submitted query leads to one training data sample.

3.2 Imitation Attack Demo

Setup. Launching imitation attacks on CV and NLP models has become a well-known threat [30, 67, 69, 70, 87, 88, 98]. To motivate our WM-based protection, we present a proof-of-concept (PoC) attack on LLCG models. Two popular LLM backbones, CodeT5 and CodeBERT, are leveraged. We use our training dataset D_{train} to fine-tune two backbones and generate two LLCG models M_{vicCB} and M_{vicT5}. We introduce CodeT5/CodeBERT and D_{train} in Sec. 7. Our attack setup is aligned with the standard setup of imitation attacks noted in Sec. 2. When attacking M_{vicT5}, the attacker randomly selects N inputs from $D_{\text{train}}.I$ to query M_{vicT5},[3] and uses the query outputs to fine-tune the same CodeT5 backbone. This would give the attacker an imitation model M_{imiT5}. The same attack is applied to M_{vicCB} to obtain M_{imiCB}.

Attack Results. Fig. 2 illustrates the attack results. We report the accuracy of the attacker's imitation model M_{imi} (measured using the test dataset D_{test}; see Sec. 7) w.r.t. the number of attacker-submitted queries Q_N to M_{vic}. We find that with a reasonably small Q_N (e.g., $Q_N = 9000$, which is about 15% of $D_{\text{train}}.I$), the accuracy of M_{imi} is already promising: 17.6 for M_{imiT5} and 17.7 for M_{imiCB}. As expected, with more queries, the accuracy of both M_{imi} increases, where M_{imiCB} can even outperform its "teacher" (the blue line) when attackers use all $D_{\text{train}}.I$ to query.

PoC and Enhancement. Fig. 2 is for PoC purpose: real-world imitation attacks often leverage heuristics or domain-specific knowledge to reduce Q_N [46], and therefore, we expect to obtain better M_{imi} with even fewer queries. Nevertheless, Fig. 2 already demonstrates the *high feasibility* of launching imitation attacks toward LLCG. We are however not stating that such imitation attack demos work for all models. Some commercial models like CodeX [77] are not open-source released, and it is hard to somehow fine-tune CodeBERT/CodeT5 and let them approximate ("imitate" via imitation attack) commercial models like CodeX. We leave exploring sophisticated attacks and optimizations to future work.

Cost Estimation. Further to the high attack feasibility, we clarify that the cost of launching a mitigation attack is small (roughly 18.7 USD at most). We estimate the cost from the following two aspects.

① *Online Querying Cost.* Billing for LLCG queries can be broadly categorized as either monthly or "pay-as-you-go". For example, GitHub Copilot charges 10 USD per month [4]. In the latter charging mode, users pay according to the number of queries they send or the total length of tokens they receive from the LLCG API. For instance, AI21 charges 0.03 USD per 1K tokens and 0.0003 USD per query [1]. Thus, suppose that we query 55,538 times (the entire $D_{\text{train}}.I$) to steal M_{vic}, the cost would range from 1.827 USD to 10 USD. Moreover, if the attackers register a new account to use such LLCG services, they would presumably receive some free credits (10 to 90 USD) to use. Thus, queries launched by imitation attacks may incur zero cost.

② *Offline Training Cost.* As a common practice, we estimate the local training cost using an Amazon EC2 P2 cloud machine (the p2.xlarge with 61 GiB instance [3]). We report that it charges about 5.1 USD for training M_{imiT5} and 8.7 USD for M_{imiCB}.

[3] $D_{\text{train}}.I$ represent LLCG queries written in NL gathered from the training set D_{train}.

3.3 Existing WM technique

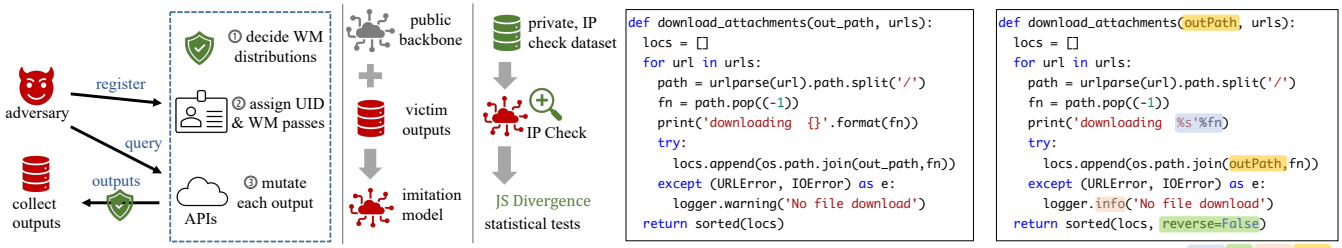
Preliminary Study Results. We also conduct a preliminary study on the effectiveness of existing WM techniques. We apply CATER, the state-of-the-art WM method designed for natural language content in black-box settings [45], to the training dataset D_{train}. CATER has 669 synonym groups as candidate WMs and contains an optimization method to decide the watermarking rules; this optimization method aims at minimizing the distortion of overall word distributions while maximizing the change in conditional word selections. However, our study shows that only two words ("new" and "other") are triggered among all 55,538 examples, illustrating that the large pool of potential word pairs in CATER is rarely used in code snippets. This result suggests that de facto WM techniques may be likely ineffective for LLCG services in black-box settings, thus motivating the design of our new WM approach.

"Soft" WM vs. "Hard" WM. Recent works [52, 60] have proposed "soft" WM technique, where the WMs are embeded during content generation, in contrast to the "hard" WM technique introduced earlier where the watermark is applied after generation. For example, Kirchenbauer et al. [52] propose a framework that embeds WM into content by modifying the model's sampling process. However, we argue that such "soft" WM technique is impractical for our scenario for several reasons. One major issue is that "soft" WM techniques are by principle designed for white-box scenarios, as they require intercepting and changing the probability distribution of the model's vocabulary and intervening the generation process.

Additionally, the semantics of the generated content often change substantially when applying "soft" WMs. Compared to ToSyn which uses synonyms and mostly preserves the code semantics, those "soft" techniques provide less guarantee of the output functionality. Furthermore, "soft" techniques require setting candidate WMs and rules to guide sampling. For instance, [52] requires defining a "green list" of the preferred tokens and a "red list" of tokens that should be avoided. However, those lists are not tailored for code snippets, and thus may likely have similar limitations as that of CATER.

4 DESIGN CONSIDERATION

In the following two sections (Sec. 5 and Sec. 6), we introduce the WM insertion and IP check phases of ToSyn. Before introducing the technical pipeline of ToSyn, we first discuss the following key design considerations. Aligned with prior relevant works, we aim to achieve the following properties in our WM design:



(a) WM insertion. **(b)** IP check. **(c)** Original output. **(d)** Output mutated by BSR PS TFR CST.

Figure 3: The technical pipeline of ToSyn, including the WM injection phase (discussed in Sec. 5) and the IP check phase (discussed in Sec. 6). We also present a mutated code snippet using four WM passes.

Fidelity. WMs should not undermine the utility of LLCG APIs. As noted in Sec. 2, the LLCG APIs cannot distinguish normal users and attackers. As a result, we require the LLCG API outputs should manifest high quality and readability, even with WMs added.

Reliability. WMs should be verified with high confidence (low false negatives) in suspect models, and with low confidence (low false positives) in normal models, i.e., irrelevant models owned by others.

Robustness. WMs should be robust against adversarial attempts which aim at removing WMs. Note that WMs should exhibit high robustness even if it is under a strong attacker who knows the technical details of our WM passes.

Stealthiness. A WM-injected code should appear close to normal code. In other words, the WMs should be stealthy and attackers can hardly tell if a code snippet is mutated by WMs or not.

Cost. WMs should be efficiently inserted within query outputs, and it should not largely slowdown the LLCG API throughput.

We clarify that ToSyn satisfies all these considerations. In the following section, we introduce the design of ToSyn, and we evaluate the corresponding WM properties in Sec. 8.

5 TOSYN – WM INJECTION STAGE

Our key observation is that certain tokens in programs written in modern programming languages (e.g., Python) can be replaced with their "synonym" tokens without changing (or only trivially changing) the semantics. Our WM design exploits this observation: in particular, given a query q_i submitted from (potentially malicious) users, ToSyn mutates the LLCG API output—code snippet o_i—with a certain number of WM passes. Each pass mutates o_i into o_i^w to inject WM. o_i and o_i^w manifest similar visual appearance and readability. The LLCG API then returns o_i^w to the user.

5.1 WM Definition

We now present the definition of WM on the basis of token synonyms as follows.

Definition 1 (Watermark (WM)). *Token synonyms have (nearly) identical semantics yet are visually distinct. Accordingly, a WM denotes a tweaked distribution $\hat{D}$ of token synonyms among LLCG outputs $o_i \in O$ given to an LLCG user. Each WM pass P_i is associated with a token synonym set TS_i. Let D_i be the token synonyms TS_i distribution over M_{imi}'s training dataset D_{train}. P_i will change D_i into $\hat{D}_i$ to inject the ith WM. To do so, given an LLCG output o, P_i mutates a specific token t with a sampled token t' where $t, t' \in TS_i$. The sampling is in accordance with $\hat{D}_i$.*

We design six WM schemes on the basis of token synonyms formed from distinct language concepts. As a proof-of-concept implementation, ToSyn instantiates each scheme with several WM passes, resulting in total 34 WM passes. WM schemes can be easily instantiated into a substantial number of possible WM passes (see Sec. 11 for discussion), and we advocate an LLCG service to equip several hundreds of WM passes when in production.

5.2 WM Insertion Procedure

Fig. 3(a) depicts the WM insertion phase in ToSyn. We now discuss three key steps in Fig. 3(a). The IP theft check phase of ToSyn (as in Fig. 3(b)) will be discussed in Sec. 6.

① **Deciding WM $\hat{D}_i$.** For each P_i, we decide a proper token synonym distribution $\hat{D}_i$, and accordingly sample one synonym $t' \in TS_i$ to replace t in each LLCG output. In particular, let D_i represent the distribution of token synonyms in TS_i over the training dataset D_{train} of M_{vic}. We decide a $\hat{D}_i$ by shuffling D_i, which manifests the *maximal cosine distance* to D_i. This $\hat{D}_i$ represents the WM distribution that P_i aims to achieve when mutating LLCG API outputs. Soon in this section, we will introduce each P_i. In particular, the P_{TFR} pass mutates three different uses of the Python `logging` API to inject WM. Thus, consider the following example:

Example 5.1. *Suppose in the training dataset D_{train} of the protected LLCG, three variants of the Python* `logging` *API (see below) form a distribution D_{TFR} as $\{0.17, 0.18, 0.65\}$. We shuffle D_{TFR} and consider $\{0.65, 0.17, 0.18\}$ as the targeted WM distribution $\hat{D}_{TFR}$, given $\hat{D}_{TFR}$ has the maximal cosine distance with D_{TFR} among all possible shuffles of D_{TFR}. Then, when P_{TFR} is activated (see ② below), and given an LLCG output containing any variant of the concerned* `logging` *API, we sample one of the three synonyms to replace the original variant. Thus, the distribution of these three variants in the watermarked LLCG outputs O^w shall become approximately $\{0.65, 0.17, 0.18\}$ with sufficient queries.*

② **Deciding UID and WM Passes.** Each (malicious) user, when registering our protected LLCG, is assigned an N-bit user id (UID) U_{id}; N equals the total number of WM passes. We require UID to be random, unique, and unpredictable by attackers.[4] We then use U_{id} to choose $\mathcal{P}_{id}$, denoting the involved WM passes. In particular,

[4]This can be achieved by using standard crypto hash (e.g., SHA256) with Salt [95].

$\mathcal{P}_{id} \leftarrow U_{id}$, where the $(i+1)$th pass is activated when the ith bit ($i \in [0, N-1]$) in U_{id} is 1.

Given that $\mathcal{P}_{id}$ is unique to each user, we use its presence in a suspect imitation model (details in Sec. 6) to localize specific adversarial users of our protected LLCG. Given $N = 34$ in our current implementation, we require that the number of activated passes (i.e., $|\mathcal{P}_{id}|$) should be in the range of $[10, 20]$. That is, the protected LLCG API can support nearly 15B users.[5] Too few activated passes increase the difficulty and reduce accuracy in the IP check phase (reflected by our study in **RQ3**; see Sec. 8.3). In contrast, too many activated passes increase the likelihood that $\mathcal{P}_a$ and $\mathcal{P}_b$, two WM pass sets prepared for users Alice and Bob, overlap. In IP theft check (Sec. 6), such overlapping reduces the possibility of distinguishing specific (malicious) users.

③ **Deciding Proper P_i.** The user then starts to submit his queries to the LLCG API. For every generated LLCG output o_i for this user, we analyze each token $t \in o_i$, and see if it is applicable to a WM pass $P_i \in \mathcal{P}_{id}$. If so, we use P_i to mutate $t \in TS_i$ with a sampled $t' \in TS_i$ according to $\hat{D}_i$.

5.3 WM Pass Taxonomy

We design six WM schemes based on different programming language features. Given ToSyn currently supports LLCG outputs in Python and the overall popularity of the Python language, our WM schemes primarily rely on Python language features. WM schemes are "lightweight," in the sense that they only slightly change the outputs and retain high visual similarity and low extra cost on the transformed outputs; see our evaluation of WM stealthiness in Sec. 8.5. We also present discussions discussion about extensibility of our proposed technique (e.g., to other languages) in Sec. 11.

Below, we introduce each WM scheme by briefing one to two WM passes of this scheme. Audiences can refer to Sec. 7 for the list of all passes in ToSyn and the full documents of passes are at our project website [22].

Built-in Structure Replacement (BSR). This scheme replaces built-in structures with their semantically equal structures. This is technically feasible due to a rich set of syntactic sugars in Python. Considering the following illustration of two WM passes:

```
d1 = dict(name='1') #initialize 1

d1 = {}
d1['name'] = '1'    #initialize 2
```

```
h,w = 'hello','world'

print('%s,%s' % (h,w))     # format1
print('{},{}'.format(h,w)) # format2
```

where the first case is mutated by an instruction-level BSR pass; since a built-in type data structure like `dict` and `list` can be initialized in different ways, we identify such cases and replace the initialization methods. The second case is at the API level: a Python API call frequently accepts syntactically different yet semantics-identical inputs. For example, Python `print` allows using a format string or simply using the percent sign to specify the output format. Notably, we manually refer to the Python language reference [74] to identify the potential passes in this scheme.

Code Style Transferring (CST). WM passes belonging to this scheme mutate code with different coding styles. They may consider different naming conventions for function parameters and the usage of parentheses, respectively. Considering the following cases:

```
def add(UserAddNum):   #camel case
    return 1 + UserAddNum

def add(user_add_num): #snake case
    return 1 + user_add_num
```

```
if userid == 0: #without parentheses
    return 0

if (userid == 0): #with parentheses
    return 0
```

where for function parameters, we design one pass to rename it from the camel case into snake case and accordingly replace every usage of it within the function. Besides these two naming conventions, WM passes in the CST scheme can support the Hungarian case and the Pascal case. The right-hand case shows that the other CST pass will add or trim off parentheses in the `if` conditions.

Param Specification (PS). Passes in this scheme mutate API outputs by adding (or removing) default parameters. Consider the following case which is mutated by one PS pass:

```
arr = [1,3,4,2]
arr_new = sorted(arr)            # without specification

arr_new = sorted(arr,reverse=False) # with specification
```

where we explicitly specify the parameter named "reverse" for function `sorted` to be "False". It is quite common that the developer sets a default value for the parameter to enhance readability and maintainability. Besides `sorted`, many functions in Python provide default values for their parameters, which facilitates instantiating this scheme into other passes.

Lib Alias (LA). These passes focus on the usage of library alias. Overall, Python features a variety of third-party libraries, and it is common for developers to import one specific module and give it an alias before usage. Consider the following cases:

```
import numpy
a = numpy.array([1,2,3])

import numpy as np     # alias
a = np.array([1,2,3])
```

```
import matplotlib.pyplot
matplotlib.pyplot.figure(figsize=(4,4))

import matplotlib.pyplot as plt # alias
plt.figure(figsize=(4,4))
```

where in the left-hand case, the widely-used scientific computing package, `numpy`, is imported. Here, one LA pass can use alias `np`. Similarly, another LA pass can use alias `plt` to represent the data visualization library, as shown in the right-hand case. Similar to the BSR scheme, we manually survey aliases used in open-source Python projects to select a set of plausible alternative names for this scheme. Given that Python has more than 411,000 packages in PyPI [14], it would be easy to contain other popular libraries such as "Pandas" and "Scipy" when more LA passes are needed. The list of such popular Python packages can be found at [18, 19].

Third-Party Function Replacement (TFR). This scheme focuses on third-party APIs. TFR passes either mutate built-in functions with third-party APIs, or explore different variants of a third-party API. Consider the following cases:

```
row = -1000
row = abs(row)    # built-in function

row = np.abs(row) # using numpy
```

```
import logging
logging.info('message1')
logging.debug('message1')
logging.warning('message1')
```

where in the left-hand side case, one TFR pass uses the `abs` function in the `numpy` package to replace Python built-in `abs`. The right-hand case replaces variants of a third-party API. As a widely-used package for recording the log, `logging` has a set of API variants, which are replaced by this TFR pass. Regarding the prevalence of synonyms in third-party libraries, synonyms are more common in libraries furnishing mathematical operations; for instance, `numpy` and `pandas` that are extensively used in daily development.

Funcall Link Re-Ordering (FLR). This scheme considers function call sequences. If the sequence of function calls can be re-ordered

[5] $C_{34}^{10} + C_{34}^{11} + \ldots + C_{34}^{20} \approx 15$ billion.

without changing the semantics, FLR passes would re-order them. Consider the following case:

```
cstr = 'hello world'
t = cstr.lower().strip() # sequence1

t = cstr.strip().lower() # sequence2
```

where two imposed function call sequences, `lower` → `strip` and `strip` → `lower`, do not influence the value in `t`. Notably, the current scheme primarily focuses on string-related operations. It fails when method calls occur across different statements, which rarely happens in our observation. Users can extend this scheme to other operations or customize it to handle edge cases.

Mutation Cases. Fig. 3(c) and Fig. 3(d) illustrate how LLCG outputs are mutated. Given a natural language query "*downloads wordpress attachments and returns a list of paths*", an LLCG model, fine-tuned from CodeT5, provides the code output in Fig. 3(c). The code output can successfully trigger four types of WM passes (as noted in the caption). We highlight the mutated code tokens with different colors in Fig. 3(d), which is the final output returned to the user.

5.4 Analysis and Alternative Design

We discuss the strength of our WM technique in the following conceptual perspective.

Defense Assumption. As a practical assumption [45], a powerful attacker may be aware of all six WM schemes. Nevertheless, we assume that the attacker is unaware of the specific WM passes instantiated by the LLCG service. This is reasonable: a WM scheme can derive a large number of WM passes, meaning a production LLCG service can easily equip hundreds of passes. As a result, we assume that the attackers are unaware of the set of specifically instantiated WM passes in its target LLCG.

Defense Analysis. Overall, we believe the design rationality here is similar to the kerckhoffs's principle in cryptography, such that the cryptographic algorithms are known to the public, but the secret key is not; again, in our context the "secret" is the specifically instantiated WM passes in the target LLCG. Also, given his UID is randomly generated and obscure to him, WMs applied to his received LLCG outputs O^w are distinct from other users. This enables localizing specific adversarial users, as we will discuss in Sec. 6.

To successfully remove WMs injected in $o_i^w \in O^w$, the attacker (with UID U_{id}) would need to detect the applied WM passes $\mathcal{P}_{id}$. This is very challenging: given that the training dataset of M_{vic} is private, the distributions D_i of token synonyms in the training dataset are unknown to the attacker. Attackers may leverage "out-of-the-box" techniques to recognize and remove WMs in LLCG outputs. Or, a "fully-knowledgeable" attacker (who knows every WM scheme) may customize WM removal tools to remove all potential WM tokens related to each WM scheme. We evaluate these attackers and find them ineffective in Sec. 8.4.

Alternative Design Analysis. Some prior works [84] propose selecting a small fraction of queries and deliberately preparing false outputs. Observing that such wrong outlier outputs can often be easily memorized by M_{imi}, the model ownership can be validated by checking the presence of such wrong outputs. Nevertheless, such a scheme would likely undermine the utility, given that normal users may suffer from those wrong outputs. In contrast, WMs inserted by our technique would not affect the utility of the LLCG outputs; see evaluations in Sec. 8.1 and Sec. 8.5.

6 IP THEFT CHECK STAGE

In this section, we first present the motivation for the IP theft check stage, as well as our assumptions. We then introduce the technical details, which consist of three steps: checking individual WMs, evaluating all WMs collectively, and localizing the attacker.

6.1 Motivation and Assumptions

Before presenting the technical details of IP check in accordance with Fig. 3, we first discuss the motivation and assumptions of this phase below.

Motivation. Given a suspect LLCG model M_{imi}, the owner of LLCG model M_{vic} would wish to check if M_{imi} is produced by launching an imitation attack toward M_{vic}. This IP check is feasible, in case M_{imi} is trained using the WM-injected outputs of M_{vic}. We have depicted the high-level procedure in Fig. 3(b). We now discuss the assumptions of this phase and techniques.

Assumption. "Attackers" refer to malicious users who register the protected LLCG service. However, as a practical assumption, we do *not* require to know the owner of M_{imi} when launching IP check; M_{imi} may be offered by the attacker himself, or by a third-party who obtains the WM-injected outputs of M_{vic} from the attacker. In fact, since each user is assigned a UID and a unique WM pass set $\mathcal{P}_{id}$, we can identify the exact malicious user who launches the imitation attack with high accuracy at this step.

As shown in Fig. 1, we assume the public backbone M_{pub} used to train M_{vic} is known to attackers; therefore, attackers can train M_{imi} by fine-tuning M_{pub} using their gathered (WM-injected) LLCG outputs O^w. As mentioned in Sec. 5.4, we also assume that "fully-knowledgeable" (FK) attackers may know all WM schemes, though given the high flexibility of instantiating each scheme, FK attackers do not know the exact WM pass set the target LLCG service is using. Sec. 8.4 evaluates the ineffectiveness of FK attackers.

When performing the IP check, we require a private verification dataset $\mathcal{V}_v$, such that we can use every sample in $\mathcal{V}_v$ to query M_{imi}, as another LLCG API, and collect its corresponding outputs. We assume that $\mathcal{V}_v$ is *not* known to the public. This is a reasonable assumption consistently taken by prior WM works [44, 45].

6.2 IP Check Procedure

We now present the technical details of the IP check procedure in the following steps.

Check Individual WM. Once we suspect that model M_{imi} may be stolen from M_{vic}, a private verification dataset $\mathcal{V}_v$ is used to query M_{imi}. We record their responses (i.e., code outputs), and compute the distribution D_i' of token synonyms belonging to each WM pass P_i. The computed distribution D_i' would be compared with the WM distribution $\hat{D}_i$. Here, we require that $\mathcal{V}_v$ and the private training dataset D_{train} of M_{imi} share the same distribution of token synonyms, e.g., $\mathcal{V}_v$ is part of D_{train}, or they are collected from the same data source. See Sec. 7 about how we prepare $\mathcal{V}_v$ and D_{train}.

At this step, we employ Jensen-Shannon Divergence (JSD) [9], a common statistical test method to measure the similarity between

two probability distributions. JSD quantifies the similarity of D'_i and $\hat{D}_i$: a smaller JSD value indicates that two distributions are more similar. Specifically, we employ a threshold τ_J (e.g., 0.05), so that when the JSD value jv_i is less than τ_J, we acknowledge that D'_i and $\hat{D}_i$ have a significant degree of similarity (from the statistics perspective). In other words, we believe that the training dataset of the suspect M_{imi} has WM $\hat{D}_i$ injected by P_i.

Consider All WM Passes. Since we do not know which malicious user contributes to the training dataset of M_{imi}, we need to test all WM passes. That is, we accumulate N JSD values in accordance with all WM passes in ToSyn. We then decide if each JSD value jv_i is smaller than τ_J; if so, P_i is detected. The suspect model M_{imi} would be eventually deemed as the imitation model if the number of detected WM passes is above or equal to another threshold τ_N. A smaller τ_J means a stricter requirement in deciding if a WM pass is detected, while a larger τ_N means more WM passes need to be detected to confirm IP theft. In Sec. 8.3, we assess how these two parameters affect the accuracy of the IP theft check phase. Notably, most WM passes are independent, meaning applying any scheme should not impact others. A few passes could influence the "transformable candidates" of others, so we fix the order in which those passes are used.

Localize Specific Attackers. Recall that each (adversarial) user is assigned a unique ID (UID) and, accordingly, a unique WM pass set $\mathcal{P}_{id}$. Thus, it is possible to localize the specific attacker by comparing all "activated" WM passes in M_{imi} with user profiles of our protected LLCG. A malicious user is identified if his/her WM pass set is consistent with the activated WM passes in M_{imi}. Given that said, we admit that this step is *challenging*, because many factors (e.g., the sensitivity of the JSD tests, imitation learning quality) may influence the detection results; for instance, one or two drifted JSD values may result in identifying different, innocent users. We evaluate the accuracy of this step in Sec. 8.3.

7 IMPLEMENTATION AND SETUP

We implement ToSyn in Python, with about 3,200 LOC. ToSyn currently mutates LLCG API outputs in Python; see Sec. 11 for extensibility to other languages and Sec. 1 for artifact links.

Parsing and Mutating Code Generation Outputs. In this research, we primarily consider inserting WM into Python code, as Python is pervasively used; We parse Python code with tree-sitter [25], a dependency-free parser generator tool. Note that it does not require the input code to be executable, meaning that incomplete code fragments can be parsed without a building script. We first parse the code generation output into a concrete syntax tree, and iterate over all nodes to apply proper WM passes.

WM Pass. We instantiate 34 WM passes in accordance with our six WM schemes, whose details are in Table 2. Column "Schemes" specify each scheme, and "Num" denotes #WM passes belonging to a scheme. We measure the "applicability" of each pass by analyzing if it is applicable to every O_i in the M_{vic} training dataset D_{train}. In "Freq.", we divide passes under each scheme into high frequency (applicability greater than 5%), medium frequency (applicability within $[1\%, 5\%]$), and low frequency (applicability within $[0.2\%, 1\%]$).[6]

[6]No WM pass has applicability lower than 0.2%.

Table 2: WM passes implemented in ToSyn.

Schemes	Num.	Freq.
Built-in structure replacement (BSR)	14	8/3/3
Code style transferring (CST)	2	2/0/0
Param specification (PS)	3	1/2/0
Lib alias (LA)	2	2/0/0
Third-party function replacement (TFR)	11	7/3/1
Funcall link re-ordering (FLR)	2	0/0/2

Task. As noted in Sec. 2, LLCG APIs may accept queries in natural language (NL) or programming language (PL). ToSyn is evaluated against the "NL-PL" code generation task, which is commercialized and extensively used in the real world [1, 6]. Nevertheless, ToSyn is also technically applicable to the "PL-PL" task, given that ToSyn does not mutate LLCG inputs.

Metric. We use BLEU4 [72], a common token-level metric, to assess the quality of generated code. We use "BLEU score" to refer to BLEU4 in the paper. To clarify, using BLEU scores to measure code generation quality and illustrating that models can find "better code generation" is a common practice [21, 28, 38, 49, 62, 73, 89, 93].

LLCG Models. To evaluate the effectiveness of our approach, we consider two LLM public backbones: CodeBERT and CodeT5. CodeBERT is built on a multi-layer bidirectional Transformer encoder and pre-trained on large-scale text-code pairs under two tasks: masked language modeling (MLM) and replaced token detection (RTD). The former task requires the model to predict the original token in the masked positions, while the latter one uses a discriminator to distinguish the replaced tokens from the normal ones. CodeT5 is a unified encoder-decoder transformer model. It first uses an encoder to encode the input sequence, and then employs a left-to-right language model to decode the output sequence conditioned on the input. Various pre-training objectives like Masked Span Prediction (MSP) and Masked Identifier Prediction (MIP) are proposed to fulfill the gap between NL and PL. Overall, both considered models have been shown as proper for many sequence-to-sequence tasks [56, 97], including the "NL-PL" code generation.

Dataset Setup. We reuse the dataset proposed by Wei et al. [94] following its official split: train (55k), valid (18k), and test (18k). We use D_{train} and D_{test} to present the train and test splits of this dataset, respectively. Moreover, $D_{\text{train}}.l$ and $D_{\text{test}}.l$ denote the LLCG inputs (in NL) in the train/test splits, whereas $D_{\text{train}}.c$ and $D_{\text{test}}.c$ denote the generated code (the LLCG outputs; in Python) in the train/test splits. As noted in Sec. 4, we assume that both attackers and LLCG owners use the same public models M_{pub} as the backbone to generate both victim LLCG models M_{vic} and imitation models M_{imi}. Thus, with D_{train}, our victim models M_{vicCB} and M_{vicT5} are trained by fine-tuning CodeBERT and CodeT5, respectively. The attackers should have *no* access to $D_{\text{train}}.c$, as $D_{\text{train}}.c$, denoting code snippets, are assumed as proprietary to LLCG owners.

When attackers launch an imitation attack, they leverage NL questions Q to query M_{vic} (M_{vicCB} and M_{vicT5}) and collect the WM-injected code outputs O^w. The attackers then use a training dataset formed by $\{(l, o^w)|l \in Q, o^w \in O^w\}$ to fine-tune the same public backbone M_{pub} to obtain their own imitation model M_{imi} (M_{imiCB} and M_{imiT5}). We clarify NL queries in Q are generally mundane, e.g., "*generating a bubble sort function*", "*returns a function that reads an unsigned 64-bit integer*". It is the associated code generations O^w that are valuable and proprietary. For the current evaluation,

we use $D_{\text{train}}.l$ to serve Q when launching the attack. In practice, attackers often start from a few "seed" queries, and continuously generate more queries [31, 53, 68]. Nevertheless, we view using full $D_{\text{train}}.l$ as Q is preferable in our evaluation, as it shows the presumably "upper-bound" attacker capability of possessing a large number of diverse NL queries. To evaluate the performance of M_{imi}, we use D_{test}. Moreover, we treat D_{test} as the verification dataset $\mathcal{V}_v$ when conducting IP theft check toward M_{imi}. This is a realistic and consistent setting taken by prior works [44, 45]. All experiments are conducted on a machine with Intel Xeon Patinum 8276 CPU, 256 GB main memory and 4 NVIDIA A100 GPUs.

8 EVALUATION

In accordance with the five key properties of a desirable WM technique noted in Sec. 4, we design the following five research questions (RQs) and answer each of them in this section.

- **RQ1 (Fidelity)**: Do our WMs undermine the output quality of the protected LLCG APIs?
- **RQ2 (Cost)**: Do our WMs incur too much overhead on the API or the outputs?
- **RQ3 (Reliability)**: How accurate it is when performing IP check toward a suspect model?
- **RQ4 (Robustness)**: Are our WMs resilient to removal attacks?
- **RQ5 (Stealthiness)**: Are WM-injected code snippets distinguishable from normal code snippets?

Table 3: Fidelity evaluation. All data are BLEU scores.

Setting		CodeBERT	CodeT5
Clean		24.56	23.23
Outputs with WM injected		24.15	22.88
M_{imi}	Set1 (10 WM passes)	23.61	22.73
	Set2 (15 WM passes)	23.45	22.35
	Set3 (20 WM passes)	23.32	22.39

8.1 RQ1: Fidelity

To measure the fidelity, we consider the influence of the WMs toward both normal users and attackers. The results are reported in Table 3. *Clean* stands for the responses provided by the LLCG API (M_{vicT5} and M_{vicCB}); this is the baseline, denoting the response quality (in terms of BLEU scores) without injecting WMs. The third row reports the BLEU scores of WM-injected responses. It is unsurprising that BLEU scores of WM-injected responses are slightly lower than the clean ones. To clarify, BLEU is a token-level metric and thus would decrease marginally when outputs appear slightly different from ground truth, although they share identical functionality. We interpret the evaluation as highly encouraging; the WM-injected code outputs are nearly as good as the clean outputs (downgrade 0.41 for M_{vicT5} and 0.35 for M_{vicCB}). That is, for normal users, the WM-injected LLCG APIs can still provide high-quality service.

In the M_{imi} rows, we launch three sets of imitation attacks, each of which contains five attackers. In total, we launch 30 imitation attacks as a trade-off between comprehensiveness and pragmatism given our computational resources. It takes approximately 20 hours to train one imitation model, making it infeasible to evaluate hundreds of attackers. We assign different, randomly generated UIDs to attackers, and therefore, each attacker is assigned a randomly chosen set of WM passes $\mathcal{P}_{id}$. Note that we benchmark three settings of the $\mathcal{P}_{id}$ size, meaning that a $\mathcal{P}_{id}$ may include 10, 15, or 20 WM passes. We report the mean BLEU scores of attacker-obtained imitation models M_{imi} under each setting.

The evaluation results over M_{imi}, together with the PoC attacks in Sec. 3.2, indicate that high-quality imitation models can be stolen from the LLCG APIs. Moreover, it is seen that even with WMs injected, the imitation models can still achieve high BLEU scores. When we increase the number of the involved WM passes, the BLEU scores only fluctuate within a small range (± 0.4). Overall, the results reflect that the WMs are "stealthy" and do not degrade the output, even being taken by the attackers. This is desirable, because if WMs would downgrade the imitation model performance, it is mostly inevitable that the normal users would also suffer.

Table 4: Cost evaluation.

	Response Time (s)	WM Insertion Time (s)	Response Slowdown (%)	Avg. #token Changes (%)
M_{vicCB}	858	2.3	0.26	-0.54
M_{vicT5}	57	2.3	4.03	-0.50

8.2 RQ2: Cost

We study the cost of our technique from two aspects. First, we measure the time taken by inserting WMs into LLCG outputs. Injecting WM should be rapid, given that slow instrumentation delays the API throughput and undermines user experience. We benchmark and compare the processing time of 1K user queries with and without WMs. The results are in Table 4, showing that the response time of LLCG APIs is not significantly affected by our WMs. To be specific, due to distinct model architectures, M_{vicCB} (an encoder-only model) takes 15.05 times longer to generate the code than M_{vicT5} (an encoder-decoder model), resulting in the relatively smaller response slowdown ratio.

We also study how WMs influence the LLCG code outputs. It is not straightforward to benchmark the overhead of WM toward LLCG outputs, given that they are often incomplete code snippets with varying lengths and complexity. At this step, we resort to the average changes in terms of token numbers in each code output. With WMs injected, we report that the average changes are around -0.54% (for M_{vicCB}) and -0.50% (for M_{vicT5}), meaning that the number of tokens is slightly decreased in the WM-injected outputs. This is reasonable, as many WM passes slightly simplify the code syntax. In sum, we interpret that WM passes only perform mild changes, incurring negligible extra overhead.

8.3 RQ3: Reliability

This section studies the accuracy of performing the IP theft check (Sec. 6). In accordance with Sec. 6, we evaluate 1) the accuracy of confirming a suspect model M_{imi} as a WM-injected model, and 2) the accuracy of localizing the specific malicious user who launches the imitation attack.

Confirming a Suspect Model M_{imi}. Recall that given a suspect model M_{imi}, we leverage a verification dataset $\mathcal{V}_v$ to decide if M_{imi} is stolen from the victim model M_{vic}. We require $\mathcal{V}_v$ to be private. For the evaluation, we use the test split D_{test} as our verification dataset; this is a setting also taken by previous WM works [44, 45]. In **RQ1**, we launch a total of 15 + 15 attackers to conduct imitation

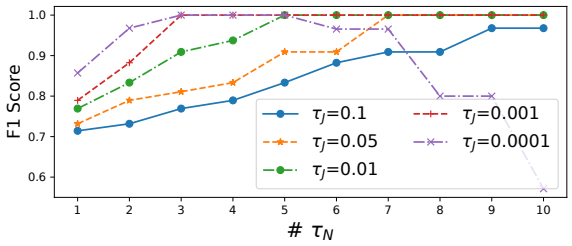


Figure 4: F1-score for checking IP theft over M_{vicCB}.

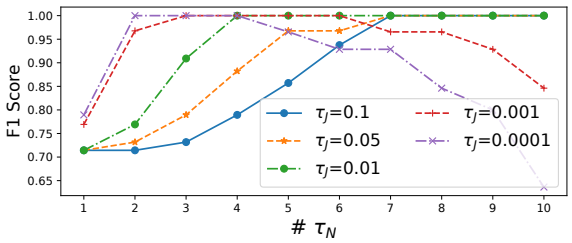


Figure 5: F1-score for checking IP theft over M_{vicT5}.

attacks toward M_{vicCB} and M_{vicT5}. At this step, we take those 15 imitation models as "positive" samples for IP theft check under the CodeT5 setting (similar for CodeBERT as well). We also prepare the same amount (15) "normal" models which are trained with 40,000 samples randomly selected from the train split D_{train}. These models represent "innocent models" because the samples are randomly selected from the entire D_{train} without injecting any WMs.

As introduced in Sec. 6.2, our IP theft check phase involves two hyper-parameters, τ_J and τ_N, denoting the JSD value threshold and the number of WM passes needed to confirm the IP theft. For the JSD value threshold τ_J, we consider five reasonable values in our research context. As for τ_N, given that we inject at least ten WM passes for a user (recall as shown in Table 3, we inject 10, 15, or 20 WM passes), τ_N is set between one to ten. We report the evaluation results for two settings in Fig. 4 and Fig. 5. We compute the F1 score: $F1 = 2 \times \frac{P \times R}{P+R}$, where P (precision) denotes the number of correctly identified stolen models and R (recall) is the number of stolen models that are correctly identified.

We interpret the results as encouraging: ToSyn can precisely identify IP theft with high accuracy in many settings. We observe that it performs poorly when the value for τ_N is too high or too low. This is because a large τ_N will result in more false negative cases (i.e., treating imitation models as normal models), whereas a small τ_N inclines to treat normal models as imitation models, leading to more false positive cases. Empirically, we recommend proper thresholds $\tau_N = 7$ and $\tau_J = 0.01$. With such thresholds, it is nearly impossible to treat normal models as imitation models, as illustrated in the green lines of Fig. 4 and Fig. 5.

Localizing the Malicious User. As a more challenging task, we evaluate when given the above 15 + 15 imitation models, whether we can localize the UID of the specific malicious users. When checking each suspect model, we have already identified all "activated" WM passes via statistical tests. To evaluate the localization accuracy, for each imitation model, we use its activated WM passes to form a "fingerprint", denoting a vector v of N bits ($N = 34$ in our setting) where "1" denotes the presence of an activated WM pass. Then, v is compared with the assigned WM passes $\mathcal{P}_i$ of N users (the ground truth user is within this N user); a closer cosine distance means a higher probability that v matches the specific user. We return the top-1 user with the lowest cosine distance as the localized "malicious" user.

Table 5: Average top-1 accuracy in localizing malicious users.

Setting	Number of Total Users N				
CodeT5	15	15 + 1000	15 + 2000	15 + 5000	15+ 50000
top-1 Accuracy	1.0	1.0	1.0	1.0	1.0
CodeBERT	15	15 + 1000	15 + 2000	15 + 5000	15+ 50000
top-1 Accuracy	0.93	0.93	0.93	0.93	0.92

Table 5 reports the evaluation results (average top-1 accuracy) for all 15 + 15 imitation models. We consider different settings, where we consider a small group of only 15 malicious users (among which one is the ground truth), or larger groups including 15 malicious users and randomly sampled normal users. Overall, although localizing specific adversaries are generally challenging, we interpret the evaluation results as promising, given that the true malicious user can be accurately localized within top-1 among most settings.

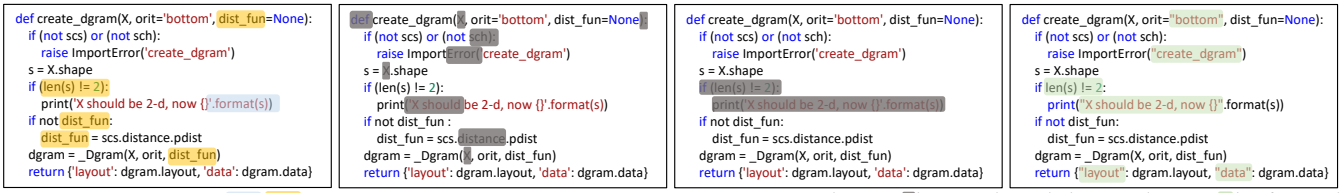
While we aim to assign a unique set of 10 to 20 WM passes to each user (meaning ToSyn's capacity is about 15B users), it is *not* necessary to cross-compare such a large N. In reality, investigators can first check which users have issued a huge number of queries (e.g., over 10K), and then only compare with those "suspicious users" to localize the specific malicious user. This can effectively reduce the search space to a few thousand likely model stealers.

8.4 RQ4: Robustness

This section studies the robustness of ToSyn, which quantifies the difficulty of launching WM removing attacks toward ToSyn-inserted WMs. As already clarified in Sec. 6.1, we consider two types of attackers, i.e., "partially-knowledgeable" (PK) attackers and "fully knowledgeable" (FK) attackers. PK attackers are aware that WM techniques have been integrated by M_{vic}. Nevertheless, they do not know the WM schemes proposed in this work. In contrast, FK attackers know all the WM schemes designed in this work (but do not know the exact WM passes). We now elaborate on settings and results for PK and FK attackers.

PK Attackers. The PK attackers face a typical "black-box" challenge, and therefore, it is reasonable to assume that they resort to heuristics to recognize WMs in LLCG outputs. Inspired by recent research in model extraction attack towards LLCG [57], we consider two representative WM recognition methods. Following a state-of-the-art work [45], we first employ a popular WM recognition technique for textual content, named ONION [75]. This technique uses the drop of perplexity (ppl) score provided by an LLM (e.g., GPT2 [77]) as an indicator to recognize the potential WM words. To be specific, given an LLCG output, as a sequence y of tokens, ONION iteratively deletes each token in y, and checks if deleting certain tokens can cause a decrease in the ppl score. These "sensitive" tokens would be recorded. Then top-ten most sensitive tokens would be returned as WM words and dislodged from y. It should be noted that the effectiveness of ONION is significantly affected by the choice of backbone LLM, as the ppl score may vary significantly across different LLMs.

We also use anomaly code recognition techniques to detect WMs. While "anomaly code" may be vaguely defined, we consider a practical, automated and scalable setting, where we use a popular static



```
def create_dgram(X, orit='bottom', dist_fun=None):
    if (not scs) or (not sch):
        raise ImportError('create_dgram')
    s = X.shape
    if (len(s) != 2):
        print('X should be 2-d, now {}'.format(s))
    if not dist_fun:
        dist_fun = scs.distance.pdist
    dgram = _Dgram(X, orit, dist_fun)
    return {'layout': dgram.layout, 'data': dgram.data}
```

(a) Output with watermark BSR CST .

```
def create_dgram(X, orit='bottom', dist_fun=None):
    if (not scs) or (not sch):
        raise ImportError('create_dgram')
    s = X.shape
    if (len(s) != 2):
        print('X should be 2-d, now {}'.format(s))
    if not dist_fun :
        dist_fun = scs.distance.pdist
    dgram = _Dgram(X, orit, dist_fun)
    return {'layout': dgram.layout, 'data': dgram.data}
```

(b) ONION-removed outputs. is removed.

```
def create_dgram(X, orit='bottom', dist_fun=None):
    if (not scs) or (not sch):
        raise ImportError('create_dgram')
    s = X.shape
    if (len(s) != 2):
        print('X should be 2-d, now {}'.format(s))
    if not dist_fun:
        dist_fun = scs.distance.pdist
    dgram = _Dgram(X, orit, dist_fun)
    return {'layout': dgram.layout, 'data': dgram.data}
```

(c) Semgrep-removed outputs. is removed.

```
def create_dgram(X, orit="bottom", dist_fun=None):
    if (not scs) or (not sch):
        raise ImportError("create_dgram")
    s = X.shape
    if len(s) != 2:
        print("X should be 2-d, now {}".format(s))
    if not dist_fun:
        dist_fun = scs.distance.pdist
    dgram = _Dgram(X, orit, dist_fun)
    return {"layout": dgram.layout, "data": dgram.data}
```

(d) Black-removed outputs. is reformatted.

Figure 6: Various examples of WM removing attacks.

code security analyzer, Semgrep [15], to evaluate the chance of flagging abnormal code patterns in WM-injected LLCG outputs. Semgrep is a commercial, query-based static security analyzer. It has 139 rules to detect a variety of risks in code. Ideally, adversaries can use Semgrep to flag suspicious code snippets, and further investigate the flagged code snippets to decide if WMs are injected. At this step, we run all 139 rules for each WM-injected LLCG output, and check if the reported suspicious code snippets are indeed WMs.

We illustrate some removal examples in Fig. 6, which reports the WM-injected LLCG output (Fig. 6(a)), the ONION-removed LLCG output (Fig. 6(b)), and the Semgrep-removed LLCG output ((Fig. 6(c))). It is clear that ONION is impractical if the attackers would employ it in real-world scenarios, as it brutally deletes tokens in generated content, which breaks the code both semantically and syntactically. For example, it identifies the Python keyword "def" as a potential WM-injected token and removes it. This is less acceptable, as code with obviously broken syntax and unclear specifications are not useful. Semgrep, on the other hand, removes potential WMs in a more acceptable manner, as it ensures that the code remains syntactically correct. However, even with hundreds of rules, it still fails to detect all injected WMs, as noted below.

Table 6: WM recognition evaluation using PK attackers.

WM Recognition Method	Time (s)	Precision	Recall	F1-Score
ONION-GPT2	1514.81	0.012	0.029	0.017
ONION-CodeGPT	1370.02	0.007	0.028	0.011
Semgrep-Default	45.83	0.093	0.002	0.004
Semgrep-FK	101.41	0.429	0.116	0.183

Table 6 reports the results, where we setup ONION with two LLM backbones (GPT2 and CodeGPT [62]). To clarify, CodeGPT is a Transformer-based model pretrained on multiple programming languages and performs better than GPT2 on code generation tasks. Both models are used to compute the ppl scores. We prepare 1K LLCG outputs with WM injected by ToSyn, and we measure the accuracy in terms of precision, recall, and F1-Score. Here, precision denotes the number of correctly identified WM tokens, and recall is the number of WM tokens that are correctly identified. F1-score is computed over precision and recall, accordingly. The column "Time" measures the time cost for processing all 1K outputs with each recognition technique.

From Table 6, it is clear that none of the WM recognition settings is accurate. ONION takes on average 1.4 seconds to analyze one LLCG output, since it uses GPU to infer and collect ppl scores. Even worse, we find that ONION flags many irrelevant words as "suspicious", which are not the real WM words, and results in low "Recall" and "Precision." For instance, Onion-GPT2 flags 14,361 tokens as potentially "suspicious," while only 173 of these are genuinely related to our watermarking schemes, resulting in a low "Precision" of 0.012. Similarly, Onion-CodeGPT has a "Precision" of 0.007, as only 161 out of the 24,291 flagged tokens are actually related. We attribute this due to the fact that ONION is designed to recognize WMs in NL text, which are often "context-free," obviously abnormal words. Given that our WM passes blend in the code context well (also reflected in **RQ5**; see Sec. 8.5), it is unsurprising that ONION is inaccurate.

The 4th row in Table 6 reports the results of using Semgrep to flag "abnormal" tokens that may represent WMs. Although Semgrep is much faster than ONION, the recognition accuracy is still low. During the evaluation, we find that Semgrep is conservative and it only flags a negligible number of "abnormal" tokens. This illustrates the stealthiness of ToSyn-injected WMs to a reasonable extent, and explains the slightly higher "Precision" yet low "Recall."

Overall, we deem that de facto NLP WM recognition techniques (ONION) and popular code security detectors (Semgrep) with their default settings are not suitable to combat ToSyn. We thus foresee that PK attackers are not effective in detecting and removing WMs, which justifies the robustness of ToSyn.

Alternative Removal Solution. Besides watermarking detectors and anomaly code recognition techniques, an intuitive method is to employ code formatters to "regulate" LLCG outputs and thus remove WMs. To this end, we use black [40], a mature Python code formatter maintained by the Python Software Foundation with more than 386 million downloads. To be specific, we compare the code before and after reformatting to check whether the WM has been successfully removed in the formatted Python code.

Fig. 6 (d) shows an example where the code formatter converts all strings in single quotes to double quotes and removes the parentheses around the `if` condition statement. Code formatters like black focus on reformatting code to conform to a specific coding style, such as PEP 8. One advantage is that the reformatted code remains compilable and semantically equivalent, which has little impact on users' experience. However, we notice that black can only detect one type of pass in the CST scheme among all 34 WMs, making it impractical as a watermark removal tool.

Table 7: WM removal by FK attackers. We show the BLEU scores of the imitation models M_{imi}, when the FK attackers have removed tokens that may be mutated by ToSyn.

	CodeBERT	CodeT5
Original (w/o WM)	24.56	23.23
WM Removed	15.91	14.88

FK Attackers. As aforementioned, an FK attacker is assumed to have the knowledge of all six WM schemes. Nevertheless, we assume he does not know the instantiated WM passes, given there exist a substantial number of possible WM passes that can be derived from the WM schemes. Thus, in his received LLCG outputs, he does not know the tokens that have been mutated.

To recognize potential WMs in a large amount of LLCG outputs, we assume the attackers would write a set of customized rules to detect the potential WMs automatically. Therefore, we provide the special Semgrep rules for all six WM schemes and imitate the WM recognition process. The results are demonstrated in the last row of Table 6. With the given knowledge, both the recall and precision are well-improved compared to the PK scenario. However, the F1-score (0.183) is still unpromising. More importantly, Table 7 further shows that the BLEU scores of the obtained imitation models (trained over WM-removed LLCG outputs) are largely downgraded. That is, FK attackers are trapped in a vicious cycle: they have to remove potential WMs to impede ToSyn's IP check, but the obtained imitation models exhibit low performance and bring little benefit. We thus believe that ToSyn manifests a strong robustness against WM removing attacks launched by FK attackers.

8.5 RQ5: Stealthiness

Stealthiness reflects how ToSyn-mutated tokens can blend into the normal code. This metric indicates the difficulty in distinguishing WM-injected code from normal code, thus reflecting the difficulty in recognizing the usage of WMs in LLCG outputs.

We measure the stealthiness by comparing the "distance" of normal code snippets and their ToSyn-mutated versions. The distances are computed regarding two aspects: the syntax and the structure. For the syntax aspect, we feed each Python code into `pygment` [13] and count the number of different built-in token types. For example, given a Python code snippet `def sort(): pass`, two keywords (`def` and `pass`), one function name (`sort`), and several punctuations are extracted as the "built-in" tokens. We compute the syntax-level distance in the following way:

$$distance = \frac{\sum_{i=1}^{m} abs(ttype_i^o - ttype_i^w)}{\sum_{i=1}^{m} ttype_i^o} \quad (1)$$

where m is the total different number of built-in token types supported by `pygment`, while $ttype_i^o$ and $ttype_i^w$ represent the number of build-in token type i on a code snippet and its mutated version, respectively. A lower distance indicates better stealthiness.

We use `pycode-similar` [12] to compute structural-level stealthiness. `pycode-similar` is commonly used to calculate Python AST-based code similarity. Let a Python code snippet's AST have n nodes, where m nodes are matched with nodes on the mutated code snippet's AST. `pycode-similar` yields a ratio $\frac{m}{n}$, and we define the structural-level stealthiness as $1 - \frac{m}{n}$.

We report the (cumulative) distribution of both stealthiness metrics in Table 8. For most mutated code snippets (over 96%), structural-level distance is less than 0.1, meaning over 90% of the AST nodes are matched. Considering both syntax- and structure-level metrics, the distance is less than 0.1 for most mutated code snippets (over 98% and 96% for the two metrics). These findings illustrate the high stealthiness of the mutated code snippets.

9 RELATED WORK

Software Watermark. Watermarking techniques are increasingly popular in the digital world as a way of safeguarding against piracy. This trend is particularly evident in the software industry where watermarking techniques have become widely established over the last couple of decades [35, 42, 104]. It is worth noting that software watermarking techniques tend to overlap with software obfuscation techniques since both methods involve mutating the software hierarchy to embed WMs. For instance, basic blocks [34, 76], control graphs [32, 71], and branch conditions [65, 82] can all be manipulated during the watermarking process. Recent works have also explored the usage of chaos encryption in source code [51] or directly into executables [39]. In general, chaos encryption introduces an added layer of complexity that can make it harder for adversaries to access or manipulate the watermarked content.

DNN Watermark. With the threat of IP theft and model stealing, several DNN WM methods are proposed [20, 29, 33, 37, 50, 59, 61, 92, 101, 102]. They allow the model owner to verify his ownership by checking the existence of WMs, usually in the form of secret messages, in a suspect DNN model. Both *black-box* and *white-box* WM techniques have been proposed and constantly improved. The white-box WM techniques embed and verify WMs in model internals, i.e., parameters [29, 37, 61] or activation maps [33, 59]. Typically, embedding a white-box WM requires to re-train the model, with specially-prepared training data samples or new learning objectives (loss functions). In contrast, blackbox WM techniques embed WMs in the model's prediction outputs, usually with respect to a special set of inputs [20, 50, 84, 102]. Our proposed technique falls in the black-box WM technique, which is independent of the underlying model architecture and training process.

10 ETHICS AND RESPONSIBLE DISCLOSURE

Imitation attacks on LLCG models can pose a threat to the intellectual property of model owners. We take several steps to mitigate any possible harm caused by our paper. First, we disclosed all information on the WM scheme design involved in this research project, as we firmly believe that a transparent WM system is essential for the benefit of the entire community. Additionally, we clarify that all experiments related to this project were performed within our lab environment, and we never attempted to attack any real-world system or released any imitation model. Given that we use two backbone models (CodeT5 and CodeBERT) and one publicly-available dataset, we strictly follow their licenses and ensure proper attribution.

We believe that publishing our paper and publicly disclosing the WM schemes are both ethical and responsible. Currently, we have observed only limited cases (see Sec. 1) in which companies were accused of conducting imitation attacks. Our goal with this work is to propose a practical solution for protecting models and to encourage the use of carefully-protected LLCG models in the future.

11 DISCUSSION

Feasibility of LLCG Model Stealing. Our motivation example in Sec. 3.2 demonstrates that conducting an imitation attack on CodeT5 and CodeBERT is highly feasible. We also notice that recent work [57] has set ChatGPT [66] as the target LLCG model and proposed two methods to "remove" the potential WM. On one hand, we believe that their work reinforces the real-world threat of imitation attacks on LLCG models. Thus, applying WM techniques

Table 8: Distribution of structural correlation scores.

Distance		[0, 0.05]	[0.05, 0.1]	[0.1, 0.15]	[0.15, 0.2]	[0.2, 0.9]	[0.9, 1.0]
Syntax	Frequency (%)	90.00	8.12	1.44	0.31	0.13	0.0
	cumulative frequency (%)	90.00	98.12	99.56	99.87	100.0	100.0
Structure	Frequency (%)	88.14	8.13	2.28	0.79	0.66	0.0
	cumulative frequency (%)	88.14	96.27	98.55	99.34	100.0	100.0

to protect LLCG models is demanding and urgent. Nevertheless, in Sec. 8.4, we have indeed evaluated our WM technique against their proposed WM removal methods (and even more, as we have additionally considered code formatter as another possible WM removal method). In Sec. 8.4, we have achieved highly promising results to detect WMs even in front of powerful attackers who actively remove them. Therefore, we believe our WM technique is effective against their proposed WM removal methods.

WM Passes Extension. More WM passes can be instantiated from our WM schemes. For instance, as noted in designing the WM scheme LA (Sec. 5.3), more LA passes can be easily formed by considering other Python packages. Similarly, we can generate more TFR passes by considering additional third-party APIs and built-in functions. Developers can choose packages and APIs from either generic resources or in a way that is tailored to their domain-specific requirements. For example, if the LLCG model is designed to specifically output machine learning code, its model owner could focus on libraries related to scientific computing.

Moreover, another direction for extension is to support other programming languages like Java and C/C++. It is worth noting that, out of 34 WM passes currently implemented in ToSyn, 13 passes do not rely on Python-specific features. Since tree-sitter can support most mainstream languages[7] with a unified interface, extending to other languages with those 13 WM passes are easy. For the remaining 21 WM passes, we believe identifying similar features in other languages is feasible. Although modern programming languages often have divergent design philosophies and intended application domains, many of the fundamental operations and architectural patterns nevertheless remain similar across languages. For instance, similar to the Python "print" function, Java allows printing with either API "System.out.println()" or the built-in Formatter class. As a result, ToSyn should be extensible to more WM passes and support other languages. However, extending ToSyn to less commonly used languages such as Eiffel and Erlang remains challenging. These languages may be outdated with minimal active developer communities. Additionally, some less popular languages are unsupported by tree-sitter.

Semantics Drifting. Each WM pass is associated with its "synonym set". Overall, a considerable number of WM passes (and their accompanied synonym sets) are required to achieve high watermarking robustness. Currently, we design 34 WM passes where for each query, we randomly apply a subset $P_{id} \subset \mathcal{P}$ to mutate LLCG outputs. We admit some of our WM passes cause slight drifting in the code functionality.

For instance, in one LA pass, we replace `np.max` with `max` in the Python math library. While `np.max` can process `np.array`, we note that `max` cannot. Overall, we believe a mild "semantics drifting" is hardly avoidable, given the strict semantics of real-world programming languages like Python. Nevertheless, this should *not* undermine the utility of WM-injected code completion outputs. From a holistic perspective, modern LLCG APIs are *not* championed to replace human programmers; rather, it aids human programmers by suggesting useful code. That is, users may tweak the code received from LLCG, if needed.

Preventing LLCG Model Stealing in an Online Setting. This research proposes a post-verification process (as in Sec. 6) to check if a suspect LLCG model is trained from WM-injected data samples. Another promising and orthogonal direction is to design online schemes to prevent imitation attacks without undermining the experience of normal users. This is an interesting albeit difficult direction, as it is generally challenging to distinguish normal users from attackers. We leave it as one future work to explore techniques to detect and terminate on-going imitation attacks launched toward victim LLCG models.

Availability of Imitation Model. To verify if a suspect LLCG model is derived from imitation attacks, we need to know its owner first. In other words, if Alice launches imitation attacks to steal a model M_{imi}, but releases the model under Bob's name, then it becomes very difficult, if at all possible, to verify the ownership of M_{imi}, as we may not even have Bob's record in our protected LLCG. Moreover, if the attackers do not even publicly release M_{imi} (i.e., M_{imi} is only for private usage), then it becomes infeasible to access and verify the ownership of M_{imi}. We view this as a general limitation in this line of research.

Multi-Account Attack. Each LLCG user is assigned with an N-bit UID. One may hypothesize that the attackers can register multiple accounts, and each of those accounts would get its own UID with a different resulting distribution of watermarked tokens. As a result, the attackers can gather the responses via hundreds of accounts to intermingle the injected WMs. To the best of our knowledge, no previous WM works present such multi-account attacks, and our adversarial setting is aligned with state-of-the-art papers [44, 45]. While we expect that this may compromise our WM, in practice, launching such an attack is likely infeasible since prominent services like GPT-J [36] and ChatGPT [66] have mechanisms to detect bots. Overall, "Bot Detection" is mature in the industry, e.g., used by online retailers.

12 CONCLUSION

We propose for the first time ToSyn to protect LLCG from imitation attacks. ToSyn injects WMs by replacing tokens in LLCG outputs with their synonyms to change the synonym distribution, and it uses statistical tests to verify the ownership of a suspect model and localize malicious users. Evaluation shows that ToSyn can protect LLCG with high fidelity, reliability, robustness, stealthiness, and low cost. We believe that ToSyn can be a useful tool for protecting LLCG models in reality.

[7] Treesitter version 0.20.7 now supports 113 different programming languages through 14 distinct language bindings.

REFERENCES

[1] [n. d.]. billing for AI21 LAB. https://studio.ai21.com/pricing.
[2] [n. d.]. billing for aixcoder. shorturl.at/evxzA.
[3] [n. d.]. billing for amazon p2. https://aws.amazon.com/ec2/instance-types/p2/.
[4] [n. d.]. billing for GitHub Copilot. https://docs.github.com/en/billing/managing-billing-for-github-copilot/about-billing-for-github-copilot.
[5] [n. d.]. Codex. https://openai.com/blog/openai-codex/.
[6] [n. d.]. Copilot. https://github.com/features/copilot.
[7] [n. d.]. deeptabnine. shorturl.at/bS049.
[8] [n. d.]. Google Bard steal. https://www.theinformation.com/articles/alphabets-google-and-deepmind-pause-grudges-join-forces-to-chase-openai.
[9] [n. d.]. JensenShannon divergence. shorturl.at/hAGK2.
[10] [n. d.]. jurassic. https://www.ai21.com/studio.
[11] [n. d.]. OpenAI Watermarking prototype. https://scottaaronson.blog/?p=6823.
[12] [n. d.]. pycode-similar. https://pypi.org/project/pycode-similar/.
[13] [n. d.]. Pygment. https://pygments.org/.
[14] [n. d.]. PyPI Website. https://pypi.org/.
[15] [n. d.]. Semgrep. https://semgrep.dev/.
[16] [n. d.]. Tabnine. https://www.tabnine.com/.
[17] [n. d.]. Tabnine-with-Copilot. https://www.tabnine.com/blog/tabnine-vs-github-copilot/.
[18] [n. d.]. Top ten Python Package. https://interviewbit.com/blog/python-libraries/.
[19] [n. d.]. Top twenty six Python Package. https://mygreatlearning.com/blog/open-source-python-libraries/.
[20] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. 2018. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *27th USENIX Security Symposium (USENIX Security 18)*. 1615–1631.
[21] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).
[22] Artifact. 2022. ToSyn. https://sites.google.com/view/tosyn.
[23] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. *arXiv preprint arXiv:2206.01335* (2022).
[24] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[25] Max Brunsfeld. [n. d.]. Tree-sitter. https://github.com/tree-sitter/tree-sitter.
[26] Huili Chen, Bita Darvish Rouhani, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. 2019. DeepMarks: A Secure Fingerprinting Framework for Digital Rights Management of Deep Learning Models *(ICMR '19)*. New York, NY, USA. https://doi.org/10.1145/3323873.3325042
[27] Jialuo Chen, Jingyi Wang, Tinglan Peng, Youcheng Sun, Peng Cheng, Shouling Ji, Xingjun Ma, Bo Li, and Dawn Song. 2022. Copy, Right? A Testing Framework for Copyright Protection of Deep Learning Models. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 824–841.
[28] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[29] Xuxi Chen, Tianlong Chen, Zhenyu Zhang, and Zhangyang Wang. 2021. You are caught stealing my winning lottery ticket! Making a lottery ticket claim its ownership. *Advances in Neural Information Processing Systems* 34 (2021), 1780–1791.
[30] Yufei Chen, Chao Shen, Cong Wang, and Yang Zhang. 2022. Teacher Model Fingerprinting Attacks Against Transfer Learning. In *31st USENIX Security Symposium (USENIX Security 22)*.
[31] Yufei Chen, Chao Shen, Cong Wang, and Yang Zhang. 2022. Teacher model fingerprinting attacks against transfer learning. In *31st USENIX Security Symposium (USENIX Security 22)*. 3593–3610.
[32] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. *On the limits of software watermarking*. Technical Report. Citeseer.
[33] Bita Darvish Rouhani, Huili Chen, and Farinaz Koushanfar. 2019. Deepsigns: An end-to-end watermarking framework for ownership protection of deep neural networks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 485–497.
[34] 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.
[35] Ayan Dey, Sukriti Bhattacharya, and Nabendu Chaki. 2019. Software watermarking: Progress and challenges. *INAE Letters* 4, 1 (2019), 65–75.
[36] EleutherAI. [n. d.]. GPT-J. https://huggingface.co/EleutherAI/gpt-j-6B.
[37] Lixin Fan, Kam Woh Ng, Chee Seng Chan, and Qiang Yang. 2021. Deepip: Deep neural network intellectual property protection with passports. *IEEE Transactions on Pattern Analysis & Machine Intelligence* 01 (2021), 1–1.
[38] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP Findings*.
[39] Liu Fenlin, Lu Bin, and Luo Xiangyang. 2006. A chaos-based robust software watermarking, information security practice and experience. (2006).
[40] Python Software Foundation. [n. d.]. black. https://github.com/psf/black.
[41] Jia Guo and Miodrag Potkonjak. 2018. Watermarking deep neural networks for embedded systems. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018*, Iris Bahar (Ed.). ACM, 133.
[42] James Hamilton and Sebastian Danicic. 2011. A survey of static software watermarking. In *2011 World Congress on Internet Security (WorldCIS-2011)*. IEEE, 100–107.
[43] Hanieh Hashemi, Yongqin Wang, and Murali Annavaram. 2021. DarKnight: An Accelerated Framework for Privacy and Integrity Preserving Deep Learning Using Trusted Hardware. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 212–224.
[44] Xuanli He, Qiongkai Xu, Lingjuan Lyu, Fangzhao Wu, and Chenguang Wang. 2022. Protecting Intellectual Property of Language Generation APIs with Lexical Watermark. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*. AAAI Press, 10758–10766.
[45] Xuanli He, Qiongkai Xu, Yi Zeng, Lingjuan Lyu, Fangzhao Wu, Jiwei Li, and Ruoxi Jia. 2022. CATER: Intellectual Property Protection on Text Generation APIs via Conditional Watermarks. *arXiv preprint arXiv:2209.08773* (2022).
[46] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the Knowledge in a Neural Network. *CoRR* abs/1503.02531 (2015). arXiv:1503.02531 http://arxiv.org/abs/1503.02531
[47] Jiahui Hou, Huiqi Liu, Yunxin Liu, Yu Wang, Peng-Jun Wan, and Xiang-Yang Li. 2021. Model Protection: Real-time Privacy-preserving Inference Service for Model Privacy at the Edge. *IEEE Transactions on Dependable and Secure Computing* (2021).
[48] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. 2020. DeepSniffer: A DNN Model Extraction Framework Based on Learning Architectural Hints. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, 385–399.
[49] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588* (2018).
[50] Hengrui Jia, Christopher A Choquette-Choo, Varun Chandrasekaran, and Nicolas Papernot. 2021. Entangled watermarks as a defense against model extraction. In *30th USENIX Security Symposium (USENIX Security 21)*. 1937–1954.
[51] Yin Ke-xin, Yin Ke, and Zhu Jian-qi. 2009. A robust dynamic software watermarking. In *2009 International Conference on Information Technology and Computer Science*, Vol. 1. IEEE, 15–18.
[52] John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. 2023. A watermark for large language models. *arXiv preprint arXiv:2301.10226* (2023).
[53] Kalpesh Krishna, Gaurav Singh Tomar, Ankur P Parikh, Nicolas Papernot, and Mohit Iyyer. 2019. Thieves on sesame street model extraction of bert-based apis. *arXiv preprint arXiv:1910.12366* (2019).
[54] Klemens Lagler, Michael Schindelegger, Johannes Böhm, Hana Krásná, and Tobias Nilsson. 2013. GPT2: Empirical slant delay model for radio space geodetic techniques. *Geophysical research letters* 40, 6 (2013), 1069–1073.
[55] Erwan Le Merrer, Patrick Perez, and Gilles Trédan. 2020. Adversarial frontier stitching for remote neural network watermarking. *Neural Computing and Applications* 32, 13 (2020), 9233–9244.
[56] Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Unleashing the Power of Compiler Intermediate Representation to Enhance Neural Program Embeddings. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2253–2265.
[57] Zongjie Li, Chaozheng Wang, Pingchuan Ma, Chaowei Liu, Shuai Wang, Daoyuan Wu, and Cuiyun Gao. 2023. On the Feasibility of Specialized Ability Extracting for Large Language Code Models. *arXiv preprint arXiv:2303.03012* (2023).
[58] Zheng Li and Yang Zhang. 2021. Membership Leakage in Label-Only Exposures. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. ACM, 880–895.
[59] Jian Han Lim, Chee Seng Chan, Kam Woh Ng, Lixin Fan, and Qiang Yang. 2022. Protect, show, attend and tell: Empowering image captioning models with ownership protection. *Pattern Recognition* 122 (2022), 108285.

[60] Aiwei Liu, Leyi Pan, Xuming Hu, Shu'ang Li, Lijie Wen, Irwin King, and Philip S Yu. 2023. A Private Watermark for Large Language Models. *arXiv preprint arXiv:2307.16230* (2023).

[61] Hanwen Liu, Zhenyu Weng, and Yuesheng Zhu. 2021. Watermarking Deep Neural Networks with Greedy Residuals.. In *ICML*. 6978–6988.

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

[63] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. 2021. PPFL: privacy-preserving federated learning with trusted execution environments. In *MobiSys '21: The 19th Annual International Conference on Mobile Systems, Applications, and Services, Virtual Event, Wisconsin, USA, 24 June - 2 July, 2021*. ACM, 94–108.

[64] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. 2020. DarkneTZ: towards model privacy at the edge using trusted execution environments. In *MobiSys '20: The 18th Annual International Conference on Mobile Systems, Applications, and Services, Toronto, Ontario, Canada, June 15-19, 2020*. ACM, 161–174.

[65] Jasvir Nagra, Clark Thomborson, and Christian Collberg. 2002. A functional taxonomy for software watermarking. In *ACSC*, Vol. 4. 177–186.

[66] OpenAI. [n. d.]. ChatGPT. https://openai.com/blog/chatgpt/.

[67] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. 2019. Knockoff Nets: Stealing Functionality of Black-Box Models. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 4954–4963.

[68] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. 2019. Knockoff Nets: Stealing Functionality of Black-Box Models. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 4954–4963.

[69] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. 2020. Prediction Poisoning: Towards Defenses Against DNN Model Stealing Attacks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

[70] Soham Pal, Yash Gupta, Aditya Shukla, Aditya Kanade, Shirish K. Shevade, and Vinod Ganapathy. 2019. A framework for the extraction of Deep Neural Networks by leveraging public data. *CoRR* abs/1905.09165 (2019).

[71] Jens Palsberg, Sowmya Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, and Yi Zhang. 2000. Experience with software watermarking. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*. IEEE, 308–316.

[72] Kishore Papineni, Salim Roukos, Todd Ward, and Wei jing Zhu. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation. 311–318.

[73] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. *arXiv preprint arXiv:2108.11601* (2021).

[74] Python. 2021. The Python Language Reference. https://docs.python.org/3/reference/index.html.

[75] Fanchao Qi, Yangyi Chen, Mukai Li, Yuan Yao, Zhiyuan Liu, and Maosong Sun. [n. d.]. ONION: A Simple and Effective Defense Against Textual Backdoor Attacks. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*.

[76] Gang Qu and Miodrag Potkonjak. 1998. Analysis of watermarking techniques for graph coloring problem. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*. 190–193.

[77] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[78] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. 2022. DeepSteal: Advanced Model Extractions Leveraging Efficient Weight Stealing in Memories. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 1157–1174.

[79] Alexander Schlögl and Rainer Böhme. 2020. eNNclave: Offline Inference with Model Confidentiality. In *AISec@CCS 2020: Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security, Virtual Event, USA, 13 November 2020*. ACM, 93–104.

[80] Tianxiang Shen, Ji Qi, Jianyu Jiang, Xian Wang, Siyuan Wen, Xusheng Chen, Shixiong Zhao, Sen Wang, Li Chen, Xiapu Luo, et al. 2022. SOTER: Guarding Black-box Inference for General Neural Networks at the Edge. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 723–738.

[81] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, 955–970.

[82] Julien P Stern, Gael Hachez, Francois Koeune, and Jean-Jacques Quisquater. 1999. Robust object watermarking: Application to code. In *International Workshop on Information Hiding*. Springer, 368–378.

[83] Zhichuang Sun, Ruimin Sun, Long Lu, and Somesh Jha. 2020. ShadowNet: A Secure and Efficient System for On-device Model Inference. *CoRR* abs/2011.05905 (2020).

[84] Sebastian Szyller, Buse Gul Atli, Samuel Marchal, and N Asokan. 2021. Dawn: Dynamic adversarial watermarking of neural networks. In *Proceedings of the 29th ACM International Conference on Multimedia*. 4417–4425.

[85] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. Efficient transformers: A survey. *ACM Computing Surveys (CSUR)* (2020).

[86] Florian Tramèr and Dan Boneh. 2019. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *ICLR*.

[87] Eric Wallace, Mitchell Stern, and Dawn Song. 2020. Imitation attacks and defenses for black-box machine translation systems. *arXiv preprint arXiv:2004.15015* (2020).

[88] Bolun Wang, Yuanshun Yao, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. 2018. With Great Training Comes Great Vulnerability: Practical Attacks against Transfer Learning. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 1281–1297.

[89] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 382–394.

[90] Jialai Wang, Han Qiu, Yi Rong, Hengkai Ye, Qi Li, Zongpeng Li, and Chao Zhang. 2022. BET: black-box efficient testing for convolutional neural networks. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 164–175.

[91] Tianhao Wang and Florian Kerschbaum. 2019. Robust and undetectable white-box watermarks for deep neural networks. *arXiv preprint arXiv:1910.14268* 1, 2 (2019).

[92] Tianhao Wang and Florian Kerschbaum. 2021. Riga: Covert and robust white-box watermarking of deep neural networks. In *Proceedings of the Web Conference 2021*. 993–1004.

[93] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[94] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. [n. d.]. Code Generation as a Dual Task of Code Summarization. *CoRR* abs/1910.05923 ([n. d.]).

[95] Wiki. [n. d.]. Salt. https://en.wikipedia.org/wiki/Salt_(cryptography).

[96] Yecheng Xiang, Yidi Wang, Hyunjong Choi, Mohsen Karimi, and Hyoseung Kim. 2021. AegisDNN: Dependable and Timely Execution of DNN Tasks with SGX. In *42nd IEEE Real-Time Systems Symposium, RTSS 2021, Dortmund, Germany, December 7-10, 2021*. IEEE, 68–81.

[97] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3520312.3534862

[98] Qiongkai Xu, Xuanli He, Lingjuan Lyu, Lizhen Qu, and Gholamreza Haffari. 2021. Beyond model extraction: Imitation attack for black-box nlp apis. *arXiv preprint arXiv:2108.13873* (2021).

[99] Qiongkai Xu, Xuanli He, Lingjuan Lyu, Lizhen Qu, and Gholamreza Haffari. 2022. Student Surpasses Teacher: Imitation Attack for Black-Box NLP APIs. In *Proceedings of the 29th International Conference on Computational Linguistics*. International Committee on Computational Linguistics, Gyeongju, Republic of Korea, 2849–2860. https://aclanthology.org/2022.coling-1.251

[100] Sangwon Yu, Jongyoon Song, Heeseung Kim, Seongmin Lee, Woo-Jong Ryu, and Sungroh Yoon. 2022. Rare Tokens Degenerate All Tokens: Improving Neural Text Generation via Adaptive Gradient Gating for Rare Token Embeddings. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 29–45.

[101] Jie Zhang, Dongdong Chen, Jing Liao, Weiming Zhang, Gang Hua, and Nenghai Yu. 2020. Passport-aware normalization for deep model protection. *Advances in Neural Information Processing Systems* 33 (2020), 22619–22628.

[102] Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, Marc Ph Stoecklin, Heqing Huang, and Ian Molloy. 2018. Protecting intellectual property of deep neural networks with watermarking. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 159–172.

[103] Jialong Zhang, Zhongshu Gu, Jiyong Jang, Hui Wu, Marc Ph. Stoecklin, Heqing Huang, and Ian M. Molloy. 2018. Protecting Intellectual Property of Deep Neural Networks with Watermarking. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim (Eds.). ACM, 159–172.

[104] William Zhu, Clark Thomborson, and Fei-Yue Wang. 2005. A survey of software watermarking. In *International Conference on Intelligence and Security Informatics*. Springer, 454–458.